ВВЕДЕНИЕ В ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Арвинд Кумар Бансал

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Издательство CRC Press
Издательская группа Taylor & Francis
6000 Брокен Саунд Парквей NW, Сьют 300 Бока Рэйтон FL 33487-2742

Глава 1

Введение

Компьютерные науки стали все более захватывающими со времен своего появления в конце 1950-х годов.

То, что было задумано как инструмент для крупномасштабных научных вычислений в момент его появления, в настоящее время используется во всех сферах нашей жизни, в том числе в медицине, при освоении космоса, в телекоммуникации, при обмене информацией и удаленном сотрудничестве, моделировании, автоматизированном проектировании и навигации, автоматизированном производстве, операциях, при планировании, проектировании, как средства повышения производительности, для электронных сделок и торговли, на транспорте и в управлении электростанций. Сейчас мы даже не можем представить нашу жизнь без использования встраиваемого или персонального компьютера. Компьютеры встроены во многие современные гаджеты, такие как автомобили, мобильные телефоны, самолеты, космические аппараты, высококачественные стиральные и сушильные машины, печи и системы домашней безопасности. Сейчас мы говорим об Умных Домах, которые будут иметь встроенные компьютеры для обработки информации в режиме многозадачности и заботиться о множестве каждодневных задач в доме.

За всей этой компьютерной деятельностью стоят толковые умы программистов, которые решают большое количество комплексных проблем и представляют пути их решения, используя инструкции высокого уровня. Эти инструкции переводятся в машинные инструкции низкого уровня автоматизированными переводчиками. Такие инструкции низкого уровня приводят в исполнение компьютерные программы. Инструкции на уровне механизмов ограничены в своей выразительной силе и не способствуют моделированию и модификации высокоуровневых решений с изменением требований. *Очевидно, что существует*

потребность в высокоуровневых языках и языковых конструкциях для выражения решений для сложных проблем и изменения этих решений по мере изменения требований. Эти требования начинают существовать в результате реорганизации общества, вызванной технических прогрессом. Это требует дополнительного совершенствования и урегулированного изменения программного обеспечения так, чтобы изменение одной его части не оказывало отрицательного воздействия на другую. Для того, чтобы локализовать эффекты, программное обеспечение должно быть модульным: различные модули с четко определенной, не перекрывающейся функциональностью. Эти модули слабо связаны с ограниченным обменом внутренними данными и операциями.

Современные проблемы комплексны и для решения этих проблем требуются сотни тысяч строк инструкций. Разработка таких решений-задача не простая и она включает в себя множество человеко-лет: количество программистов, умноженное на время в годах. Такие усилия означают крупномасштабные обязательства организационных и финансовых ресурсов. Эти усилия не могут быть продублированы и должны быть дополнены эволюционной модификацией и программном обеспечении. В противном случае, задержка в развитии и стоимость разработки программного обеспечения будут непозволительно огромными.

1.1 МНОЖЕСТВО ПРЕДМЕТНЫХ ОБЛАСТЕЙ

В эпоху современности, проблемы, решаемые компьютерами, лежат в различных предметных областях, таких как научные вычисления, обработка текста, программирования баз данных, бизнес-приложения, системное программирование, автоматизация технологических процессов, интеллектуальные системы, веб-приложения и обработка данных в реальном времени. Все эти области сильно отличаются друг от друга различными требованиями.

Примером научных вычислений является рассуждение о вселенной, где существуют миллиарды звезд. Для того, чтобы судить об их взаимодействии друг с другом и с нашей солнечной системой, нам нужно обрабатывать и анализировать данные из компьютеризированных телескопов, которые получают триллионы байт данных. Другим примером научных вычислений является моделирование и отслеживание атмосферных явлений, таких как

торнадо, которые вызывают разрушения в прибрежных районах. Еще один пример научных вычислений- обработка, анализ и отслеживание сейсмической активности на земле. Обработка научных данных и разработка полезных моделей для решения научных проблем требует возможности выделять и обрабатывать большие матрицы в компьютере. Числа должны быть обработаны очень точно. Например, вычисление траектории космического корабля требует точных расчетов и высокой точности.

Примерами обработки текста являются текстовые процессоры и средства повышения производительности, которые мы используем каждый день при подготовке презентация и написании писем. Проблемы такого рода требуют возможности для представления и обработки большого количества строк, рисунков, таблиц, видеоклипов и других мультимедийных объектов в эффективном ключе. Хотя здесь и требуется большой объем вычислений, тем не менее, вычислительная потребность не так велика, как в области научных вычислений. Однако, этот процесс требует более тесного взаимодействия с людьми и предполагает удобство для пользователя.

Программирование баз данных требует организации, обработки и поиска огромного объема данных в логическом порядке, так что они могут быть доступны сразу, без необходимости дублирования. Примеры программирования баз данных в режиме реального времени, это: 1) обработка данных студента в офисе регистратора и 2) считывание кредитной карты в автозаправке. Когда студент приходит в офис регистратора, он предоставляет свой *студенческий билет* и бухгалтер сразу же должна точно знать информацию обо всех платежах студента, его адресе проживания и о изучаемых им предметах.

Бизнес-приложения требуют обширной генерации отчетов, которые могут быть представлены для клиентов и руководителей высшего звена в удобной для использования форме. Бизнес-приложения так же требуют возможности для интеграции программирования баз данных с удобной для пользователя системой отчета. Однако частично процесс обработки бизнес-приложений может быть выполнен в *пакетном режиме*; при этом обработка в реальном времени не всегда для этого требуется.

Системное программирование требует обработки нескольких процессов, выполняющихся одновременно, обеспечивая виртуальную среду для повышения производительности труда программистов, а также обеспечивая взаимодействие с программированием низкого уровня для повышения эффективности выполнения задачи. Системное

программирование требует пошаговой работы памяти компьютера, обработки ошибок, выдачи предупреждений в случае сбоя процесса и взаимодействие вызовов пользовательского уровня с вызовами систем высокого уровня.

Обработка в режиме реального времени требует сбора и обработки данных в настоящий момент. Любая работа по поддержанию счета не должна замедлять сбор или обработку данных, в противном случае события, происходящие в реальном времени, могут быть потеряны с катастрофическими результатами. Например, если компьютер на АЭС реагирует медленно, перегревая систему, то сама АЭС может быть повреждена. Если же медленно реагирует бортовой компьютер самолета-истребителя, этот самолет может быть сбит вражескими ракетами. Эти примеры доказывают необходимость обеспечения высокого уровня приоритетности задачи в реальном времени и принятия решения немедленно, в режиме реального времени для облегчения сбора и обработки событий.

В последние годы интеллектуальные системы были размещены во многих сферах жизни, таких как работа промышленных роботов, планирование цехов, планирование работы аэропорта, обработка естественного языка и даже игры, такие как шахматы, которые могут победить гроссмейстеров и проводить анализ изображений. Эти системы должны работать в очень большом пространстве с множеством проблем и меть возможность логически анализировать проблему и грамотно предлагать пути ее решения. Как правило, при решении этих проблемы руководствуются эвристическим программированием-звуковым обоснованным предположением на основе математического моделирования, направленного на быстрое решение проблемы. Проблема усложняется из-за присущей неопределенности в поведении феномена реального мира и отсутствия полной информации.

В последние годы, веб-программирование в совокупности с мультимедийными системами приводит к различным требованиям в языках программирования. Программы должны быть доступны для поиска с удаленных сайтов (URLs) и выполняться на локальных машинах. В то время как извлечение информации из удаленных машин требует преобразования структур данных в строки и наоборот, эффективное выполнение требует эффективного перевода в инструкции низкого уровня. К сожалению, перевод и выполнение веб-программ происходит одновременно, замедляя выполнение программ и нуждается

в компиляции «на лету» (JIT-компиляция), чтобы ускорить обработку. В будущем, мы увидим предметно-ориентированные языки для обработки различных доменов.

1.2 МОТИВАЦИЯ

Учитывая набор требований в предметной области, программы могут быть разработаны для автоматизирования процесса. Тем не менее, связь решений для компьютеров сложна, поскольку компьютерам не достаточно интеллекта, чтобы понять намерения человека, а так же компьютер не может исправить человеческие непреднамеренные ошибки и отсутствие возможности для понимания последствий, скрытых за нашими сообщениями. Все должно быть смоделировано и передано в доступном виде на компьютеры. Субъекты должны быть смоделированы абстрактно и точно, решение должно быть построено пошагово, без учета какой-либо двусмысленности.

Язык программирования является организованным способом общения с компьютером, таким образом компьютер выполняет задания точно в соответствии с инструкциями, заданными программистом. Инструкции могут использовать любые средства массовой информации, такие как текстовые, визуальные, знаки, жесты, аудио или же любые их комбинации для определения решений. Однако, используются следующие важные критерии:

1. Решение проблемы должно быть просто и полностью выраженным.
2. Должен быть потенциал для развития спецификации решения.
3. Введение в языки программирования. Должен быть индивидуальный однозначный перевод между намерениями программиста и действиями, предпринятыми компьютером. Поскольку решения определены на высоком уровне, существует возможность недопонимания между программистом и компьютером в случае недостаточности однозначных конструкций для выражения решения.

С повышением уровня автоматизации, общество перестраивает себя, поглощая уровень автоматизации, а также разрабатывая новые предмет-

ные области. Например, в 1950х, когда компьютеры были изобретены, важным требованием являлось наличие научных вычислений. Требования нарастали с большой скоростью для обработки текста, для инструментов для повышения производительности, графического дизайна, автоматизации бизнеса, интеллектуальных систем, веб-транзакций и веб-сотрудничества.

Развитие технологий, социальной реструктуризации и развитие предметных областей пересекаются. По мере развития технологий, растут и требования, при этом решение проблемы становится все более запутанным. Взаимодействуя, эти решения требуют более доступного человеческого языка, нежели компьютерного, языка программирования, который может быть автоматически переведен в инструкции низкого уровня для лучшего обслуживания программного обеспечения. Языки на основе Интернета и веб-программирование существуют едва ли 20 лет и продолжают развиваться. Языки высокого уровня для массово-параллельной архитектуры все еще находятся в процессе развития. Новые сложные домены так же развиваются, что требует взаимодействия в различными стилями программирования.

При создании дизайна и разработке языков программирования руководствуются множеством аспектов, таких как развитие технологий, развитие компьютерной архитектуры, операционных систем, необходимость разработки крупномасштабного модульного программного обеспечения, а также необходимость поддержания программного обеспечения на длительный срок. С развитием новых предметных областей, будут появляться все новые требования, и таким образом, увеличивается потребность в новых языках программирования.

1.3.РЕЗУЛЬТАТЫ ОБУЧЕНИЯ

Результатом изучения данного курса является следующее:

1. *Сокращение кривой роста производительности для новых языков*: Требования создаются путем улучшения социальной инфраструктуры, предоставляемой автоматизацией. Язык программирования будущего будет более высокого уровня, будет включать в себя множество парадигм программирования и будет использоваться для разработки сложного программного обеспечения. Программисты должны будут изучать новые парадигмы языков программирования. Невозможно

переоснастить, пока кто-либо имеет глубокий абстрактный уровень понимания парадигм программирования, абстракций, программирование конструкций и их подводных камней. Более глубокий абстрактный уровень понимания различных парадигм программирования позволит программистам накладывать синтаксис новых языков на абстракцию для программирования на новых языках.

2. *Программистам станет известно об исполнении поведении низкого уровня*: Курс описывает влияние конструкций высокого уровня, путем их перевода в инструкции низкого уровня, которые выполняются на низком уровне абстрактных машин. Понимание низкоуровневого поведения поможет избежать многих подводных камней ошибок программирования. Это так же позволит улучшить студенческий стиль программирования вводя их в курс эффективного программирования и получения побочного эффекта. Побочными эффектами являются нежелательные вычислительные эффекты, вызванные абстрактными моделями вычислений и могут привести к сбою в работе программы.

1. *Программист сможет принимать участие в развитии компилятора*: С развитием новых предметно-ориентированных языков существует необходимость в разработке новых компиляторов. Понимание поведения языков программирования на низком уровне является основой генерации кода для эффективного выполнения.

2. *Улучшение стиля программирования*: Студенты узнают множество конструкций в различных классах языков программирования, которые расширяют сферу применения стиля программирования. Они также смогут эффективно выражать свои решения путем выбора соответствующих данных и абстракции управления. Как правило, узнав ограниченное количество языков программирования, таких как C++, Ява, PHP или C#, программисты получают влияние определенных стилей программирования. Знание других программных конструкций, используемых в различных парадигмах программирования, улучшает непосредственно само программирование.

3. *Программист будет иметь возможность выбирать соответствующие языки программирования и парадигмы*: Студенты будут иметь возможность составить предметные области с конкретными парадигмами программирования и выбрать соответствующие языки для разработки программ.

1.4 Программы и компоненты

Когда появляется вопрос, автоматизировать процесс или решить проблему, системный аналитик должен создать модель системы, параметризировать поведение ввода и вывода, а также подключить различные модули, используя блок-схему. Эти модули резюмируют реально происходящий процесс. Программы представляют собой технические условия для решений, разработанных для обработки данных и управления потоками данных между этими модулями.

Технические условия для решения реальных проблем описываются на высоком уровне для облегчения понимания для человека, а действия компьютера основываются на машинных командах низкого уровня. Существует необходимость в процессе перевода, который однозначно отображает (без какой-либо двусмысленности) высокоуровневые инструкции последовательно эквивалентным инструкциям низкого уровня. Инструкции должны содержать четкое уникальное значение, чтобы избежать двусмысленности.

Программа представляет собой последовательность значимых символов с целью официально указать решение к комплексной задаче (см. рисунок 1.1). Программа имеет три основных компонента*: логика + абстракция + контроль*. *Логика* подразумевает появление технических условий высокого уровня для решения проблемы. Это требует повторного решения сложной проблемы в структурированной комбинации более простых задач, а также сочетая решения этих простых задач, используя строго определенные операции для вычисления окончательного решения.

Абстракция подразумевает моделирование объекта требуемыми атрибутами, необходимыми для решения проблемы. Объект может иметь множество атрибутов. Однако все атрибуты не могут быть необходимы для решения проблемы. Преимуществом абстракции является то, что программы легко осмыслить и легко модифицировать в простоту обслуживания программы.

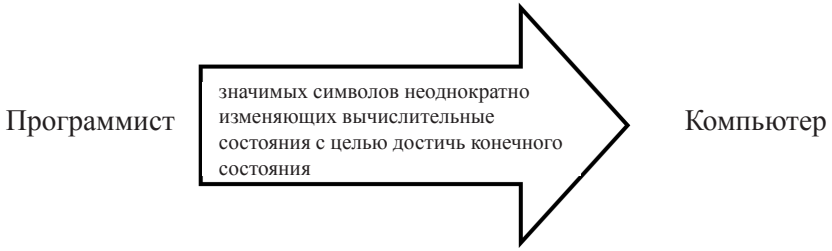


Рисунок 1.1 Абстрактное определение программ.

Контроль подразумевает отображение решения задачи, основанное на работе машины фон Неймана, где память компьютера постоянно изменяется, чтобы вывести конечное состояние вычисления, содержащее решение. Каждая инструкция изменяет вычислительное состояние в новое состояние. Использование явного управления в программе дает программисту возможность точно изменять память компьютера. Модификация компьютерной памяти для реализации логики изменяется от программиста к программисту и делает программу сложной для понимания.

Пример 1.1

Этот пример показывает три компонента через сортировку простыми обменами (сортировка пузырьком), которая сортирует последовательность неупорядоченных чисел. Сортировка простыми обменами использует стратегию «повторяющегося максимального элемента массива» для более мелких элементов подмассивов. Программа для сортировки простыми обменами включает в себя три основных следующих компонента:

- **Абстракция:** Представляет собой набор чисел, как индексируемую последовательность.
- **Логика:** Сравнивает смежные числа и меняет позицию числа, если следующее за ним число меньше до тех пор, пока не будет достигнут конец последовательности. Сравнение чисел в текущей последовательности дает максимум текущей последовательности. Этот элемент исключен в будущих сравнениях и процесс сравнения повторяется с остальными элементами до тех пор, пока не останется ни

одного элемента.

- **Контроль:** При контроле используется код для многократного обмена и обновления значений в различных ячейках памяти, связанных с переменными.

Программа организована с использованием нескольких блоков: (1) название программы, (2) импортированные из библиотеки программного обеспечения или модулей подпрограммы, разработанные в прошлом, (3) декларация информации о типе и переменных, необходимых для выражения логики, (4) параметры для обмена информацией между различными программными модулями и (5) последовательность команд для управления объявленными переменными.

Абстракции в программах

Существует два типа абстракции, используемых программой: *абстракции данных* и *контроль абстракции*. Абстракция данных используется для моделирования реальных объектов путем определения конкретных атрибутов, необходимых для решения существующей проблемы. Например, чтобы смоделировать класс для классификации цели, класс будет смоделирован как набор студентов, где каждый студент смоделирован как совокупность трех форм (*идентификатор студента, имя студента, буквенные оценки*). Многие другие характеристики студентов, такие как пол, рост, вес, адрес, спортивные интересы и прочее- не используются, потому что они не нужны для решения существующей проблемы. Абстракции данных могут быть осуществлены при помощи различных методов представления данных. Например, последовательность студентов может быть представлена в виде массива студентов, а совокупность трех форм- в виде «структуры» с тремя полями.

Пример 1.2

Например, одно такое представление класса выглядит как const class_ size = 20;

struct student **{string** student-id;
string student-name;
char letter-grade;
}
student class[class_size];

Абстракция управления классифицирует различные типы программных

конструкций в разные группы в зависимости от их общих свойств. Например, *утверждение* (оператор вида), *блок операторов, конструкции выбора* такие как *условные операторы (if-then-else конструкция)* или операторы *многозначного выбора* (case конструкция), *неопределенная итерация*- условное повторение последовательности конструкций, *определенная итерация*- точное количество предопределенных итераций, *вызов процедур* и *вызов функции*- это различные абстракции управления. Основным преимуществом абстракций является выражение конструкции высокоуровневой программы, которое легко изменяется и сохраняется.

Абстракция управления также связана с потоком управления управлением информационных диаграмм. Например, *если (<предикат>) затем <then-оператор> или же <else-оператор>* это абстракция управления которая выбирает *<then-оператор>* или *<else-оператор>* на основании значения истинности *<предиката>*. Если мы представим оценку предиката как *ромбовидную блок-схему* и операторов в виде *прямоугольной блок-схемы*, то условные операторы могут быть представлены, как на рисунке 1.2а. Аналогично,

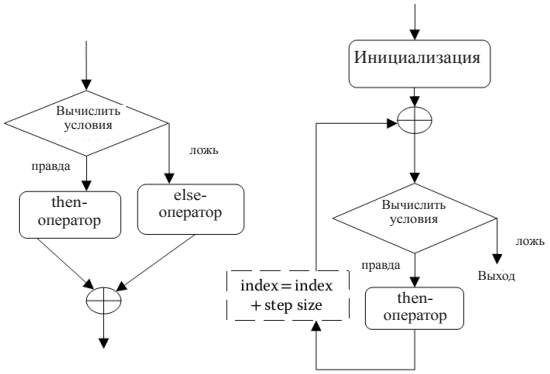


Рисунок 1.2 Порядок выполнения условного оператора (схемы if-then-else) и определенной итерации.

мы можем представить повторения блока операторов как *определенную итерацию*, *неопределенную итерацию* или *определяемую данными итерацию*- итерационные конструкции, где число итераций определяется числом элементов данных в коллекции.

Определенная итерация повторяет блок операторов фиксированное количество раз, используя комбинацию переменной индекса, нижнюю

границу, верхнюю границу и фиксированный размер шага. Индексная переменная принимает начальное значение либо нижнюю границу (или верхнюю), и постепенно переходит с размером каждого шага по направлению к другому концу до тех пор, пока не выйдет за пределы *нижняя граница* ≤ *переменная индекса* ≤ *верхняя граница*. Это называется *определенной итерацией*, так как количество раз выполняется блоком оператора фиксированно, опираясь на три значения: нижняя граница, верхняя граница и размер шага, и эти значения не могут быть изменены в пределах блока оператора внутри определенной итерации. Обобщенная блок-схема порядка выполнения для цикла «FOR» представлена в Рисунке 1.2b. Оценка состояния дается в ромбовидной блок-схеме. Блок операторов заключен в прямоугольник, а пунктирная рамка показывает внедренное приращение индекса.

Конструкция *неопределенной итерации* оценивает *логический тип данных (булевый тип)* для выполнения следующего цикла итерации, а компоненты, составляющие предикат, могут быть изменены в теле блока операторов внутри конструкции. Благодаря этому свойству, существует вероятность неопределенного зацикливания в неопределенном итерации.

Может существовать множество типов циклов, которые были изучены и которые содержат несколько условных операторов, следующих за блоком операторов. Однако, было доказано, что функциональная мощность всех этих различных типов циклов введена с помощью двух типов неопределенной итерации: цикла «WHILE» и цикла «DO WHILE». Циклы «DO WHILE» и « WHILE DO» имеют одну точку входа и выхода и не допускается переход извне цикла. Кроме того, они имеют один условный оператор, который определяет, стоит ли перейти к следующему циклу итерации или выйти из цикла. Обобщенная блок-схема порядка выполнения для циклов «WHILE» и «DO WHILE» показаны на Рисунке 1.3.

Цикл «WHILE» проверяет соответствие *логическому типу данных* перед составлением блок-схемы. Цикл *«DO WHILE»* в некоторых языках, таких как Pascal, также именуемый как цикл *«REPEAT-UNTIL»*, сначала выполняет

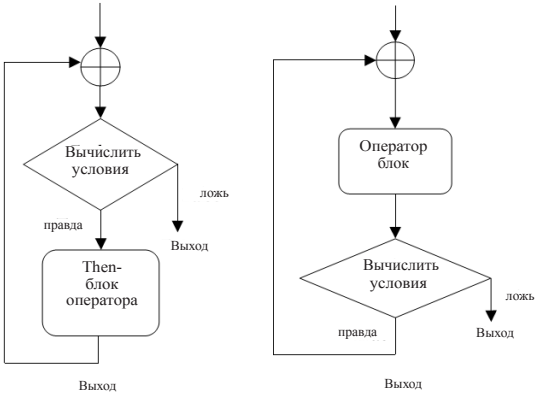


Рисунок 1.3 Обобщенная блок-схема порядка выполнения для неопределенной итерации.

Блок операторов, а затем проверяет состояние. Разница между этими двумя конструкциями в том, что цикл «DO WHILE» выполняет блок операторов по крайней мере, один раз, в то время как цикл «WHILE» может не использовать блок оператора даже единожды.

Управляемые базами данных итерации, также называемые итераторами как правило используются в абстракции данных, которые обобщают показатели элементов данных, таких как мультисеты и связные списки моделирования мультисетов. В отличие от классических схем итерации, использующих массивы, внутреннее представление структуры данных скрыто от программиста. Эти итераторы выполняют блок данных для каждого элемента в совокупности. Итераторы- это интересные абстракции, потому что число итераций зависит от размера данных, и они автоматически переступают через элементы в списке. Одна такая конструкция это цикл *«FOREACH»*в Lisp и Java, который выполняет итерацию через каждый элемент списка. Общая конструкция для итерации следующая:

foreach element **in** <*multiset*> **{**
<*block of statements*> ;**}**

Разберем чертеж блок-схемы и вложенных блок-схем, используя простые примеры. Пример 1.3 показывает блок-схему порядка

выполнения для вложенных условных операторов. Пример 1.4 показывает блок-схему порядка выполнения для определенной итерации, вложенной в неопределенную итерацию. Метод чертежа блок-схем заключается в том, чтобы перевести конструкцию высшего уровня и затем перевести постепенно перевести следующие уровни конструкций.

Пример 1.3

В этом примере мы сначала разработаем блок-схему для внешнего условного оператора; при разработке блока-схемы для внешней блок-схемы, полный условный оператор будет рассматриваться как один из блоков оператора. Вторым шагом мы расширим вложенный внутренний блок условного оператора. Результат показан на Рисунке 1.4.

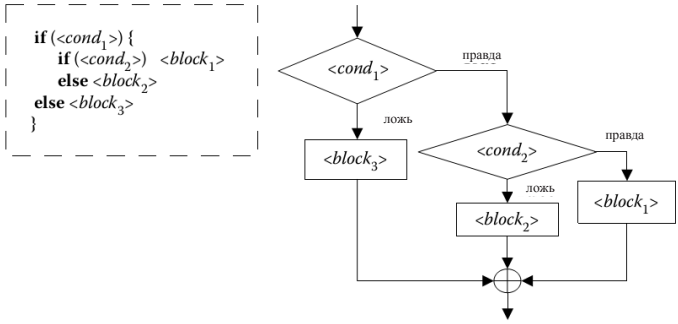


Рисунок 1.5 блок-схема для вложенных операторов.

Пример 1.4

Этот пример показывает развитие блок-схемы для вложенного цикла «WHILE». Сначала блок-схема для внешнего цикла «WHILE» создается путем обработки внутреннего условного оператора как оператора в рамках внешнего цикла «WHILE». Далее, создается блок-схема для внедренного условного оператора. Готовая блок-схема изображена на рисунке 1.5.

Ручной анализ (понимание программы) и Переходы

Одним из важных вопросов жизненного цикла программного обеспечения является *сопровождение программного обеспечения*, так как

(1) (1) существует необходимость развиваться, (2) программисты двигаются дальше, (3) программисты забывают свой подход за определенный период времени, и (4) архитектура и технологии значительно меняются. Обслуживание программного обеспечения непосредственно относится к ручному анализу: если элемент управления является сложным, тогда он труден для понимания, и любые попытки изменить программное обеспечение приведут к ошибкам в нем. В отсутствие должных языковых функций, поддерживающих понимание и стандарты программирования, старое ПО или станет бесполезным, или организациям нужно будет тратить значительное количество времени и ресурсов для поддержания ПО.

Программисты понимают части кода при помощи (1) понимания значения известных абстрактных конструкций, (2) понимания порядка выполнения в абстрактных областях, (3) моделирования информационных потоков между различными программными подразделениями, и (4) прогнозирования окончательного условия путем постепенного прогнозирования условий после простраивания индивидуальных конструкций в уме, без фактического выполнения программы.

Существует множество факторов, способствующих пониманию программ человеком. Иметь базовые знания и обучение в программировании необходимым, но не достаточным условием для понимания кода, написанного другими. Вот некоторые факторы, влияющие на понимание разработанного кода: (1) уровень абстракции на языке программирования, (2) простота абстракций, необходимых для перевода логики в код, (3) удаление чрезмерного контроля из кода, и (4) написание достаточно стандартизированных комментариев на уровне переменных, уровне блока, уровне модуля и уровне алгоритма, которые могут быть поняты другими. Кроме того, многие организации также налагают определенный стиль программирования и используют предпочтительный язык.

Люди очень хорошо понимают программы, которые структурированы, доступно классифицированы, модули функциональны и следуют прямо порядку выполнения. Так как число строк кода в модуле увеличивается, то для людей становится трудно понять и классифицировать общую функциональность модуля. Кроме того, слишком большое количество

переходов увеличивает неструктурированность характера программы и люди не могут мысленно визуализировать поведение программы во время ее выполнения.

Было показано, что функциональная мощность безусловных переходов (оператор *перехода*) максимальна. Безусловные переходы используются для реализации многих абстракций управления, таких как (1) *выбор*: условных оператор или оператор выбора и (2) *итерация*: цикл «WHILE DO», цикл «DO WHILE» и цикл «FOR», а также вызов процедур и вызов функции. В модуле используются переходы: (1) для передачи управления локально несколько инструкций удаленно и (2) для выхода из блока операторов. В 1970-х, среди компьютерных ученых проходила здоровая дискуссия о ручном анализе и использования переходов. Использование безусловных переходов и ручного анализа обратно пропорциональны друг другу. Люди могут обрабатывать переходы на несколько инструкций удаленно специально в прямом направлении. Тем не менее, переходы назад, переходы удаленно на длинные расстояния и слишком много переходов в целом, значительно сокращают анализ программы. Структурирование программы в блоки, подпрограммы, объекты и абстракции управления с четко определенными функциями, такими как цикл «WHILE DO» и «DO WHILE» улучшает анализ программы.

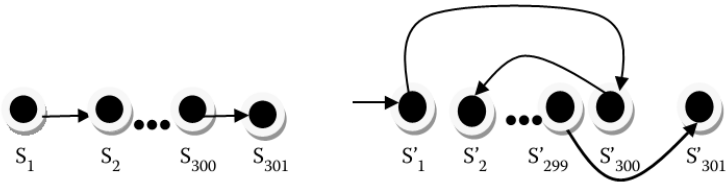
Рисунок 1.6a и b показывают схему порядка выполнения двух одинаковых программ. Программа на рисунке 1.6a перемещает одну инструкцию за один раз в прямом направлении, в то время как программа на рисунке 1.6b использует множество переходов в комбинации с прямыми и обратными переходами, а операторы перемешиваются с вложенными операторами перехода (операторами «GOTO»).

Представим, что S_1 эквивалентна S'_1, с последующим безусловным переходом к S'_{300}, S_2 эквивалентна S'_{300} с последующим безусловным переходом в обратном направлении к S'_2, S_3 эквивалентна S'_2, S_{300} эквивалентна S'_{299} с последующим безусловным переходом к S'_{301}, S_{301} эквивалентна S'_{301} и все операторы с S_4 по S_{299} эквивалентны соответствующим операторам с S'_3 по S'_{298}. Если мы посмотрим на *функциональную эквивалентность*- возможность делать одну и ту же задачу- обе программы функционально эквивалентны. Тем не менее, легко понять

структурированную программу на рисунке 1.6a.

Из-за недостатка понимания, вызванного свободным использованием безусловных операторов перехода, были разработаны различные

абстракции управления. Использование операторов перехода ограничивается осуществлением контроля абстракций и для выхода из вложенных блоков. Для



(a)Структурированная программа (b) Программа с переходами

Рисунок 1.6 Блок-схема структурированной программы по сравнению со спагетти-кодом с переходами.

можно доказать, что программы, использующие операторов перехода, могут быть переведены на функционально эквивалентную структурированную программу с использованием комбинации булевых переменных и абстракций управления, таких как, условный оператор (IF-THEN-ELSE), цикл «WHILE» или цикл «DO WHILE».

Выполнение программ

Основные критерии для звукового выполнения программы:

1. Языковые конструкции языка программирования должны быть четко определены.

2. Должно содержаться уникальное значение для каждой языковой конструкции, так как компьютеры не обрабатывают двусмысленных запросов.

3. Каждая высокоуровневая конструкция должна быть переведена к последовательности конструкций низкого уровня, которые последовательно выполняют на компьютере одни и те же действия каждый раз.

4. Компьютер должен последовательно выполнять порядок действий низкоуровневых конструкций, производящих один и тот же конечный результат.

Существует несколько слоев программного обеспечения до слоя языка программирования, как показано на рисунке 1.7. На самом низком уровне это «пустая» (без ПО) машина и машинный код. Следующий уровень- это операционная система, системные утилиты, и языковые переводчики. Внешний слой- это высокоуровневое программирование и протокол прикладного уровня. Программа высокого уровня переводится в инструкции низкого уровня и использует несколько интерфейсов промежуточного слоя, чтобы использовать компьютерные системные ресурсы и утилиты, необходимые для выполнения программы.

Существует три способа, которыми инструкции высокого уровня могут быть переведены в набор инструкций низкого уровня: (1) компилирование инструкций высокого уровня перед использованием программы; (2) интерпретирование программы высокого уровня, и выполнение ее, используя встроенную абстрактную машину; и (3) своевременная компиляция высокоуровневого языка, которая дает эффект частично скомпилированного кода и частично интерпретируемого кода.

Как показано на Рисунке 1.8, процесс компиляции переводит инструкции высокого уровня в инструкции низкого уровня перед выполнением программы. С другой стороны, переводчики переводят и выполняют одну инструкцию за раз. В отличие от скомпилированного кода, интерпретированный код проходит через цикл перевода и исполнения для самого оператора.

Рисунок 1.7 Различные слои программного обеспечения в компьютере.

Рисунок 1.8 Компиляция в сравнении с интерпретацией. (a) схематическая компиляция (b) схематическая интерпретация.

Преимуществом компилированного кода является то, что он не имеет накладных расходов перевода во время выполнения программы. Другое преимущество использования компиляторов- это то, что обнаруживается высокий процент программных ошибок и распределение памяти оптимизируется перед выполнением.

Эффективность выполнения интерпретированного кода на порядок медленнее, чем эффективность выполнения скомпилированного кода, так как процесс перевода перемежается с выполнением во время выполнения. Другим недостатком интерпретации является то, что все ошибки не могут быть обнаружены до начала выполнения программы, и интерпретированные программы могут дать сбой после выполнения многих инструкций, что делает его небезопасным для критически важных программ. Тем не менее, переводчики легко разработать и они использовались ранее для тех языков, для которых компиляторные технологии не были хорошо развиты.

В декларативном программировании языки, которые имеют возможность разрабатывать программу в качестве данных, а затем превращая их в программу во время выполнения, переводчики были выбором. Переводчики способствуют более интерактивным возможностям отладки на уровне инструкций, отображая все значения переменных. Переводчики поддерживают выполнение динамически типированных языков, где переменная может быть связана с любым типом объекта данных во время выполнения. Переводчики также лучше поддерживают машинно-независимые программы, поскольку

программы выполняются в рамках переводчика, и программа не должна быть перенесена на разные машины.

В языках, основанных на интернет-технологии, код передается в виде потока на клиентские компьютеры, которые имеют неприятие абстрактной машины, установленной для данной операционной системы. *Абстрактная машина* представляет собой программу, которая работает на данной машине и выполняет стандартный набор инструкций, который не зависит от операционной системы и компьютерной архитектуры. Виртуальная машина языков Java (JVM)- это одна из таких абстрактных машин, которая выполняет программы Java на нескольких платформах. Выполнение инструкций на JVM происходит гораздо медленнее, чем выполнение машинных команд на родной машине, потому что JVM использует *нулевую адресацию* для универсальной совместимости, в то время как сегодняшние компьютеры используют двух- или трехадресные наборы инструкций. Выполнение кода на машинах с более старшим адресом происходит быстрее, чем на машинах с нулевой адресацией, как описано в Разделе 2.1.

Чтобы ускорить выполнение на более высоких адресных машин, *JIT* компиляторы сохраняют библиотеку выполняемых методов, используя набор команд родной операционной системы, которая намного быстрее, чем выполнение эквивалентных инструкций на машине Java. После проверки безопасности, фрагменты промежуточного кода переводятся в машинный код при первом наступлении промежуточного кода и кэшируются в библиотеке для дальнейшего поиска. При последующем наступлении фрагментов, выполняемый машинный код ищется и выполняется. Аналогично, если высокоуровневый метод Java уже имеет скомпилированный код в своей библиотеке, то он извлекается и выполняется. Если двоичный код недоступен или фрагмент не может быть скомпилирован в реалистичные сроки, то код высокого уровня переводится согласно инструкциям в машину Java и выполняются. Общая схема для JIT компиляции показана на Рисунке 1.9.

Существуют накладные расходы перевода, когда кодовые фрагменты компилируются в первый раз. Тем не менее, на последующих появлениях фрагментов, не существует никаких накладных расходов перевода. Эффективность выполнения кода при помощи JIT компиляции лежит между эффективностью исполнения интерпретируемого кода и скомпилированного кода.

Некоторые производители, такие как Microsoft имеют свой собственный промежуточный язык, который называется *Общий*

Промежуточный Язык (CIL). Преимущества общего промежуточного языка заключается в обеспечении совместимости между различными языками высокого уровня. Язык высокого уровня сначала переводится на общий промежуточный язык, а затем он передается через Интернет и компилируется на различных архитектурах с использованием JIT компиляции, как показано на рисунке 1.10. Языки, такие как C#, visual C++, F#, и Visual Basic переводятся в общий промежуточный язык.

Рисунок 1.9 Схематичная JIT компиляция.

Современные компиляторы используют двухступенчатый вариант перевода для выполнения программ высокого уровня на различных архитектурах. Различные архитектуры имеют разные языки сборки и не представляется возможным написать единый компилятор для нескольких архитектур. Вместо этого был разработан промежуточный код для языков программирования. На первом этапе компилятор переводит программы высокого уровня в код промежуточного уровня, который не зависит от компьютерных архитектур. На втором этапе переводится промежуточный код в машинный код низкого уровня.

Первый этап перевода состоит в (1) *лексическом анализе*, (2) *синтаксическом анализе*, и (3) *семантическом анализе* и *генерации кода* для создания промежуточного кода, как показано на Рисунке 1.11.

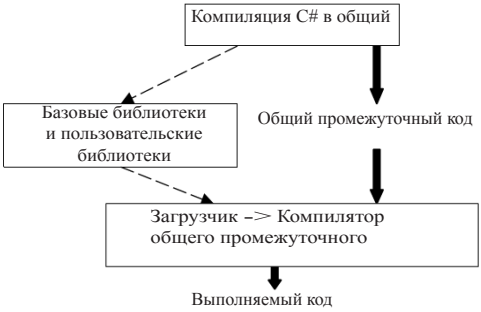


Рисунок 1.10 JIT компиляция на C#.

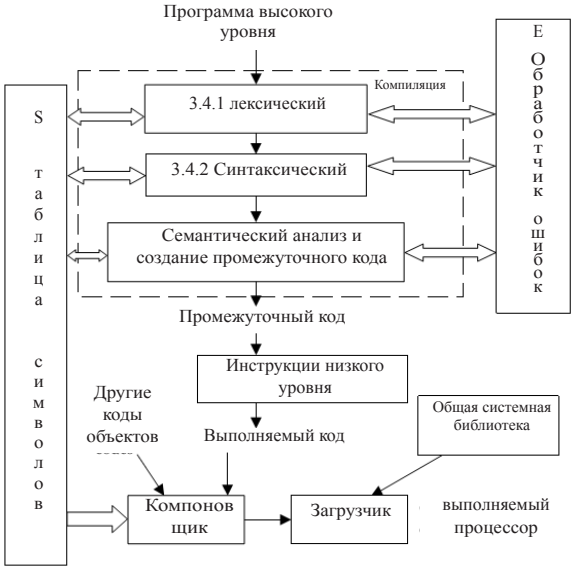


Рисунок 1.11 Перевод программ высокого уровня для выполнения на компьютере.

Лексический анализатор проверяет зарезервированные слова-значимые слова, которые являются частью языка, идентификаторов, переменных и чисел- и преобразует их во внутреннее представление, называемое *маркерами* для простоты синтаксического анализа. Выход лексического анализа это маркированный поток, который становится входом для синтаксического анализатора. Синтаксический анализатор

проверяет структуру программного предложения в соответствии с языком программирования. Его выход представляется в виде дерева (структура данных) для облегчения эффективной внутренней обработки в процессе генерации кода. Семантический анализатор проверяет, что сгенерированные предложения имеют смысл и генератор кода линеаризирует представление структуры данных к соответствующей *промежуточной кодовой* последовательности абстрактных инструкций низкого уровня. Уровень оптимизации удаляет избыточные фрагменты кода и улучшает использование процессора регистров для повышения эффективности выполнения исполняемого кода. *Таблица символов* используется для хранения информации из предыдущих этапов, которые могут быть необходимы или, возможно, потребуется разрешение в следующих этапах. Тип информации, который содержит таблица символов содержит имена переменных и их расположение, уровень вложенности процедур, информацию о нелокальных переменных, где был названа процедура, какая процедура была названа, и так далее.

В дополнение к двухступенчатой компиляции, *компоновщик* связывает множество объектных файлов (скомпилированный код) в один большой исполняемый код. Связывание осуществляется в порядке, определенном программистом в команде компоновки и не зависит от вызывающего шаблона процедур. Скомпилированный код подпрограммы происходит только один раз в исполняемом коде и не зависит от частоты вызова подпрограммы. После связывания, информация о взаимном расположении процедур фиксируется. *Загрузчик* загружает связанный код, который будет выполнен в сегменте- области памяти, выделяемую пользовательскому процессу, соответствующей исполняемому коду. Загрузчик перемещает логический адрес, полученный после ссылки на физический адрес в ОЗУ (оперативное запоминающее устройство). Загруженный исполняемый файл программы называется *процессом* и выполняется с использованием комбинации программного обеспечения операционной системы и аппаратных методик, которые вы будете изучать на курсе об операционных системах. Библиотеки динамической компоновки- это общие системные библиотеки для многократного использования различными программными приложениями. Вы узнаете о механизмах библиотеки динамической компоновки на курсе об операционных системах.

1.5 СОВМЕСТИМОСТЬ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Совместимость языков программирования это способность

фрагмента кода на одном языке, взаимодействовать с фрагментами кода на других языках программирования. Современные языки программирования обеспечивают совместимость языка для поддержки (1) смешивания предметных областей в сложной задаче; (2) использования абстракций, специфичных для различных языков; (3) эффективности путем взаимодействия с библиотеками низкоуровневых языков; и (4), максимизации повторного использования кода уже разработанной библиотеки программного обеспечения, улучшения повторного использования и обслуживания программного обеспечения.

Для обеспечения совместимости, необходимо иметь возможность включать скомпилированный модуль, разработанный на других языках, для обмена информацией с составленным модулями, написанными на других языках, и иметь общие *метаданные* - отдельно заявленные сведения о модуле и экспортируемых типах данных, которые могут быть использованы кодами на других языках. Кроме того, обмен информацией должен быть преобразован обратно в собственные абстракции данных на языке хоста.

Наблюдается тенденция обеспечивать общую спецификацию языка и стандартные спецификации общих типов данных для равномерного обмена информацией между языком хоста и сопряженными языками. Общие языковые спецификации включают (1) определение и использование согласованных типов между языками и (2), определяющий единый механизм для хранения и извлечения информации о типе корреспонденции на разных языках. Некоторые компании, а именно Microsoft, разработали общую спецификацию языка, использующую проект .NET, чтобы обеспечить стандартные общие черты и правила, необходимые для обеспечения взаимодействия.

1.6 ЦИКЛ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Теперь мы обратим наше внимание на взаимосвязи языков программирования к разработке программного обеспечения. В конце концов, языки программирования используются для разработки большого программного обеспечения для автоматизации некоторых реальных процесс. Программное обеспечение может состоять из сотен тысяч строк кодов. Успешное развитие объемного программного обеспечения требует команд программистов, системных аналитиков и отладчиков.

Существует несколько этапов в жизненном цикле программного обеспечения: *анализ требований*, *системный анализ и проектирование*, *разработка*, *внедрение*, *начальная проверка*, *полевые испытания*, *ввод в действие* и *эволюция*. Прежде чем перейти к новому этапу, после каждого выполненного этапа используются обратные связи из следующих процессов для уточнения предыдущих. В конце ввода в действие, возможности системы хорошо понятны клиентам и их потребности развиваются. Существует необходимость в уточнении в области системного анализа и проектирования, чтобы удовлетворить

новые потребности. Эволюция инкрементна, она учится на предыдущих анализах и совершённых проектных ошибках, а также получает выгоду от технологических усовершенствований в течение следующего цикла разработки программного обеспечения.

На этапе *анализа требований*, системный аналитик выслушивает запросы клиента для автоматизации, неполной и зачастую плохо определенной информации, а также тип запросов, который клиент намерен просить. Объем автоматизации является фиксированным, и цели проекта также фиксированы. В фазе *системного анализа*, группа системных аналитиков изучает объем и систему, которую предстоит автоматизировать, а затем разрабатывают модель информационных потоков для процесса. На *стадии разработки* с помощью ведущих программистов и системных проектировщиков, модель потока информации разделяется на различные взаимосвязанные модули. Эти модули имеют независимые функциональные возможности, так что эволюция одного модуля существенно не влияет на другие модули. Устройства ввода-вывода информации из различных модулей четко фиксируются. Проектировщики проектируют модули с потенциалом для развития, так как потребности клиентов развиваются. Эта часть визуализации и прогнозирования будущей эволюции потребностей клиентов является довольно сложным процессом и требует опытных аналитиков и разработчиков систем.

На *стадии развития*, после того, как модули и их связи (ввод-вывод к модулям), были разработаны, команда ведущих программистов разрабатывает абстракции данных, интерфейсы и алгоритмы для каждого модуля, учитывая технические трудности и ограничения конструкций в доступных языков программирования в предметной области. Если автоматизация включает в себя более одной предметной области, то модули могут быть разработаны с использованием нескольких языков программирования, использующих функциональную совместимость.

На *стадии реализации*, после разработки основных высокоуровневых алгоритмов, программный код считается разработанным. Эта часть выполнена командой программистов и преобразуется в интерфейс потока данных, разработанный на предыдущих этапах и транслируется в программный модуль интерфейса между различными модулями программирования. Принимаются меры, чтобы обеспечить достаточное сокрытие информации для предотвращения использования программистами локальных особенностей других модулей, чтобы минимизировать число непреднамеренных ошибок. Разработка кода это только треть всего процесса реализации программного обеспечения. Основные усилия прилагаются на стадии проверки, стадии тестирования и стадии ввода в действие. Программисты должны прислушаться к будущим потребностям и жизненным циклам программного обеспечения и обеспечить простоту развития сферы программного обеспечения.

В противном случае, даже небольшие программные или проектные ошибки могут сделать программное обеспечение непригодным для использования и калечат ежедневные основные функции. Примером таких ошибок была проблема Y2K, исправление которой стоило миллиардов долларов. Программное обеспечение должно также делать послабление для будущих усовершенствований в технологиях низкого уровня, таких как более быстрые компьютерные архитектуры и улучшенное аппаратное обеспечение.

На *начальном этапе проверки*, программисты проверяют программу на выборочных данных, предоставленных клиентом. После того, как этап удачно завершен, разработанное программное обеспечение получает статус альфа-версии и отправляется отладчикам на полевые испытания для дальнейшего тестирования клиентом. Отладчики вносят изменения и получают обратную связь от клиентов, чтобы скорректировать модель и внести изменения в программу. В ходе испытаний и проверок, новая, исправленная версия, получившая название *бета-версия* тестируется отладчиками во всех направлениях с целью поиска любой ошибки, используя фактические данные клиента до тех пор, пока не будет запущен окончательный безошибочный вариант.

На этапе ввода в работу, уточненный вариант выпускается для публичного использования (или для использования клиентом), а также начинаются полевые испытания и сбор обратной связи. Клиент видит потенциал программного обеспечения и предлагает улучшение после того, как программное обеспечение интегрировано с общей системой на рабочем месте. Предложенные изменения быстро включаются в программное обеспечение на сайтах на местах (на рабочем месте) или с помощью *патчей программного обеспечения*- небольших частей программного обеспечения, чтобы исправить незначительные ошибки и проблемы безопасности.

В течение определенного периода времени определяются сильные и слабые стороны программного обеспечения, затем клиент определяет новые потребности, которые будут включены в программное обеспечение, а также начинается новый итерационный цикл эволюции программного обеспечения. В течение этого периода, аппаратные технологии могут также меняться, а программное обеспечение должно поддерживаться, чтобы сделать его переносимым для различных архитектур. Схема цикла разработки программного обеспечения приведена на рисунке 1.12.

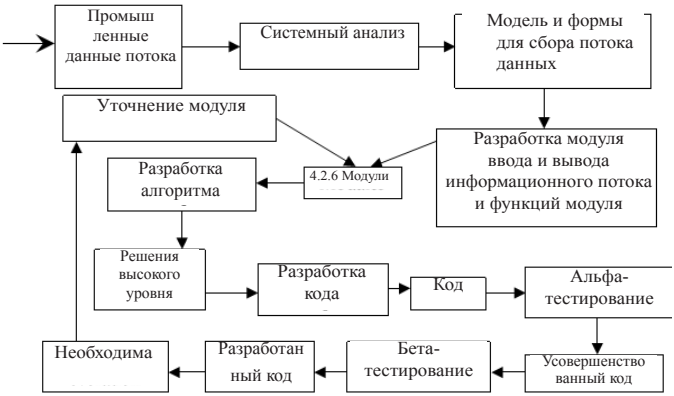


Рисунок 1.12 Цикл разработки программного обеспечения.

Существует много моделей программного цикла, таких как «водопадная модель (разрабатываемого проекта)» и «спиральная модель» для разработки программного обеспечения. Эти модели неявно включены в приведенном выше обсуждении. Тем не менее, сравнение и детальное изучение этих моделей лежит в рамках курса по программной инженерии.

Программное обеспечение продолжает развиваться с течением времени по мере необходимости развиваться до тех пор, пока новая технология не сделает старые технологии устаревшими. В этом случае новое программное обеспечение должно быть разработано для лучшей автоматизации. Существовало четыре основных изменения в истории промышленной автоматизации:

(1) развитие высокоуровневых, объектно-ориентированных языков программирования; (2) разработка визуальных и мультимедийных технологий для лучшей визуализации; (3) развитие высокоуровневых баз данных языков; и (4) развитие веб-языков, таких как XML для облегчения передачи и совместного использования ресурсов через Интернет.

Очевидно, что разработка программного обеспечения требует командной работы, обслуживания программного обеспечения, портативности и непрерывной эволюции. Любая задача, которая требует большой команды в течение длительного периода времени требует стандартизации программного обеспечения, модульности, удобства для пользователя, и легкости понимания.

1.7 КРИТЕРИИ ДЛЯ ХОРОШЕГО ЯЗЫКА ПРОГРАММИРОВАНИЯ

Критерии для хорошего языка программирования руководствуются потребностями предметной области ; разработкой программного обеспечения, обслуживанием и эволюцией; и взаимодействием и эффективностью исполнения. Некоторые из критериев это: (1) *абстракция*, (2) *модульность*, (3) *ортогональность*, (4) *обработка исключений*, (5) *удобство для пользователя*, (6) *дискретность*, (7) *легкость понимания* и *технического обслуживания*, (8) *общая простота* и (9) *портативность*. Мы уже обсуждали абстракции, модульность, мобильность и легкость понимания. *Ортогональность* означает, что конструкции в языках программирования должны быть независимы друг от друга и не должны быть избыточными. Возможность *обработки исключений* — это необходимость фиксировать условия возникновения ошибок во время выполнения путем вызова пользовательской процедуры для тонкой коррекции условий ошибки или выхода, чтобы избежать сбоев программы. В крупных программах, то есть, критичных программах, таких как программа мониторинга атомной электростанции, а также критически важных , таких как программы, управляющие самолетом или космический кораблем , - это является важным критерием. *Дискретность* означает, что язык должен поддерживать конструкции и возможности, которые делают программы читаемыми для простоты понимания. Например, должны быть разрешены большие имена переменных, чтобы программисты могли использовать понятные имена переменных. Удобство понимания и общей простоты напрямую связаны с разработкой программного обеспечения, эволюцией и техническим обслуживанием, а также зависят от количества абстракции, присутствующей в языке.

Руководители отрасли задали интересный вопрос: какой язык является хорошим для автоматизации моей отрасли ? Этот вопрос исходит от базовой необходимости стандартизации, таким образом, компаниям приходится тратить меньше ресурсов и финансовых средств для поддержания программного обеспечения в течение определенного периода времени. К сожалению, нет никакого чудодейственного средства; не существует ни одного языка, который может обрабатывать все предметные области. Каждая область имеет различные требования.

Например, язык системного программирования должен быть близким к языкам сборки и должен иметь инструкции для эффективного доступа к модулям памяти и устройств ввода/вывода; включение слишком большого количества абстракций более высокого уровня не будет

способствовать эффективности.

Язык научных вычислений должен представлять большие матрицы без их дублирования, потому что дублирование будет тратить драгоценное пространство памяти и вызывать вычислительные накладные расходы на копирование больших матриц из одного пространства памяти в другое . Язык в режиме реального времени должен быть эффективным, чтобы избежать потери событий в реальном времени. Дружественный язык позволит избежать объявлений любого типа, но тогда он страдает от множества проблем, таких как отсутствие оптимизации памяти, отсутствие эффективного выполнения, и иногда программы могут давать сбой во время выполнения.

Многие из этих требований, предъявляемых к хорошему языку , противоречивы. Если мы хотим развивать хороший язык, который имеет много абстракций высокого уровня, процесс перевода будет генерировать много избыточных инструкций низкого уровня из-за обобщенных механизмов, используемых в переводчиках. Точно так же, если мы делаем язык ближе к языкам сборки, тогда портативность и, следовательно, обслуживание программного обеспечения страдает, так как языки сборки различаются для разных архитектур. Мы перешли от 16-разрядных инструкций к 32-разрядным инструкциям, а в последнее время и к 64-разрядным. Если программа тесно связана с инструкциями на уровне сборки, то изменять архитектуру порта каждый раз будет сложно. Технологии продолжают меняться все более быстрыми темпами, что требует адаптации программных решений с минимальной задержкой и минимальным количеством инвестиций. Таким образом, новые языки требуют ухода от низких языковых конструкций ради портативности, модифицируемости и сопровождения программного обеспечения. В противном случае, мы будем тратить миллионы и миллиарды долларов только лишь для того, чтобы переносить программное обеспечение на новые архитектуры.

1.8 ИСТОРИЯ ПАРАДИГМ И ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Парадигма программирования - это стиль программирования. По мере развития программирования, пользователи и компьютерные ученые обнаружили недостатки и преимущества в предыдущих стилях программирования и стали развивать новые стили программирования. Современные языки представляют собой сочетание двух или более парадигм программирования, как показано в Приложении I.

Мы можем классифицировать программы на комбинации одной или нескольких парадигм программирования:

(1) (1) императивное программирование, (2) декларативное программирование, (3) объектно-ориентированное программирование, (4) параллельное и распределительное программирование, (5) визуальное программирование, (6) веб-программирование, (7) событийно-ориентированное программирование, (8) мультимедийное программирование, (9) мультиагентное программирование и (10) синхронное программирование. На этом список не заканчивается. Однако объем этого учебника ограничивается изучением класса языков программирования, поддерживающих основные парадигмы программирования.

1.8.1 Парадигма императивного программирования

Основой императивного программирования является *утверждение* или присваивание, которое реализуется на фон-неймановской архитектуре. Программист вручную переводит логику в явном виде, чтобы сказать компьютеру, что нужно делать. Переменная в императивных программах отображается в ячейку памяти в компьютере, а ячейка памяти может быть изменена несколько раз с помощью операторов присваивания. Действием оператора присваивания является то, что новое значение записывается в ячейку памяти; старое значение теряется. Преимуществом оператора присваивания является *повторное использование памяти*, так как та же ячейка памяти используется для хранения нескольких значений. Тем не менее, есть много недостатков в записи в ячейку памяти несколько раз:

1. Если ячейка памяти переписывается, старое значение теряется, что делает невозможным использовать старое значение в будущем. При программировании искусственным интеллектом, решение ищется в огромном пространстве поиска, возврат - поворот назад и отмена части поиска, и апробирование альтернативных путей поиска, - становится невозможным.

2. Если объем памяти, принадлежащий к другой процедуре или функции, записывается вызываемой в настоящее время процедурой, то старое значение теряется. Если вызывающая процедура или процедуры не знают ничего о модификации или нуждаются в использовании старого значения, то обновление повредит ячейки памяти. Результатом будет генерирование неверных решений. Эта проблема называется *побочным*

эффектом, и она была подробно рассмотрена в разделе 4.8.

3. Чередование оператора присваивания с логикой программы вызывает проблемы в понимании программы, поскольку контроль очень субъективен для стиля мышления отдельного программиста и вызывает проблемы с обслуживанием программы в долгосрочной перспективе.

Парадигма императивного программирования была первой разработанной программой в конце 1950-х и 1960-х годов в разных версиях *FORTRAN* и в начале 1960-х годов в разработке блочно-структурированных языков, таких как *ALGOL*. Эти четыре ранних языка были довольно популярны среди парадигм императивного программирования: FORTRAN (Фортран), ALGOL (Алгол) и COBOL (Кобол), и C (Си), потомок языка ALGOL (Алгол).

Ранние версии Фортрана широко использовались для научных вычислений, сделав либеральным использование операторов перехода, и не поддерживали указатели. В Алголе появилось представление о программе, как о блочной структуре, поддерживающей управляющие конструкции, такие как *цикл WHILE DO* и *цикл DO WHILE*, что улучшило понимание программы. Язык Алгол также использовал указатели и «структуры», которые использовались для организации рекурсивных процедур, таких как «связный список» и «деревья». Алгол также известен как основоположник современных императивных языков программирования: большинство абстракций данных и абстракций управления, описанных Алголом, до сих пор используются в современных языках.

Язык Си изначально был языком программирования общего назначения, разработанный сотрудником компании Bell Labs для реализации операционной системы UNIX. Это было подмножество языка Алгол 68, и оно завоевало популярность благодаря популярности UNIX и его вариаций. Были и другие современные языки, такие как SDL (разработан корпорацией Burroughs, которые поддерживали развитие Алгол 68) и BLISS 32 (разработанный Digital Equipment Corporation), которые проиграли из-за популярности публичных версий Unix, используемых в промышленности и научных кругах . Другие примечательные языки в этой категории - это ADA (Ада), Pascal (Паскаль), и семейство языков Модула, в том числе Модула-3 и Оберон, которые поддерживаются блочно-структурированным программироанием высокого уровня. Семейство языков Модула поддерживает понятие модулей и концепцию импорта и экспорта функций. Все эти языки были «выходцами» из

языка Алгол 68. Тем не менее, все эти языки, за исключением языка Си и его вариаций, всегда оставались в академическом мире и никогда не получили коммерческой популярности.

Кобол - это язык для разработки бизнес-приложений. Он делает акцент на удобство для пользователя, составление отчетов и обработки финансовых данных. Была также попытка IBM (Ай-Би-Эм) разработать язык, PL-I, целью которого было добавление конструкций для структурного программирования на языках в Фортран, Алгол и Кобол. PL-I был спроектирован как язык, который может быть использован для нескольких доменов программирования. Тем не менее, из-за его громоздкой природы, он проиграл гонку.

Многие императивные языки программирования, такие как Ада, Фортран и Кобол, продолжали развиваться за счет включения проверенных конструкций. Более новые версии языков Фортран и Кобол имеют множество функций, таких как блочно-структурное программирование, рекурсивное программирование, внедрение на основе списков (стеков), обработка строковых данных, указатели, структуры и объектно-ориентированное программирование. С течением времени, разработчики языка определили преимущества и недостатки современных языков и включили полезные конструкции. Когда язык развивается, необходимо соблюдать осторожность, чтобы сохранить его совместимость со старыми версиями, так что эволюционировавшие программы могут использовать старые библиотеки, и программы, написанные с использованием старых версий языков, могут быть скомпилированы с использованием компиляторов для новых версий.

1.8.2 Парадигма декларативного программирования

В декларативном программировании, контроль был взят из программы. Существует абстрактная машина, которая неявно заботится о части управления. Декларативная программа состоит из *логики+ абстракции* на программируемом уровне. Понятие переменной в декларативных программах довольно сильно отличается от понятия переменных в императивных языках. Переменная в декларативной программе является держателем значения; как только значение было присвоено переменной, оно не может быть изменено программистом. *Преимуществом* являются следующие свойства: (1) существует меньшая вероятность побочных эффектов, например , когда вызываемая процедура (или функция) не может переписать в пространство памяти вызывающей процедуры (или

функции) и (2) старые значения переменных могут быть сохранены и использованы при необходимости. Также существует множество недостатков, среди которых:

1. Ячейка памяти не может быть повторно использована, даже если она не нужна в качестве использования индексной переменной для цикла FOR. Обработка итеративного вычисления и процесса ввода-вывода, таких как печать большого массива элементов данных вызывает взрыв памяти.

2. Рекурсивное программирование используется чаще, так как традиционные итерации с использованием *изменяемой переменной индекса* не допускаются из-за ограничений, наложенных на переменную мутации. Рекурсивное программирование имеет как память, так и выполнение накладных расходов и представляет собой более сложный процесс программирования в сравнении с итеративным программированием.

3. Глобальные изменяемые переменные, которые поддерживают хранение частичных вычислений и которые могут быть использованы позже другими частями программы для повышения эффективности исполнения, не допускается. Тем не менее, многие декларативные языки программирования попытались преодолеть это ограничение: Лисп (Lisp) допускает частичное переписывание в ячейку памяти, разрешая ограниченное использование глобальных переменных; а Пролог (Prolog), в зависимости от реализации, разрешает конструкции, такие как «assert» (конструкция, позволяющая проверять предположения о значениях произвольных данных в произвольном месте программы), «доска объявлений» (область памяти, общедоступная для всех модулей системы) или глобальных изменяемых переменных, чтобы сохранить результат частичных вычислений.

Есть два основных типа декларативных языков программирования: функциональные языки программирования и языки логического программирования. Функциональные языки программирования основаны на использовании математических функций, и языки логического программирования основаны на использовании логики предикатов - булевой логики в сочетании с понятием количественной оценки, как описано в главе 2.

Лисп, популярный язык, который в сочетании с функциональным

программированием с некоторыми императивными конструкциями был разработан в начале 1960-х годов для реализации искусственного интеллекта программирования - отрасли информатики, которая пытается имитировать человеческий интеллект с помощью вычислительных средств. Пролог, популярный язык логического программирования, был реализован в 1970-е годы для автоматизированного доказательства теорем и искусственного интеллекта программирования. Оба эти языка были развиты в начале 1980-х годов, когда были разработаны обширные методы компиляции. Их составители применили многие инновации, чтобы преодолеть трудности эффективного выполнения и повторного использования памяти в течение определенного периода времени. В последующие годы, другие функциональные языки, которые находились под влиянием Лисп, были разработаны и стали весьма популярными. Некоторые из популярных примеров: Scheme (Ским), ML (ЭмЭль), Miranda (Миранда) и Haskell (Хэскелл). В настоящее время многие мультипарадигмальные языки, такие как Ruby (Руби), и Scala (Скала) поддерживают функциональную парадигму программирования. Полный список представлен в Приложении I.

Декларативные языки программирования традиционно используются в области искусственного интеллекта. Ключевые особенности декларативных языков, необходимых в области искусственного интеллекта заключаются в следующем:

1. Системы искусственного интеллекта должны динамично развиваться и обновлять знания. AI языки поддерживают абстракции для включения знаний.

2. Большинство декларативных языков рассматривают программу в качестве первого класса, что означает, что программы могут быть построены во время выполнения, так как данные затем преобразуются в программу, которая может быть выполнена. Искусственный интеллект нуждается в этом свойстве для компиляции новых знаний.

3. Декларативные языки обеспечивают возможность метапрограммирования- это программа, которая рассуждает о другой программе в абстрактной области. Это свойство метапрограммирования полезно при разработке рассуждения и объяснения возможностей в системах искусственного интеллекта.

1.8.3 Парадигма объектно-ориентированного программирования

По мере увеличения компьютерного объема памяти, выполнение

больших программ стало возможным. Первым объектно-ориентированным языком в 1960х стал язык Симула. Тем не менее, понятие объектно-ориентированного программирования прижилось в сообществе разработчиков программного обеспечения в начале 1980-х годов. По мере того как размер программы вырос, люди начали понимать ценность *модульности* и *повторного использования программного обеспечения* для разработки еще более сложного программного обеспечения.

Многие важные взаимосвязанные понятия, такие как *модульность, повторное использование программного обеспечения, не требующие доработок библиотеки* и *сокрытие информации* были задуманы, чтобы обеспечить простоту разработки программного обеспечения, технического обслуживания и его эволюции. *Модульность* означает деление больших компьютерных программ на набор взаимосвязанных модулей, таких, что каждый из них имеет четкую функциональность, которая не пересекается с функциональностью других модулей. Основным преимуществом модульного развития является то, что модификация в одном модуле не оказывает негативного влияния на функциональность других модулей. *Повторное использование программного обеспечения* означает, что ранее разработанное программное обеспечение может быть использован повторно путем простого хранения его в файле и импортирования необходимых модулей или подпрограмм из архива, когда это необходимо. *Сокрытие информации* означает сделать часть реализованного программного модуля недоступным для других модулей, если он не нужен для взаимодействия с другими модулями.

Для того, чтобы обеспечить модульность и сокрытие информации, были разработаны понятия объектов и классов. Объект имеет как абстракции данных и методов - связанных функций, так и процедуры, необходимые для манипулирования абстракциями данных в объекте. Информация внутри объектов может быть *частной*, *общедоступной* или *защищенной. Общедоступные методы* доступны для внешнего мира; *частные методы* являются специфическими для объекта, обеспечивающего сокрытие информации; и защищенные методы могут видеть объекты в подклассах текущего класса. Сокрытие информации также связано с понятием модульности, так как скрытая информация не может быть использована другими объектами или классами. Понятие повторного использования программного обеспечения стало возможным благодаря использованию *наследования*, когда подкласс может наследовать методы из родительского класса, а также разработки

библиотеки классов, которые могут быть разработаны и включены в программное обеспечение.

Первым языком, включенным в объектно-ориентированное программирование был язык Симула (SIMULA). Языки Smalltalk (Смоллток), разработанный компанией Xerox Parc и Эйфель, продемонстрировали концепцию объектов. С тех пор многие языки были дополнены объектно-ориентированной парадигмой. Например, C ++ - это интеграция языка C (Си) и объектно-ориентированная парадигма, язык Клос (Common Lisp Object System — «объектная система Common Lisp'а»), является интеграцией языка Лисп и объектно-ориентированной парадигмы программирования. Множество вариаций языка Пролог имеют библиотеку для объектно-ориентированного программирования. Современные скриптовые языки, такие как Python (Питон), Ruby (Руби), и PHP, имеют встроенное объектно-ориентированное программирование от своих императивных предков. Фортран 2008 и Кобол 2002 - последние эволюции языков Фортран и Кобол, которые имеют интегрированную объектно-ориентированную парадигму с более ранними императивными версиями. Язык Java является преемником C ++, которая интегрирует парадигмы интернет-программирования и объектно-ориентированные парадигмы программирования. В последние годы язык X10 является параллельным объектно-ориентированным языком программирования, который был разработан исследователями IBM для разработки программного обеспечения высокого уровня на компьютерах с массово-параллельной архитектурой.

1.8.4 Парадигма параллельного программирования

В течение десятилетий 1980-х годов, с тех пор как программное обеспечение начало становиться больше, аппаратная технология стала меняться также очень быстро. Множественные быстрые процессоры становятся доступны в одном компьютере, а также разработка в области компьютерных сетей разрешала обмен информацией между несколькими компьютерами.

На основе этого нового продвижения, независимые подзадачи могут быть отображены на отдельных процессорах с целью повышения эффективности выполнения программ. Развитие шло по двум направлениям: (1) р аспараллеливающие компиляторы, которые могли бы принять последовательную программу в качестве входных данных, и автоматически преобразовывать её в параллельной версии для параллельного выполнения и

(1) (2) включающие программные конструкции высокого уровня, такие как *поток выполнения, разветвление, цикл активного ожидания*, а также *удаленные вызовы процедур* в существующих языках. Во всех этих конструкциях, процесс- активная часть программы, выполняемой на CPU, - может запустить одну или несколько одновременно работающих подзадач с целью повышения эффективности выполнения.

Параллельные аналоги многих языков с парадигмой параллельного программирования получили общее название как «Concurrent X» или «Parallel X», где X - это имя существующего языка. Например, интеграция параллельных конструкций программирования в Си было названо «Concurrent C»; интеграция параллельных конструкций программирования в Паскаль была названа как «Concurrent Pascal»; интеграция параллельных конструкций программирования в Прологе была названа «Concurrent Prolog»; а интеграция параллельных конструкций программирования в Фортран была названа «Parallel FORTRAN.»

Параллельные программные конструкции становятся все более распространенными с развитием многоядерных процессоров на основе современных персональных компьютеров. Современные программы и языки включают многие из этих параллельных конструкций для повышения эффективности исполнения программ. Например, использование потоков выполнения является довольно распространенным явлением в современных языках, как в качестве интерфейса существующих библиотек потоков выполнения или в виде встроенной библиотеки. Например, обширная библиотека была предусмотрена для C, C ++ и Java.

1.8.5 Парадигма визуального программирования

Парадигма текстового программирования использует только одно измерение. Тем не менее, мы, люди, очень хорошо воспринимаем понятие близости, а не только последовательное одномерное программирование. Большинство разработанных языков программирования являются текстовыми и страдают от этого ограничения секвенциальности, вызванного одним измерением, присутствующим в текстовом представлении программ.

В конце 1980-х годов, начиная с более ранней работы с языком Smalltalk, парадигма визуального программирования взяла несколько направлений, чтобы обеспечить дружественный интерфейс. Некоторые предложения

были свободны для языков программирования, используя символическое представление для различных данных абстракций и управления абстракциями при низкоуровневом программировании. Тем не менее, попытка включить визуальное программирование было ограничено способом оперирования Drag-and-drop (буквально тащи-и-бросай; Бери-и-Брось), чтобы обеспечить удобные интерфейсы и анимации. В последнее время визуальное программирование использовалось в таких языках, как C # для событийно-ориентированного программирования: символы, соответствующие событиям и атрибутам объекта перетаскиваются, чтобы сделать сложный сценарий взаимодействующих объектов. На низком уровне эти события и взаимодействующие объекты автоматически переведены на низкоуровневую текстовую версию языка. Аналогичным образом визуальные образовательные языки, такие как Alice (Элис) используют визуальное программирование для разработки кода для анимации и языки, такие как SMIL, VRML и Java3D используют визуальное программирование для основанных на интернет-технологиях мультимедийных презентаций и анимации.

Использование парадигмы визуального программирования для крупномасштабных общих целей низкоуровневого программирования остается бездействующим по следующим причинам: (1) синтаксический анализ сложности в двумерных плоскостях, (2) сложность человеческого понимания крупномасштабных программ в двумерных плоскостях, и (3) трудность представления больших визуальных программ для программистов. Также не существует стандартизации символов, используемых в визуальном программировании.

1.8.6 Парадигма мультимедийного программирования

Парадигма мультимедийного программирования подразумевает интеграцию различных режимов визуализации: текста, изображения, аудио, видео и жестов. Люди взаимодействуют друг с другом, используя все эти сигналы. Без этих подсказок, наша коммуникация друг с другом и наше восприятие предметов и явлений реального мира было бы неполным. В начале 1990-х годов, с развитием веб-программирования и разработки комплексных форматов для аудио, изображений и видео, стало возможным встраивать мультимедийные объекты и видеоклипы в программы для упрощения визуализации и восприятия. Видео может быть представлено в виде последовательности кадров, где каждый кадр представляет собой набор, возможно, взаимодействующих объектов. Существуют языки, такие как Alice (Элис) и язык моделирования

виртуальной реальности (VRML), которые создают 3D-анимированные объекты и движения для моделирования виртуальной реальности. В последние годы многие из таких 3D языков моделирования, как X3D и Java3D получили развитие, которое интегрировало вычисления и 3D-моделирование для анимации в реальном времени через Интернет.

1.8.7 Парадигма Веб-программирования

Появление Интернета в начале 1990-х годов предоставило нам огромную возможность для обмена данными, изображениями, аудио- и видеоматериалами, базами данных и мобильными кодами, расположенными на удаленных веб-сайтах. Также нам была предоставлена возможность мобильности кода и данных. Если удаленный ресурс не хочет совместно использовать код, он может вычислить данные в источнике, и передавать полученные данные. С другой стороны, если сервер не хочет быть перегруженным, он посылает код клиенту, чтобы выполнить вычисление в конце. Веб-программирование стало большим двигателем для мультимедийной визуализации и оказало большое влияние на финансовые вычисления, такие как вычисления фондовых рынков и банковского дела. Существует множество языков, созданных на базе Интернет, таких как Java и SMIL и языков веб-разработки, таких как PHP, Javascript и XML (расширяемый язык разметки). XML стал популярным промежуточным языком для представления баз данных, вычислений и моделирования через Интернет. Java имеет популярную абстрактную машину среднего уровня, которая называется *виртуальной машиной Java (JVM)*.

Java программы интерпретируются с использованием виртуальной машины Java, которая схожа с языком ассемблера для машин с нулевым адресом. Машины с нулевым адресом не используют регистр или адрес памяти в инструкции. Скорее, это стековая вычислительная машина, где операнды помещаются в список оценки, выталкиваются, оцениваются, и отбрасываются обратно в стек. Причиной использования машины с нулевым адресом является то, что язык Java был разработан, чтобы быть реализованным на встроенном компьютере в современных повседневных устройствах, таких как микроволновые печи, умные дома и холодильники. Машины с нулевым адресом являются наиболее распространенными абстрактными машинами, которые могут быть запущены на всех встраиваемых компьютерах.

Проблемой машины с нулевым адресом является то, что инструкция высокого уровня переводится на многие другие инструкции, тратя

впустую цикл синхронизации в процессоре. Для того, чтобы ускорить процесс выполнения, был разработан новый тип *JIT компиляции,* в котором известные библиотеки классов языка высокого уровня, таких как Java или C # компилируются в родном двоичном коде. Когда программный фрагмент языка высокого уровня переводится на эквивалент низкого уровня, существуют две возможности: (1) перевести код к скомпилированному родному двоичному коду для более быстрого выполнения или (2) использовать абстрактный интерпретатор машины, такой как виртуальная машина Java или .NET (для операционной системы Microsoft Windows). JIT скомпилированные программы выполняются быстрее, чем интерпретированные в виртуальную машину Java коды.

1.8.8 Парадигма событийно-ориентированного программирования

События - это случаи , создающие условия, которые вызывают некоторые действия или вычисления. Например, нажатие кнопки на мыши является событием; перемещение компьютерной мыши над изображением является событием; и инструмент достижения порогового значения является событием. Разницей между традиционным вводно-ориентированным программированием и событийно-ориентированным программированием является то, что вводно-управляемое программирование запрашивает некоторые данные ввода, ожидает ввода, а затем принимает действие, основанное на значении входного сигнала. В противоположность этому, событийно-ориентированное программирование никогда не ждет никакого входного сигнала от внешнего мира. Тем не менее, оно реагирует на один или несколько событий, как только они происходят. Событийно-ориентированное моделирование может быть использовано для моделирования реальных явлений, поскольку события могут стать причиной для дальнейших каскадных событий, приводящих к дополнительным действиям. Понятие событийного программирования имеет свои корни в программе Симула (SIMULA) разработанной в начале 1970-х годов. Тем не менее, в последние годы она стала популярной вследствие графических и веб-взаимодействий. Более одного современного языка, такого как C #, используется событийное программирование для моделирования и анимации для графических пользовательских интерфейсов.

1.8.9 Интеграция парадигм программирования

Современные языки не привязаны к одной лишь парадигме

программирования; несколько парадигм программирования встроены в язык. Например, C++ поддерживает парадигму императивного программирования и парадигму объектно-ориентированного программирования. Visual C++ также поддерживает парадигму визуального программирования. C# поддерживает парадигмы императивного, объектно-ориентированного, событийного программирования, и поддерживает широкие возможности визуализации - функции парадигмы мультимедийного программирования. Java содержит множество характеристик из парадигм императивного, объектно-ориентированного, веб-программирования, параллельного и событийного программирования. Scala и Ruby объединяют парадигмы функционального и объектно-ориентированного программирования. Язык программирования X10, разработанный компанией IBM объединяет парадигмы императивного, объектно-ориентированного и параллельного программирования. Современные скриптовые языки PHP и Python объединяют парадигмы императивного, структурного и объектно-ориентированного программирования в одно целое. Подробный перечень парадигм программирования, поддерживающихся в некоторых популярных языках программирования, приведен в Приложении I.

Каждая парадигма программирования имеет как преимущества, так и недостатки. Например, понятие объектов в объектно-ориентированном программировании, наряду с сокрытием информации и понятием модулей, подходит для крупномасштабного развития программного обеспечения и в настоящее время принято многими старыми языками, такими как Фортран (в последней версии, Фортран 2008), которые развились со временем, чтобы включать в себя многие хорошо зарекомендовавшие себя функции, такие как реализации на основе списков, рекурсии и объекты в течение определенного периода времени. Аналогичным образом, язык COBOL (последняя версия, COBOL 2002) стали включать объекты. Если язык с течением времени перестает развиваться, то программисты прекращают использование данного языка из-за отсутствия возможностей, и старые языки заменяются новыми языками, поддерживающими популярные парадигмы программирования.

1.9 КЛАССИФИКАЦИЯ ЯЗЫКОВ

Языки могут быть классифицированы по множеству критериев, таких как парадигмы программирования, модели реализации, понятия типов, используемых в языках программирования, а также их предполагаемых предметных областей. Тем не менее, четкое разделение отсутствует во

многих современных языках, поскольку они включают совокупность признаков и парадигм с целью использовать основные преимущества.
Классификация на основе парадигм программирования

Мы уже обсудили различные парадигмы программирования. Например, ранние версии языков Фортран, Си, Алгол 60, Паскаль, АДА и PL/I были императивными языками; ранние версии Лисп, ML и Scheme - популярные языки функционального программирования; Пролог - популярный язык логического программирования; а C++, Java, Scala, Ruby, Python и X10 - представители языков с множеством парадигм программирования. Так, C++ - это объединение парадигм императивного и объектно-ориентированного программирования; CLOS («объектная система Common Lisp'а») - это интеграция парадигм функционального и объектно-ориентированного программирования; Concurrent C, Concurrent Pascal, и Parallel Fortran - это примеры интеграции параллельного и императивного программирования; и так далее. Потоки выполнения Java допускают параллельное программирование. Таким образом, Java является интеграцией парадигм императивного, объектно-ориентированного, параллельного, событийно-ориентированного и веб-программирования. Microsoft C# - это интеграция парадигм императивного, объектно-ориентированного и событийно-ориентированного программирования. В сочетании с .NET, Microsoft C# также подходит для веб-программирования. Последние языки, такие как Scala и Ruby, поддерживают парадигмы множественного программирования. В будущем будет развиваться все больше таких языков, которые будут интегрировать несколько парадигм программирования. Более старые языки включали параллельное программирование и объектно-ориентированное программирование.

1.9.2 Классификация на основе реализации

Языки могут также быть классифицированы на основе моделей реализации. Есть четыре модели широкой реализации: (1) статическая реализация, (2) реализация на основе списков,

(1) (3) реализация, основанная на переполнении хипа (heap - куча), и (4) интегрированная модель реализации, которая интегрирует статические, стековые, и хиповые распределения памяти для повышения эффективности исполнения. Как описано в главе 5, статическая реализация распределяет память для каждой объявленной структуры данных и

переменных во время компиляции, и нет никаких шансов на рост памяти во время выполнения. *Реализация на основе списков* использует список (стек), называемый также *«управляющий стек»*, чтобы разделить память для локальных переменных и других требований во время выполнения программы, таких как различные указатели, необходимые для запуска вызываемой процедуры. Преимуществом стековой реализации является то, что стек может увеличиваться и уменьшаться во время выполнения, основываясь на памяти цепи необходимоых вызываемых подпрограмм. Это расширение во время рабочего цикла разрешает динамический рост памяти, а также поддерживает реализацию рекурсивных процедур и повторного использования памяти. Модель реализации, основанной на переполнении хипа, использует центральную общую область памяти, доступную для всех процедур; памяти нужны структуры данных, которые могут быть заимствованы из хипа тогда, когда это нужно, и выпускать динамически во время выполнения, либо вручную с помощью явных действий программиста или автоматически после того, как структура данных больше не используется. Реализация, основанная на переполнении хипа, допускает распределение динамических структур данных, таких как объекты в объектно-ориентированном программировании, а также структур рекурсивных данных, таких как связные списки и деревья.

Статическая реализация не поддерживает рекурсивные процедуры, рекурсивные структуры данных, такие как связные списки и деревья, и динамические объекты. Тем не менее, статическая реализация использует абсолютный адрес памяти, используя доступ к одной памяти для доступа к элементу данных. Следовательно, статическая реализация может получить доступ к элементам данных быстрее, чем реализация на основе стека и реализации на основе переполнения хипа, использующие основанные на указателях доступы, которые эквивалентны более чем одному доступу к памяти.

Размер стека в реализации на основе стека ограничен только памятью, допустимой операционной системой. Ячейки памяти, необходимые для процедуры, могут быть выделены во время выполнения, когда вызывается соответствующая процедура. Ячейки памяти располагаются на верхней части стека на основе вызывающего шаблона процедуры и восстанавливаются после того, как вызываемая процедура закончена. Это позволяет *повторно использовать память*, поскольку тот же объем памяти может быть использован при вызове другой процедуры. Подводя итог, можно сказать, что есть два основных преимущества реализации на основе стека: выполнение рекурсивной процедуры и повторное

использование памяти. Тем не менее, механизм адресации при реализации на основе стека использует указатели и смещения, как описано в главе 5. Следовательно, по сравнению со статической реализацией, реализация на основе стека имеет дополнительные накладные расходы на доступ к переменным и структуры данных.

Одним из основных недостатков данной реализации является то, что рекурсивные структуры данных, такие как связанные списки и деревья, а также структуры с динамическими данными, такие как динамически создаваемые объекты не до сих пор могут быть реализованы . Существуют две основные проблемы в структуре рекурсивных данных и динамических объектов, которые препятствуют их реализации при реализации на основе стека. Первая проблема заключается в неопределенности при создании структуры данных во время компиляции, а второй проблемой является размер структуры данных. В случае структуры рекурсивных данных, во время компиляции не ясно, сколько памяти будет необходимо во время выполнения задачи. В зависимости от вызова и вводных данных, структуре рекурсивных данных, возможно, потребуется быть вызванной несколько раз в течение различного количества ячеек. Кроме того, во время компиляции не ясно, какой объект будет создан во время выполнения. К тому же , структуры рекурсивных данных и динамические объекты могут иметь срок жизни по истечении срока процедуры, в которой они были созданы. По этой причине используется общая область памяти, называемая *хипом*. Хип - это банк памяти, из которого процессы могут одалживать память, тогда и так, как это необходимо и вернуть память, когда их потребность в этом будет закончена. Одна структура логических данных может быть распределена по всему хипу в различных блоках памяти, соединенных с помощью указателей. Эти блоки памяти выделяются в разное время в ответ на запрос, направленный во время выполнения программы. Удаление структуры рекурсивных данных или динамического объекта эквивалентно освобождению памяти в хипе. После того, как память освобождается, сборщик мусора очищает память, так что она может быть использована другими процессами. Обратите внимание, что *этот хип отличается от сортирующего дерева*, о котором *вы узнали в структуре данных*.

Реализация на основе хипа медленнее, чем реализация на основе стека, так как есть (1) накладные расходы распределения времени выполнения объектов, (2) накладные расходы чрезмерного использования указателей для обхода структуры данных в куче, и (3) накладные расходы по

утилизации памяти для повторного использования.

Современные языки используют все три метода разделения памяти, так как они хотят дать все возможности, наряду с максимально возможной эффективностью исполнения. Например, такой современный язык как C++ использует статические переменные, которые используют статическое распределение, реализацию на основе хипа для обработки рекурсивных процедур и повторное использование памяти, а также использует динамическую память для динамических объектов данных и структур рекурсивных данных. Языки программирования также могут быть классифицированы согласно типу системы, их поддерживающей. Существуют две основные классификации типов: мономорфный тип и полиморфный. Как следует из названия, функция в языке мономорфного типа может обрабатывать только один тип структуры данных. Например, если мы напишем функцию для подсчета количества элементов в связном списке, тогда множественные аналогичные функции должны быть написаны на мономорфном языке, в зависимости от того, рассчитываем ли мы список целых чисел или список чисел с плавающей точкой из списка другого типа данных. Тем не менее, только одна общая функция должна быть написана на полиморфном языке, который будет адаптироваться к различным типам связных списков. Такие языки как Фортран, Алгол, Паскаль и Си - мономорфные языки: они поддерживают только ограниченное количество полиморфизма в форме принуждения и перегрузки, как описано в главе 7. Многие современные языки, такие как C++, Лисп, ML, Хэскелл, Java и Пролог - полиморфные языки.

Еще один интересный способ классификации языков программирования заключается в определении их по классу предметов, в которых они лучше, которая зависит от функции языка в которой нуждается предметная областью. Язык, который отвечает всем требованиям, предъявляемым к предметной области, может быть классифицирован как язык подходящий для этого домена. Например, C++ является хорошим языком программирования для разработки большого программного обеспечения. Тем не менее, это не хорошо для режима программирования в реальном времени . Точно так же, Фортран является хорошим языком для научного программирования.

1.10 КРАТКИЕ ВЫВОДЫ

Программа имеет три основных компонента: *логика*, *абстракция* и *контроль*. Разные языки хороши для разных предметных областей. Критерии для хороших языков программирования зачастую противоречат

друг другу. Разные парадигмы программирования, такие как парадигма императивного программирования, парадигма функционального программирования, парадигма логического программирования, парадигма объектно-ориентированного программирования, парадигма визуального программирования, парадигма параллельного программирования, парадигма событийно-ориентированного программирования, парадигма мультимедийного программирования и парадигма веб-программирования, могут быть охарактеризованы различными функциями. Вместе с развитием различных парадигм программирования, языковые ученые обсудили и определили хорошие черты различных парадигм программирования. Современные языки объединили эти черты и добавили парадигму множественного программирования. Например, язык Java - это язык, который включает в себя парадигмы императивного, объектно-ориентированного, событийного, параллельного и веб-программирования.

Языки высокого уровня должны поддерживать абстракции данных и управления для лучшего понимания, управления и развития программного обеспечения. Языки должны также поддерживать модульность таким образом, чтобы эффект изменения одного модуля был локализован в этом модуле. Мы обсудили три различных механизма перевода: перевод, обеспечивающийся переводчиком, компиляцию, обеспечивающуюся компиляторами и JIT-компиляцию, обеспечивающуюся JIT-компиляторами. Скомпилированные коды выполняются наиболее эффективно, так как процесс перевода не является частью исполнения. Переводчики выполняют программу медленно, так как перевод и выполнение пересекаются в момент выполнения. Веб-языки используют виртуальную машину и JIT-компиляцию чтобы перевести код низкого уровня для выполнения. Эффективность выполнения JIT-компиляции лежит между скомпилированным кодом и интерпретируемым кодом.

Для реализации языка программирования, используются различные абстрактные модели низкого уровня. Статическая реализация использует прямой доступ к памяти. Однако из-за отсутствия поддержки роста памяти, статическая реализация не в состоянии обрабатывать рекурсивные процедуры, повторное использование памяти, структуры рекурсивных данных, таких как связные списки и деревья, а также динамические объекты данных. Реализация на основе стека позволяет выполнять рекурсивные процедуры и повторное использование памяти. Однако реализация на основе стека не может обрабатывать рекурсивные структуры данных и динамических объектов данных. Для того, чтобы

обрабатывать рекурсивные структуры данных и время выполнения создания динамических объектов данных, которые могут быть выделены по частям, в зависимости от потребностей программы, нам необходимо использование *хипа* - *общей области памяти*, совместно используемой всеми *подпрограммами*.

Языки могут быть классифицированы, используя много различных критериев, таких как (1) парадигма программирования, (2) реализация, и (3) тип. Однако, классифицировать современные языки весьма сложно. Современные языки программирования извлекают пользу из совокупности признаков, и не могут быть отнесены к одной категории.

2.6 ОЦЕНКА

1.11.1 Концепция и Определения

Абстракция; алгоритм; альфа-тестирование; код сборки; бета-тестирование; бинарный код; оптимизация кода; компилятор; параллелизм; параллельное программирование; условный переход; контроль; контроль абстракции; управление потоком; блок-схема управления; абстракция данных; итерация на основе данных; парадигма декларативного программирования; определенная итерация; разрушительное обновление; парадигма событийно-ориентированного программирования; функциональная эквивалентность; парадигма функционального программирования; оператор безусловного перехода (GOTO оператор); реализация на основе хипа; императивный язык; неопределенная итерация; сокрытие информации; промежуточный код; парадигма интернет-программирования; переводчик; JIT-компиляция; лексический анализ; линкер; загрузчик; парадигма логического программирования; модульность; мономорфный тип; парадигма мультимедийного программирования; парадигма объектно-ориентированного программирования; ортогональность; синтаксический анализ; полиморфный тип; портативность; предметная область; понимание программы; читаемость; цикл разработки программного обеспечения; разработка программного обеспечения; эволюция программного обеспечения; обслуживание программного обеспечения; повторное использование программного обеспечения; реализация на основе стека; статическая реализация; таблица символов; безусловный переход; парадигма визуального программирования; возможность записи.

3.7.2 Решение Задач

1. Напишите программу, используя циклы «FOR» и «WHILE», чтобы построить цикл «FOR EACH», как указано ниже. Предположим, что «a» представляет собой совокупность структуры данных и функция «size(a)» возвращает количество элементов в «a», а в следующем (a) возвращает следующий элемент структуры данных «a.»

```
foreach x in a {
    y = f(x); print (x, y);
    }
```

1. 2. Напишите блок-схему управления для следующего кода.

```
    i = 0; j = 10;
    for (k = 0; k = < 9; k++) a[k] = 0;
while (i < 5) { j = j – 1;
        for (k = 0; k < = j; k++) {
            a[k] = a[k] + 7;
            if (a[k] mod 2 == 0} print
            ('четное число');} i = i + 1}
            while (i < 5) { j = j – 1;
        for (k = 0; k < = j; k++) {
            a[k] = a[k] + 7;
            if (a[k] mod 2 == 0} print
            ('четное число');} i = i + 1
            }
```

3.7.3 Развернутый ответ

3. 3. Объясните три основные компонента в программе: логику, контроль и абстракцию.

4. 4. Объясните преимущества контроля и абстракции данных в языках программирования.

5. 5. Сравните и сопоставьте различные критерии хорошего языка программирования.

6. 6. Объясните, используя примеры из различных предметных областей и противоположные критерии для хорошего языка

программирования, почему не может быть универсального хорошего языка программирования.

7. 7. Как «GOTO» связан с пониманием программы? Объясните, используя рисунок.

8. 8. Объясните цикл развития программного обеспечения и его эффект на развитие языков программирования.

9. 9. Объясните характеристики различных парадигм программирования.

10. 10. Каковы различные категории распределения памяти в реализации языка программирования? Объясните преимущества и недостатки каждой категории.

11. 11. Что вы понимаете под JIT-компиляцией? Чем она отличается от традиционной интерпретации и компиляции? Объясните, используя рисунок.

12. 12. Почему веб-языки используют JIT-компиляцию? Объясните.

13. 13. Какие существуют схемы категоризации языков программирования? Кратко опишите каждую классификацию.

14. 14. Что вы понимаете под блоком-схемой последовательности выполняемых команд? Объясните основные компоненты такой блока-схемы.

Глава 2

БАЗОВЫЕ КОНЦЕПЦИИ

Ранние знания по программированию; курс структуры данных; курс дискретной структуры

В этой главе описываются концепции компьютерной архитектуры, математические понятия, концепции структуры данных, а также абстрактные вычислительные концепции, необходимые для понимания абстракции и реализации языков программирования. Эти понятия не обязательно должны быть изучены в определенной последовательности. Вам может понравиться изучать концепции по мере необходимости в качестве справочного материала в разных главах.

2.1. МАШИНА ФОН НЕЙМАНА

Современные компьютеры основаны на абстрактной машине низкого уровня, которая называется машиной фон Неймана. Произошло это после того, как математик Джон фон Нейман предложил компьютерную модель, хранящую программу. Понимание машины фон Неймана очень важно для нас, потому что машинные инструкции низкого уровня разработаны для выполнения на машине фон Неймана. Программные переводчики переводят конструкции высокого уровня в аналогичные инструкции низкого уровня. Конструкции высокого уровня, такие как условные операторы, циклы «WHILE DO» и «FOR», переводятся в последовательность команд низкого уровня в машине фон Неймана. Применение машины фон Неймана и наборов команд низкого уровня станет яснее, когда мы будем обсуждать значение конструкций высокого уровня с точки зрения абстрактных инструкций низкого уровня в главе 3 и переводить управления абстракций в последовательность абстрактных инструкций низкого уровня в главе 5.

Машина фон Неймана содержит два основных компонента: (1) память, которая хранит программу и данные, и (2) центральный процессор (ЦП), который извлекает инструкции и обрабатывает данные, основанные

на инструкциях. Инструкции выбираются по одной центральным процессором с помощью подключения шины - высокоскоростным соединением между памятью и центральным процессором. Процессор интерпретирует команду и загружает данные из памяти, используя ту же шину. Данные

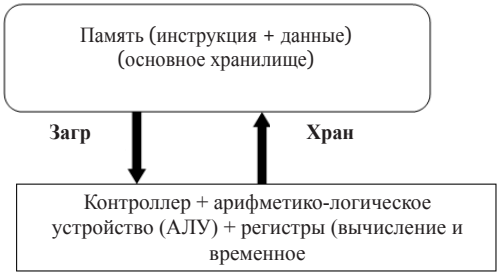


Рисунок 2.1 Машина фон Неймана

обрабатываются в арифметико-логическом устройстве (АЛУ) в центральном процессоре, и полученные в результате данные сохраняются обратно в памяти, как показано на рисунке 2.1. Часто используемые элементы данных или их ссылки хранятся в аппаратных регистрах для быстрого доступа к ним. Машина фон Неймана имеет программный счетчик, который содержит адрес памяти следующей выполняемой инструкции. Счетчик команд увеличивается на одну единицу, после получения текущей инструкции. Тем не менее, его значение может быть изменено с помощью условных и безусловных операторов перехода.

Инструкции могут быть классифицированы как загрузка из памяти (загрузка); хранение в памяти (хранилище); выполнение арифметических вычислений сложения, вычитания, умножения и деления; логические вычисления, такие как логический оператор «AND», логический оператор «OR», исключающий оператор «OR», и отрицание; сравнение двух значений; условные или безусловные переходы на другие несмежные инструкции путем изменения значения счетчика программы и проверки состояния различных флагов системного уровня. Различные флаги хранятся в регистре, называемом *слово состояния программы* (PSW). В дополнение к этим категориям инструкций, наборы инструкций также поддерживают различные механизмы адресации для устранения операнда , в зависимости от архитектуры компьютера.

2.1.1 Адресные механизмы

В машине фон Неймана данные или адреса ячеек памяти могут временно храниться в регистрах или в памяти (RAM). Аналогичным образом, область памяти может также содержать данные, а также адрес другого местоположение в памяти. Если память содержит данные, то при обращении к ней инструкцией, используя разовый доступ к памяти, называется *прямой доступ*. Однако, если ячейка памяти содержит адрес другой ячейки памяти, то два обращения к памяти необходимы для загрузки данных в центральный процессор, и этот метод называется *косвенным доступом*. Помимо прямых и косвенных доступов, смещение может быть добавлено или вычтено по адресам, хранящимся в регистре или другим адресам для расчета нового адреса. Этот метод на основе смещения используется для доступа к подмассивам сложных структур данных, где нужно сохранить базовый адрес первой ячейки памяти структуры данных и вычислить адреса подмассивов путем добавления смещения подмассива к базовому адресу.

Компьютер может поддерживать 0-адрес, 1-адрес, 2-адрес, или 3-адреса, в зависимости от архитектуры компьютера. Как показано в таблице 2.1, число адресов задается максимальным количеством аргументов набора инструкций на уровне сборки в

Таблица 2.1 Различные типы адресных механизмов

Тип инструкции	**Тип операций**
Трехадресные машинные команды: *<instruction-name> <src1>, <src2>, <dst>*	Инструкция в трехадресной машине. Инструкция может быть любой арифметической или логической диадично операции
Двухадресные машинные команды: *<instruction-name> <src1>, <src2>*	Инструкция в двухадресной машине. Инструкция может быть любой арифметической или логической диадично операции. Назначение такое же, как второй аргумент
Одноадресные машинные команды: *<instruction-name> <src>*	Инструкция может быть любой арифметической или логической диадично операции. Один из регистров по умолчанию является *аккумулятором*, который действует в качестве пункта назначения
Машинные команды с нулевым адресом: *<instruction-name>*	Загрузка, хранилище, сложение, вычитание, умножение, деление, и так далее.
	Использует оценку на основе стека. Аргументы в пользу операции, отбираются из верхней части стека

машине низкого уровня при адресации ячеек памяти или регистров. Для выражений, включающих диадические операции, есть два входных аргумента и один выходной аргумент. Например, для механизма *add_integer* потребуется три адреса: два для хранения входных аргументов и один для выходного аргумента. Аргументы, которые хранят входные значения, называются *источником*, а аргумент, который содержит выходное значение называется *назначением*.

Большинство современных процессоров поддерживают трехадресные наборы инструкций. Тем не менее, множество адресов не могут быть размещены с 16-битным размером слова. *Двухадресные машины* имеют регистр специального назначения под названием *аккумулятор*, который всегда используется в качестве пункта назначения и не упоминается в качестве одного из аргументов команд, чтобы сохранить количество битов, необходимых для адресации механизма. Оно также может быть явным образом использовано в качестве одного из исходных регистров. *Одноадресные машины* неявно имеют первый исходный аргумент в качестве аккумулятора, а адресат всегда является аккумулятором. Аккумуляторы не упоминаются в инструкции, чтобы сохранить число битов; неявное использование аккумулятора как для источников, так и для назначения дальнейшего сокращения вниз по длине инструкции.

Машины с *нулевым адресом* не имеют каких-либо конкретных аргументов и используют встроенный стек для хранения аргументов. Инструкции в машинах с нулевым адресом это: загрузки, хранилище, операции *push, pop*, *сложение*, *вычитание* и так далее. Например, когда дается команда сложения, два верхних элемента данных *извлекаются* из стека вычисления в АЛУ, оцениваются, и результат *отбрасывается* назад на вершину стека вычисления. Инструкции *загрузки* и *хранилища* могут извлекать данные из локальной переменной к вершине стека вычисления или хранить данные из верхней части стека вычисления к локальной переменной. Машины с нулевым адресом могут быть реализованы с помощью машины с более старшим адресом. Инструкции в машинах с более старшим адресом могут быть переведены на две или более команды более низких адресов. Машины с нулевым адресом были использованы для выполнения программ Java в виртуальной машине Java (JVM).

Пример 2.1

Этот пример иллюстрирует перевод арифметического выражения $X = A + B$, где

X, A, и *B* - это целые переменные. Предполагается, что эти переменные отображаются в ячейки памяти. Знак процента «%» начинает комментарий. Инструкция в трехадресной машине выглядит следующим образом:

```
integer_add A, B, X % Add data loaded from memory locations A
                    % and B, and store back in X.
```

Так как двухадресные машины не имеют возможности иметь три адреса, инструкции 3-адресных машин разделены на последовательность команд 2-адресных инструкций. Второй аргумент также выступает в качестве пункта назначения. Соответствующие инструкции низкого уровня в 2-адресной машине выглядят следующим образом:

```
load A, R0 % load the content of memory location A into the
           % register R0
integer_add B, R0 % add the contents of the memory location B
                  % and the register R0
store R0, X % store the content of R0 into the memory location X
```

Одноадресные машины неявно используют аккумулятор в качестве одного из входных аргументов и пункта назначения. Соответствующий набор низкоуровневых одноадресных инструкций заключается в следующем:

```
load A % load the content of memory location A into the
       % accumulator
integer_add B % add the contents of the memory location B with
              % the content of the accumulator
store X % store the content of accumulator in the memory
        % location X
```

Машина с нулевым адресом (как указано в наборе команд виртуальной машины Java) загружает значения ячеек памяти *A* и *B* в встроенный стек, добавляет два значения стека, и оставляет сумму на вершине стека, а затем хранит в верхней части стека в ячейку памяти *X*. Операция загрузки представляет собой последовательность из двух операций: сначала он загружает адрес ячейки памяти на вершине стека с помощью инструкции «load_literal», а затем загружает значение ячейки памяти с помощью команды загрузки.

```
load_literal 3 % Push the address of X on top of the
               % evaluation stack.
load_literal 1 % Push the address of A on top of the
               % evaluation stack.
load % Pop the address of A and push the content of A on top
     % of the evaluation stack.
load_literal 2 % Push the address of B on top of the evaluation
               % stack.
load % Pop the address of B and push the content of B on top
     % of the evaluation stack.
integer_add % Pop top two elements from the stack, add them,
            % and push the result on the stack.
store % Pop the result, pop the address X, and store result
      % into the memory location X.
```

Как видно из примера 2.1, необходимо больше инструкций в инструкциях более низкого адреса для выполнения той же задачи. Так как каждая команда означает выборку памяти, машинам с более низким адресом нужно больше памяти при выполнении выборок на машинах с более высоким адресом. Таким образом, виртуальные машины с нулевым адресом работают медленнее, чем родной скомпилированный код на машинах с более высоким адресом.

2.2 Концепции дискретных структур

В этом разделе мы обновим выбранные концепции дискретных структур, имеющих отношение к абстрактным представлениям данных. Их актуальность станет ясна, как только мы продолжим обсуждение концепций языка программирования в следующих главах.

2.2.1 Операции над множествами

Понятия множеств и операций с множествами используются в языках программирования на различных уровнях. Множества используются в (1) определении типов в теории типов, как описано в главе 7; (2) множественном программировании, например, в таких языках, как Паскаль, Сетл (SETL), Модула, Клэр, Руби, и Питон; и (3) в библиотеках для множественного программирования во многих популярных языках, таких как Java, C ++, и Пролог.

Множество представляет собой набор уникальных элементов данных. Перестановка элементов в множестве не изменяет само множество. *Мультимножество* - это набор объектов, которые могут иметь несколько элементов данных с одинаковым значением. Например, {*1, 2, 3, 2, 4, 5, 7*}

это мультимножество, где элемент *2* повторяется трижды. Тем не менее, перестановка элементов в нем не изменяет само мультимножество.

Интересный подкласс мультимножества - это *упорядоченное мультимножество*. Каждый элемент *упорядоченного мультимножества* связан с соответствующей позицией. В упорядоченном мультимножестве {*Меера , Джон, Ли*}, *Миира* связана с позицией 1, *Джон* связан с позицией 2, и *Ли* связан с позицией 3. Перестановка элементов в упорядоченном мультимножестве изменяет упорядоченное мультимножество, потому что соответствующие позиции изменены. Например, упорядоченное мультимножество {*Меера, Джон, Ли*} - это не то же самое, что и упорядоченное мультимножество {*Джон, Меера*, *Ли*}, потому что *Меера* связана с позицией 1 в первом упорядоченном мультимножестве и связана с позицией 2 в втором упорядоченном мультимножестве. Упорядоченные мультимножества играют большую роль во многих структурах языков программирования. Структуры, такие как *последовательности, строки, файлы*, и так далее моделируются как упорядоченные мультимножества, как описано в главе 7.

Предполагаются дружественные отношения с подмножествами, объединением и пересечением множеств. Множество всех возможных подмножеств данного множества называется *показательным множеством* заданного множества. Например, множество всех подмножеств {*X, Y, Z*} задается {{} (пустое множество), {*X*}, {*Y*}, {*Z*}, {*X, Y*}, {*Y, Z*}, {*X, Z*}, или {*X, Y, Z*}}. Чтобы получить подмножество, элемент либо может быть выбран, либо или игнорирован, то есть дается две возможности. Таким образом, подмножество данного множества задается 2^N, где N - число элементов в исходном множестве.

Декартово произведение включает в себя два или более множеств и получает новое множество N-сомножеств, где N - число множеств в Декартовом произведении. Учитывая Декартово произведение $A_1 \times A_2 \times, \ldots, \times A_N$ ($N \geq 2$), первое поле из N-сомножества является элементом множества A_1 и *Ith* элементом ($I \leq N$) N-сомножества является элемент I множества. Число элементов в полученном множестве задается $|A_1| \times |A_2| \times |A_3| \times |A_4| \ldots |A_I| \times \ldots \times |A_N|$, где $|A_i|$ дает количество элементов в I множестве.

Пример 2.2

Давайте рассмотрим три множества: A = {*a, b, c*}, B = {*x, y, z*}, и C = {*1, 2, 3*}. Декартово произведение A × B × C имеет 3 × 3 × 3 = 27 возможностей следующим образом: {(*a, x, 1*), (*a, x, 2*), …, (*c, z, 1*), (*c, z,*

2), (*c, z, 3*)}.

2.2.1.2 Отображение

Отображение включает в себя ассоциирование элементов одного набора множества с элементами в другом, либо через ассоциацию «one-to-one», либо через «many-to-one». На рисунке 2.2 показано соответствие доменов D и областьи значений R. Существует пять элементов в множестве D, и шесть элементов в множестве R. Если мы определим отображение как «квадрат элементов в D» то каждый элемент домена D отображается на элемент в R. Более чем один элемент в области D может отображать тот же элемент в R. Однако, один элемент из области D не может отображать более чем один элемент в область значений R.

2.2.1.3 Изоморфизм

Два множества S_1 и S_2 *изоморфны*, если существует пара биективных функций F и G так, что существует отображение между S_1 и S_2: функция F отображает все элементы S_1 на S_2 через ассоциацию «one-to-one», функция G отображает элементы S_2 на S_1 через ассоциацию «one-to-one», а функция G является обратной функцией функции «F».

Пример 2.3

Например, возьмем два множества S_1 = {*1, 2, 3*} и S_2 = {*4, 5, 6*}. Существует функция *add_3*, которая отображает все элементы множества S_1 к каждому элементу множества S_2 через ассоциацию «one-to-one» путем добавления 3 к элементу в области S_1, где

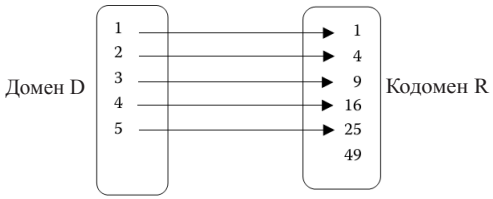


Рисунок 2.2 Отображение с помощью квадратной функции

есть соответствующая обратная функция *subtract*_3, которая отображает каждый элемент S_2 к элементу множества S_1 способом «one-to-one» путем вычитания 3. Обе функции add_3 и subtract_3 - это обратные функции друг друга.

2.2.1.4 Функции

Функция F представляет собой однозначное отображение любого элемента x ∈ области D к изображению F(x) ∈ области C. Множество изображений F(x ∈ D) ⊆ C, и называется *диапазоном* функции. *Тождественная функция* отображает элемент сам по себе, *а постоянная функция* отображает каждый элемент множества на фиксированный единый элемент. Функция называется *«one-to-one»* или *инъективной*, если для всех элементов в домене существует отчетливое изображение в области значений, то есть если $y_1 = F(x_1 \in D)$ и $y_2 = F(x_2 \in D)$, и $x1 \neq x2$ и $y_1, y_2 \in C$ означает $y_1 \neq y_2$. Функция является функцией *отображения* или *сюръективной* функцией, если для любого элемента y ∈ C, тогда существует элемент x ∈ D такой, что f(x) = y. Функция F является *биективной*, если это функция отображения и «one-to-one». Это означает, что для элемента в области значений есть уникальный образ элемента в домене. Учитывая биективную функцию F, обратная

функция F^{-1}, может быть определена как функция, отображающая уникальный образ в области значений к соответствующему элементу в области, которая является F^{-1} (F (x ∈ D)) = x.

В программировании, сюръективные функции становятся весьма важными, так как отсутствие образа определенной функции эквивалентно ошибочному условию. Несюръективная функция может стать сюръективной путем введения *нулевого символа*, обозначаемом ⊥ в диапазоне. Нулевой символ становится изображением всех символов в областях, которые не сопоставляются с любым ненулевым символом в области значений. Идея нулевого символа была использована для объяснения денотационной семантики языков программирования в главе 3 и функционального программирования в главе 9.

Программное представление функции состоит из трех частей: *переменные*, *тело выражения*, и *входные параметры*. Если задано входное значение, то входное значение связано с переменной и всюду, где в выражении есть переменная, происходит замещение. После упрощения выражения, выходное значение возвращается. Функции может быть дано имя, чтобы вызвать функцию из различных местоположений нескольких программных модулей.

Пример 2.4

Рассмотрим хорошо известную функцию *факториал(n)*, которую вы изучали в курсе структуры данных и программирования. Функция имеет переменную *n*, привязанную к данному параметру. *Тело* функции

было дано на правой части определения.

```
factorial(n) =
          if (n == 0) return(1) % base case
          else return(n * factorial(n - 1))% recursive
                                           % definition
```

Тело функции *факториала (n)* состоит из двух выражений: "*if (n == 0) then return (1); if (n > 0) return (n * factorial(n – 1))*." Оба выражения связаны через условного оператора абстракции (if-then-else), а тест (*n*> 0) подразумевает в качестве условного оператора. Когда мы вызываем *факториал* (3), значение 3 связано с переменной *n*, а затем оно заменяется всюду в теле функции, а сама функция сводится к «*if (3 == 0*), *then return 1 else return (3 * factorial(3 - 1))*.» Условие оценивается, а функция дополнительно упрощена для *return(3 * factorial(3 – 1))*. Выражение (*3 – 1*) упрощается, а функция дополнительно упрощена для *return(3 * factorial(2))*. В настоящее время оценка *function(3)* приостанавливается до того, пока *factorial(2)* не будет вычислен.

2.2.2 Булевая логика и исчисление предикатов

Булевая логика основана на ассоциировании значений истинности некоторых утверждений. Аксиомой является утверждение или предположение, которое может быть либо «истиной (true)» или «ложью (false)». Эти значения истинности могут быть комбинированны с использованием различных логических операторов, таких как логическое AND (обозначается символом «∧»), логическое OR (обозначается символом «∨»), импликация (обозначается символом «→») и отрицание (обозначается символом «¬»). Операторы логическое AND и логическое OR объединяют две логические аксиомы A и B. Как показано в таблице 2.2, A и B должны быть истинны для A ∧ B, чтобы быть истинны, и оба A и B должны быть ложными для A ∨ B чтобы быть ложными. Вывод логической операции утверждает, что A → B и, что если A является правдой, то и B тоже. Невозможно установить значение истинности B, если A - ложь. Таким образом если значение A ложно, то B может быть как истинной, так и ложью. Отрицание означает, что если аксиома A правдива, то ¬ A ложно, и если аксиома A ложна, то ¬ A истинно.

Пример 2.5

Давайте примем за аксиому A, что «люди являются инновационными» и за аксиому B, что «люди гордые». Предположим, что аксиома A и

аксиома B являются истинными; тогда A ∧ B также верно, таким образом, «Люди являются инновационными и гордыми.» Если мы рассмотрим оператора логическое OR, и если либо аксиома A истинна, либо аксиома B истинна, или если одна из них истинна, то аксиома «Люди являются инновационными или гордыми» была бы правдой.

Операторы логическое AND и логическое OR коммутативны, ассоциативны и дистрибутивны, как указано в таблице 2.3. Эта концепция широко используется в разработке программ и параллельном выполнении программ. Существует множество комбинаций булевых операций, которые являются эквивалентными. Например, теорема Де Моргана относится к операторам логическое AND с логическим OR путем использования отрицания. Закон Де Моргана гласит: (1) отрицание конъюнкции есть не что иное, как дизъюнкция отрицаний и (2) отрицание дизъюнкции есть не что иное, как конъюнкция отрицаний.

Таблица 2.2 Таблица истинности для логических операторов, используемых в языках программирования

A	**B**	**¬(A)**	**A ∧ B**	**A ∨ B**	**A → B**
Истина	Ложь	Ложь	Ложь	Истина	Ложь
Ложь	Ложь	Истина	Ложь	Ложь	Истина
Истина	Истина	Ложь	Истина	Истина	Истина
Ложь	Истина	Истина	Ложь	Истина	Истина

Таблица **2.3** Эквивалентность в булевых операциях

Тип операций	**Эквивалентность**
Отрицание	$\neg(\neg P_1) \equiv P_1$
Ассоциативность	$P_1 \wedge (P_2 \wedge P_3) \equiv (P_1 \wedge P_2) \wedge P_3$
	$P_1 \vee (P_2 \vee P_3) \equiv (P_1 \vee P_2) \vee P_3$
Коммутативность	$P_1 \wedge P_2 \equiv P_2 \wedge P_1$
	$P_1 \vee P_2 \equiv P_2 \vee P_1$
Дистрибутивность	$P_1 \wedge (P_2 \vee P_3) \equiv (P_1 \wedge P_2) \vee (P_1 \wedge P$
	$P_1 \vee (P_2 \wedge P_3) \equiv (P_1 \vee P_2) \wedge (P_1 \vee P$
Правило де Моргана	$\neg(P_1 \wedge P_2) \equiv (\neg P_1) \vee (\neg P_2)$
	$\neg(P_1 \vee P_2) \equiv (\neg P_1) \wedge (\neg P_2)$

2.2.2.1 Исчисление предиката первого порядка

Пропозициональная логика обрабатывает истинные значения простых аксиом и вывод новых высказываний с помо щью логических операторов. Если мы дополним пропозициональную логику двумя типами кванторов - квантром всеобщности (обозначается символом ∀)

и квантором существования (обозначается символом ∃) - тогда такая логика называется «логикой первого порядка.» Квантор всеобщности- это условие, которое верно для всех обозначенных элементов. После связывания свойств с каждым элементом множества, если объект является членом множества, с которым свойство было связано, объект будет также связан со свойствами. Например, если говорят, что всем мужчинам нравится жить догло , и Джон - мужчина. Естественным выводом для нас будет то, что «Джону нравится жить долго .» Давайте сопоставим эти утверждения с помощью логики первого порядка (FOPC). По правилам логики первого порядка ∀x (*мужчина*(x) → *любит_жить_долго* (x)). Если мы прочитаем эти правила, то в них будет сказано, что для всех «x», если «x» мужчина, означает «x» нравится жить долго . Джон - мужчина. Используя правило импликации, Джону нравится жить долго . Квантор существования запрашивает элемент, удовлетворяющий специфическим свойствам. Например, правило ∀x ∀y (*одноуровневый элемент*(*x, y*) → ∃z (*parent*(*x, z*), *parent*(*y, z*), *not* (*x* == *y*) утверждает, что для всех *x* и для всех *y*, *x* является одноуровневым элементом *y*, если существует *z* такой, что *z* является родительским *X*, *Z* является родительским *y*, а *x* не равен *y*. Переменная *z* - это квантор существования, как это выглядит для элемента в множестве, которое является родительским для обоих *x* и *y*.

Существует много эквивалентных свойств, касающихся двух кванторов, как показано в таблице 2.4. И квантор существования, и всеобщности коммутативны. Однако, когда квантор всеобщности смешивается с кванторами существования, то ситуация становится не так проста. Например, если свойство верно для всех элементов множества (∀*x P*(*x*), то можно сказать, что это не тот случай, когда существует такой элемент, для которого это свойство не

Таблица 2.4 Эквивалентность в кванторах

Операции	**Эквивалентность**
Коммутативность	∀x ∀y P(x, y) ≡ ∀y ∀x P(x, y)
	∃x ∃y P(x, y) ≡ ∃y ∃x P(x, y)
Двойственность	∀x P(x) ≡ ¬ ∃x (¬ P(x))
	∃x P(x) ≡ ¬∀x (¬ P(x))

является истинным (¬∃x ¬P(x)). Аналогичным образом (∃x P (x)) эквивалентно ¬ ∀x (¬ P (x)). Если мы смешаем квантор всеобщности с квантором существования, тогда закон коммутации не применяется: ∀x ∃y не эквивалентно ∃y ∀x: заявление, такое как: «Существует человек

Y для каждого человека X, которого X любит» ($\forall x \exists y$ любит (x, y)) это не то же самое, что и «Существует человек Y, которого любит каждый человек X» ($\exists y \forall x$ любит (x, y)).

Логика первого порядка лежит в основе парадигмы логического программирования, как описано в главе 10. Одним из ограничений логики первого порядка является то, что оно не может выразить отношения по поводу отношений. Исчисление высшего порядка может выразить отношения по поводу отношений. Тем не менее, обсуждение данной темы выходит за рамки данной книги.

2.2.2.2 Отношения

Подобно функциям, отношения широко используются в программировании и представлении типов, необходимых для программирования. В этом подразделе мы рассмотрим основные свойства отношений.

Бинарное отношение R определяется как собственное подмножество декартова произведения двух множеств: домена S_1 и кообласти S_2. Говоря математическим языком, отношение $R \subseteq S_1 \times S_2$, и каждый элемент $x \in S_1$ связан с элементом $y \in S_2$ через отношение R для упорядоченной пары вида $(x, y) \in R$. Связь между двумя субъектами может быть представлен в виде xRy или R (x, y).

Отношение может быть *рефлексивным, симметричным, антисимметричным* или *транзитивным*. Отношение R рефлексивно, если xRx верно для каждого элемента домена. Отношение R *симметрично*, если для каждой упорядоченной пары $(x, y) \in R$, существует другая упорядоченная пара $(y, x) \in R$. Другими словами, xRy эквивалентно yRx. Отношения *антисимметричны*, если xRy означает, что y никогда не связано с x через отношения yRx. Отношения *транзитивны*, если xRy и yRz предполагают, что существует упорядоченная пара $(x, z) \in R$. Понятие отношений широко используется в развитии фундаментальных свойств программирования. Отношение R представляет собой *отношение эквивалентности*, если R рефлексивно, симметрично и транзитивно.

Пример 2.6

Например, отношения «больше чем» (или меньше чем) *антисимметричны* и *транзитивны*. Аналогичным образом, отношения «равенство» рефлексивны, симметричны и транзитивны. Значение переменной x равно себе (xRx); если значение переменной x равно значению переменной y, то значение переменной y равно значению

переменной x (xRy подразумевает yRx); если значение переменной x равно значению переменной y, а значение переменной y равно значению переменной z, то значение переменной x равно значению переменной z (x == y и y == z предполагает x == z).

Свойства рефлексивности, симметрии и транзитивности были использованы неявно во многих операторах сравнения в обычных программах, таких как сортировка последовательности чисел. Свойство транзитивности также используется при анализе графика зависимости данных в главе 8 и при анализе эквивалентности псевдонимов переменных.

2.2.3 Рекурсия

Рекурсия - это вызов функции (процедуры) из неё же самой, непосредственно или через другие функции. Рекурсия может быть использована для определения рекурсивной функции, такой как *факториал* и *Фибоначчи*, или для рекурсивной структуры данных, таких как связный список или деревья.

Рекурсивное определение имеет, по меньшей мере, один базовый случай и, по меньшей мере, одно рекурсивное определение. Рекурсивное определение постепенно разворачивается и приближается к базовому(ым) варианту(ам). В конечном числе разверток рекурсия завершается базовым случаем, и результат вычисления передается обратно в обратном порядке к вызывающему вызову рекурсивной функции. Число вызовов рекурсивных функций определяется значением входного сигнала. Предыдущий вызов приостанавливается до того момента, пока вызов, обусловленный следующим разворачиванием, оценивается. Необходимо хранить приостановленные процедуры, нуждающиеся в неотъемлемом использовании стека для хранения ячеек памяти, необходимых для выполнения вычислений в цепи вызываемых функций.

Пример 2.7

```
factorial(0) = 1. % base case
factorial(n) = n * factorial(n - 1) % recursive
definition
```

Функция *факториал*(*n*) определяется рекурсивно *n* * *факториал*(*n* – *1*), а базовый случай факториала - *факториал*(*0*) = *1*. Вызов функции *факториал*(*4*) развертывается как *4* * *факториал*(3); *факториал*(*3*) развертывается как *3* * *факториал*(*2*); *факториал*(*2*) развертывается как

*2 * факториал*(*1*); а *факториал*(*1*) развертывается как *1 * факториал*(*0*); *факториал*(*0*) и является базовым случаем. После оценки *факториала* (*0*), значение *1* передается *факториалу* (*1*); *факториал* (*1*) проходит *1 * 1 = 1* для *факториала* (*2*); *факториал* (*2*) проходит *2 * 1 = 2* для *факториала* (*3*); и *факториал* (*3*) проходит *3 * 2* для *факториала* (*4*). Окончательное значение, 4 * 6 = 24, возвращается в конце.

Пример 2.8

Другим примером рекурсивной функции является определение «чисел Фибоначчи.» Определение чисел Фибоначчи имеет два базовых случая, как указано ниже

```
Fibonacci(0) = 1 % base case Fibonacci(1) = 1 % base case
Fibonacci(n) = Fibonacci(n - 1) + Fibonacci(n - 2) % recursive
                                                  % definition
```

Хвостовая рекурсия и итерация

Хвостовая рекурсия является важным подклассом рекурсивных определений, что требует особого внимания при изучении языков программирования. В хвостовой рекурсии рекурсивный вызов является последней выполняемой операцией в функции. Например, функция *GCD* может быть выражена с помощью хвостовой рекурсии следующим образом:

```
GCD(x, y) = gcd((bigger(x, y) modulo smaller(x, y)),
smaller(x, y)). % tail recursive definition
GCD(0, x) = x % base case
```

В приведенном выше определении, базовый вариант возвращает значение второго аргумента, если первый аргумент равен 0. Это произойдет, когда левая сторона хвоста рекурсии получит больший аргумент как множество меньших аргументов. Определение хвостовой рекурсивности вызывает функцию *GCD* рекурсивно первым аргументом в качестве остатка от деления больших и меньших аргументов, а второй аргумент в качестве исходного меньшего аргумента, который стал бы большим аргументом в новом вызове функции *GCD*.

Одним из свойств хвостовой рекурсии является то, что они могут быть смоделированы с использованием неопределенных итераций и избежать накладных расходов памяти, связанных с вызовом новых процедур рекурсивных определений. Обратите внимание, что ни определение

факториала, ни определение *Фибоначчи* не является хвостовой рекурсией: у факториала «умножение» выступает в качестве последней операции, а *Фибоначчи* имеет два рекурсивных вызова в рекурсивном определении. *Оптимизация хвостовой рекурсии* - это метод оптимизации кода для повторного использования памяти, позволяющий сократить время и издержки выполнения путем преобразования программ хвостовой рекурсии на неопределенный срок итерации. Итерационные программы снижают накладные расходы памяти и времени в стеках, которые гораздо медленнее реализуют программу и тратят пространство памяти.

2.2.3.2 Линейная рекурсия и итерация

Большой класс *линейных рекурсивных функций*- функций только с одним вызовом себя в рекурсивном определении - может быть преобразован в итерационные программы с использованием *бессрочной итерации* и *накопителей*. *Накопитель* - это абстракция, которая хранит последний вычисленный частичный результат. В итеративной версии, тот же набор переменных *используется повторно*, а вызовы рекурсивных процедур заменяются повторяющимся кодом, который начинается с базового случая и сохраняется, накапливая значение частичной выходной после каждого цикла. И, наконец, результат собирают с накопителя.

Пример 2.9

Итеративная версия функции *факториал* представлена на Рисунке 2.3. Накопитель был инициализирован как *факториал* (*0*) = 1 - базовое значение функции *факториала*; и рекурсивные вызовы были заменены итерацией. После каждого шага, значение накопителя обновляется. Это эквивалентно началу с базового случая рекурсии и построения; в данном случае не существует разворачивания рекурсивных процедур, что делало бы его более эффективным.

```
Algorithm iterative_factorial
Input: value of n;
Output: accumulator value;
{ accumulator = 1;
  for (i = 1; i =< n; i++) accumulator = i * accumulator;
  return(accumulator);
}
```

Рисунок 2.3 Итеративная версия функции *факториал*.

Конечные автоматы

Конечный автомат - это абстрактная машина для моделирования явлений реального мира при помощи моделирования различных ситуаций (состояний) в виде узлов на графах и переходов между состояниями в виде меток дуг между узлами. Автомат переходит из одного состояния в другое на основании входных значений. Все эти состояния могут не быть доступны из всех состояний. Существует одно или несколько начальных состояний, и существует одно или несколько конечных состояний. Автоматы начинают с одного из начальных состояний и заканчивают в конечном состоянии.

Пример 2.10

Система нагрева и охлаждения может быть смоделирована как конечный автомат, как показано на рисунке 2.4. Приведены три состояния: «в комнате жарко», «в комнате холодно» и «температура оптимальна» Все три состояния могут быть начальным состоянием. Тем не менее, только состояние «температура оптимальна» является конечным состоянием.

В состоянии «в комнате жарко» термостат начинает охлаждение. Результатом является переход в состояние «температура оптимальна». В состоянии «в комнате холодно», термостат начинает нагрев. Результатом является переход в состояние «температура оптимальна». Никакой сигнал не посылается, если разность между заданной температурой и температурой в помещении находится в пределах порогового значения.

Пример 2.11

Конечный автомат на рисунке 2.5 будет распознавать имена переменных в программе. Переменная определяется как английская буква, следующая за любым количеством латинских букв или цифр. Автомат содержит начальное состояние S_0 конечное состояние S_1, и состояние S_2 для обработки всех условий ошибки. Автомат стартует при условии S_0, где

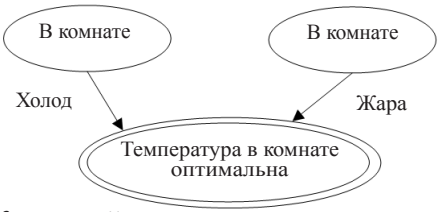


Рисунок 2.4 Конечный автомат моделирования термостата.

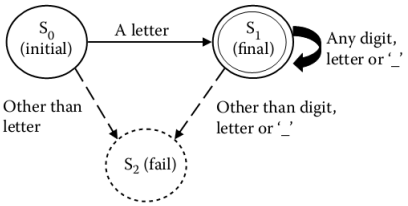


Рисунок 2.5 Конечный автомат для принятия имени переменной.

могут применяться только английские буквы для перехода к условию S_1; любые другие характеристики приведут автомат от условия S_0 к условию S_2, где будет получен сигнал об ошибочном состоянии и выполнение будет корректно прекращено. После принятия первой буквы, автомат переходит к условию S_1, где могут приняться как другая буква, так и цифра. Автомат переходит от S_1 к S_1 независимо от того, получает ли он букву, цифру или другие специальные символы, такие как "_"; в противном случае он переходит к S_2 - ошибочному условию.

2.3 КОНЦЕПЦИИ СТРУКТУРЫ ДАННЫХ

Существует множество концепций в структурах данных, которые необходимо понимать для реализации языков программирования. Некоторые из важных концепций это: деревья, используемые в обработке; стеки, неявно используемые в рекурсии и реализации механизмов языков программирования; списки и графики, используемые в некоторых эффективным схемах сборки мусора; и хэширование, используемое для неявного доступа к механизмам и объектам в реализации различных парадигм программирования.

2.3.1 Последовательность действий

Последовательность действий - это упорядоченное таким образом мультимножество элементов, в котором непосредственно предшествующий элемент в последовательности связан с положением, которое меньше ровно на один элемент от позиции опорного элемента. Например, *строка* представляет собой последовательность символов, а *файл* представляет собой последовательность объектов данных. Даже

абстрактные *стеки* типов данных и *списки* могут быть смоделированы, используя последовательность действий, как показано в следующем подразделе.

Обозначим последовательность с составляющими элементами в угловых скобках '<' и '>'. Например, *<x, y, z>* представляет собой последовательность, где элемент данных *x* связан с положением 1, элемент данных *y* связан с положением 2, а элемент данных *z* связан с положением 3. Если мы обозначим две функции -функции - *предшественника* и *преемника*- элементом данных в последовательности, тогда *предшественник(y) = x*, и *предшественник (z) = y*, так как положение элемента данных *y* больше, чем положение элемента данных *x*, а позиция элемента данных *z* больше, чем положение элемента данных *y*. Аналогично, *преемник(x) = y* и *преемник(y) = z*. Операции с последовательностями могут быть отнесены к категории извлечения элемента, вставки элемента, замены элемента другим элементом, соединения двух последовательностей, проверки, является ли последовательность подпоследовательностью другой последовательности, извлечения подпоследовательности, и замены подпоследовательности другой подпоследовательностью. Основные операции с последовательностями перечислены в Таблице 2.5.

Элемент может быть доступен с самого начала, с конца, или путем предоставления индекса элемента. Оператор *first* возвращает первый элемент последовательности, оператор *second* возвращает второй элемент последовательности, а оператор *last* возвращает последний элемент последовательности. Дополнительной операцией является выявление оставшихся подпоследовательностей после того, как элемент отпал. Оператор *rest* дает оставшимся подпоследовательностям исходной последовательности минус первого элемента, а оператор *butlast* дает подпоследовательности исходной последовательности минус последнего элемента. Оператор *select* принимает два входа-индекс элемента и последовательность-и возвращает соответствующий элемент последовательности. Оператор *cons* (аббревиатура от construct) создает новую последовательность, вставив данный элемент в начале исходной последовательности. Используемый

Таблица 2.5 Основные операции, включающие последовательность действий.

Операция	**Выход**	**Объяснение**
first(<x_1…x_n>)	x_1	Первый элемент последовательности
last(<x_1…x_n>)	x_n	Последний элемент последовательности
rest(<x_1…x_n>)	<x_2,…,x_n>	Остальная часть последовательности
butlast(<x_1…x_n>)	<x_1,…,x_{n-1}>	Последовательность, исключая последний элемент
select(i,(<x_1…x_n>)	x_i	«*i*»-тый элемент последовательности
cons(a, <x_1…x_n>)	<a, x_1,…,x_n>	Создание новой последовательности путем добавления «a»
insert(i, a, <x_1…x_n>)	<x_1,…x_{i-1},,a,,x_{i+1},x_n>	*a* вставлена как «*i*»-ный элемент следовательности
append(<x_1…x_n>, <y_1…y_m>)	(<x_1…x_n, y_1…y_m>	Сочленение последовательности в порядок
subsequence(<x_1…x_n>,i, *m*)	<x_i,…,x_{i+m-1}>	Подпоследовательность начинается с местонахождения длины *m*
is_subseq(<x_1…x_n>, <y_1…y_m>)	Boolean	Возврат «истина», если <x_1…x_n> входит в <y_1,…y_m> , в противном случае озврат «ложь»

Операция *insert* вставляет заданный элемент в заданном положении и создает новую последовательность. Оператор *cons* - это специфический случай вставки. Оператор *append* принимает две последовательности вида <*x1…xn*> и <*y1…ym*> и порождает новую последовательность вида <*x1…xn, y1…ym*> путем размещения элементов второй последовательности после элементов

первой последовательности. Оператор *subseq* принимает три входных аргумента: (1) исходная последовательность,

(2) стартовая позиция в исходной последоватльности и (3) длина последовательности и извлечение соответствующей последовательности. Например, *subseq*(<*4, 5, 6, 7*>, *3, 2*) получает подпоследовательность <6, 7>. Предикат *is_subseq* принимает в качестве входных две последовательности и проверяет, является ли первая последовательность подпоследовательностью второй последовательности. Например, *is_ subseq*(<*6, 7*>,

<*4, 5, 6, 7*>) возврат *«истина «*.

Последовательность может быть реализована с использованием связного списка, массива или вектора. Все структуры данных имеют разные характеристики. В связном списке доступ к элементам осуществляется последовательно, как правило, от начала до конца; в массиве элементы могут быть доступны хаотически; и вектор может

быть доступен случайным образом и расширен во время выполнения.

2.3.2 Стеки и Списки

И стеки, и списки важны для реализации языков программирования. И стеки, и списки используются в поиске сложного пространства поиска, смоделированного в виде деревьев или графиков. Стеки используются в (1) реализации языков программирования, (2) обработке рекурсии, и (3) управлении и переработке динамических структур данных из хипа, как обсуждалось в Главах 5 и 6. Списки используются в неявной переработке динамических структур данных, выделенных в хипе, как описано в Главе

2.3.2.1 Стек

Стек - это тип абстрактных данных, где данные вводятся только с одного конца. Введенные данные всегда помещаются на текущей вершине стека, а элемент данных удаляется из текущей вершины стека; другой конец блокируется. Кроме того, мы можем прочитать верхний элемент

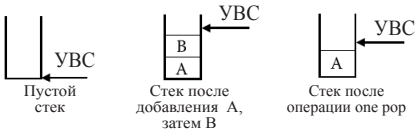


Рисунок 2.6. Операции стека.

стека без удаления элемента из стека, и мы можем проверить пустой стек . Основным преимуществом стека является его способность работать с самым последним элементом данных. Стеки также называются «последним пришёл — первым ушёл» (LIFO, акроним Last In, First Out).

Существует четыре абстрактные операции в стеке: (1) *push* (*стек, элемент данных*) - продвижение элемента данных на вершину стека; (2) *pop (стек)* - чтение и удаление верхнего элемента стека; (3) *top (стек)* - чтение верхнего элемента стека без изменения стека; и (4) *is_empty (стек)* - проверка, пуст ли стек. Инструкции *push (стек, элемент данных)* и *pop (стек)* изменяют стек. В случае операции push, вставляемый элемент становится верхним элементом стека. В случае *pop(стек)*, верхний элемент удаляется из стека. Инструкции *top(стек)* читают верхний элемент стека без модифицирования содержимого стека. Операции на основе стека описаны на рисунке 2.6.

Стек может быть смоделирован как последовательность элементов,

где данные вставляются и изымаются с конца. Отправка элемента *x* в стек S = <$a_1, a_2, ..., a_N$> дает новый стек S’ в качестве

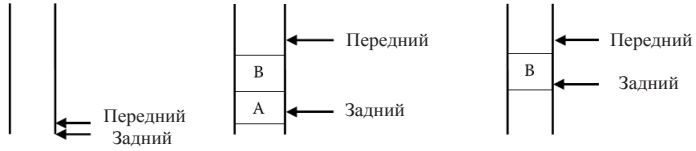
<$x, a_1, a_2, ..., a_N$>, и размер стека увеличиваются на 1. Выталкивание элемента из

стека S = <$a_1, a_2, ..., a_N$> дает новый стек S’ = <$a_2, ..., a_N$>, и размер стека уменьшается на 1. Пустой стек моделируется как пустая последовательность < >.

Реализация стеков в памяти компьютера требует два указателя: стартовый указатель и конечный указатель. Указатель вершины стека пустого стека равен стартовому указателю, и указатель вершины стека полного стека равен конечному указателю. Если указатель вершины стека пытается выйти за пределы конца указателя, то указывается ошибка «переполнение памяти».

Очередь

Очередь - это абстрактный тип данных, где данные извлекаются из одного конца, и данные будут вставлены с другого конца. Как правило, элемент данных извлекается в порядке, в котором он поступил в список. Очередям необходимо два указателя: *передний* и *задний*. *Передний указатель* используется для получения первого элемента данных, а *задний указатель* используется для вставки новых элементов данных. Очередь пуста , когда передний указатель догоняет задний указатель. Для того, чтобы вставить элемент в очередь , данные вставляются в место, указанное задним указателем, а задний указатель увеличивается на единицу. Для того, чтобы извлечь элемент данных с другого конца, элемент данных, который указывает на передний указатель, извлекается, а передний указатель увеличивается на единицу. Очередь является линейныой, если «задний» и «передний» указатели двигаются только в одном направлении, как показано на рисунке 2.7. Очередь делается круговыой с помощью арифметических операций по модулю, где «передний» или «задний» указатели проходят первый элемент списка после прохождения



Пустой список Список после вставки A, затем B Список после операции удаления

Рисунок 2.7 Операции со списками

последнего элемента в списке. Круговые списки оптимально повторно используют пространство памяти, освобожденное после удаления элементов данных.

Гипотетически, очереди могут быть смоделированы как последовательность элементов, где новые элементы данных вставляются в задней части, а первый элемент данных удаляется из переднего конца. Вставка элемента x в очердь $Q = <a_1, a_2, \ldots, a_N>$ дает новый список $Q' <a_1, a_2, \ldots, a_N, x>$, и размер очереди увеличивается на 1. Удаление элемента из очереди $Q = <a_1, a_2, \ldots, a_N>$ дает новую очередь $Q' = <a_2, \ldots, a_N>$, и ее размер уменьшается на 1.

2.3.3 Референтные механизмы

Ячейки памяти и регистры могут хранить два типа информации: (1) значения данных, на которых производятся расчеты и (2) ячейки памяти, указывающие на другие ячейки памяти Указатели - это адреса памяти других ячеек памяти, которые хранятся в регистре или ячейке памяти. Существует множество преимуществ в использовании указателей:

1. 1. Реферирование сложных структур данных, которые хранятся в какой-либо другой части памяти.

2. 2. Избегание накладных расходов на перемещение физически больших структур данных.

3. 3. Отсрочка выделения памяти для переменной до времени выполнения. Это свойство может быть использовано для оптимального распределения памяти для расширяемых структур данных, таких как связные списки, деревья, векторы, и так далее.

4. 4. Выделение блоков памяти для сложной логически смежной структуры данных, для которых несколько блоков физической памяти распределяются в разное время и привязанного через указатели.

5. 5. Использование в качестве ссылки переадресации, обеспечивая независимость программы от перемещения данных в физической памяти.

6. 6. Совместное использование части сложной структуры данных с другими структурами данных.

Указатели были использованы при реализации рекурсивных структур данных, таких как связные списки и деревья, так как точный размер

памяти, необходимый для рекурсивных структур данных неизвестен во время компиляции. Более того, в зависимости от входных данных, размер рекурсивной структуры данных может изменяться и не может быть оценен во время компиляции.

У системного программирования есть потребность получать доступ шаг за шагом по ячейкам памяти. Это необходимо , так как указатели должны быть обеспечены возможностью сложения и вычитания так, чтобы системные программы могли пройти через область памяти, которая моделируется как одномерный массив. Это нужно для того , чтобы определенные пользователем указатели не пересекали границу сегмента- области в оперативной памяти, выделенной для выполнения конкретной программы.

Несмотря на преимущества указателей, есть три основных недостатка при использовании указателей на языках высокого уровня:

1. 1. Арифметические операции над указателями позволяют перестраиваться памяти скачкообразно. Попытка арифметической операции получить доступ к ячейке памяти вне сегмента вызывает ошибку «нарушение сегмента».

2. 2. После компиляции программы, различные переменные, имеющие различные типы данных, преобразуются в единое пространство линейной памяти; вся информация о границах между этими типами данных теряется. Если мы разрешим арифметику в указателях , то указатели могут неправильно пройти через различные типы данных объектов во время выполнения чтения или изменения значений.

3. 3. Ячейки памяти, используемые несколькими структурами данных, не могут быть использованы повторно, пока все указатели, указывающие на ячейку памяти, не будут освобождены.

Из-за этих проблем с указателями, программы могут быть подвержены ошибкам, и может произойти сбой во время выполнения. Трудно вычислить такие ошибки, связанные с указателями.

2.3.4 Рекурсивная структура данных

Существуют классы структур данных, таких как связные списки, деревья и векторы, которые могут быть определены рекурсивно. Связанный список определяется рекурсивно как узел информации с

последующей остальной частью списка, а базовым случаем является пустой список.

<linked-list> ::= <information-node> <linked-list>
<linked-list> ::= null

Бинарное дерево определяется рекурсивно как информационный узел, за которым следует левое поддерево и правое поддерево. Базовый случай представляет собой пустое дерево.

<binary-tree> ::= <binary-tree><information-node><binary-tree>
<binary-tree> ::= null

Указатели на основе структуры данных имеют эффективные операции вставки и удаления. Однако, рекурсивные структуры данных не могут быть выделены во время компиляции из-за отсутствия знаний о размере времени выполнения рекурсивных структур данных. Рекурсивные структуры данных растут во время выполнения, в зависимости от выполнения программы и данных ввода. Память для рекурсивных структур данных выделяется по частям, основываясь на требовании программиста во время выполнения. Нам нужна специальная область общей памяти под названием хип для реализации рекурсивных структур данных. Этот хип отличен от того, который мы изучили в структурах данных. Рекурсивные структуры данных физически разбросаны в хипе, основываясь на наличии слотов памяти и порядке, которого требует программа для выделения памяти во время выполнения. Различные физические фрагменты той же рекурсивной структуры данных соединены через цепочку указателей.

2.3.5 Деревья

Дерево является рекурсивной структурой данных, которая состоит из нескольких информационных узлов, организованных в виде общего информационного узла, который содержит один или несколько элементов данных и имеет возможные поддеревья, укорененные отношениями с узлами-потомками информационного узла. Базовый случай дерева называется «нулевым» деревом. Дерево может иметь любое количество ветвей. Бинарное дерево имеет одну ячейку для каждого информационного узла и не более двух поддеревьев: левое

поддерево и правое поддерево. Дерево зовется «n-арным деревом», если максимальным количеством ветвей, исходящих от узла, является «N». Деревья могут быть реализованы с использованием двух методов: (1) индексируемые структуры данных, такие как массивы или векторы, и (2) указатели, соединяющие родительские узлы с дочерними узлами. Индексируемые структуры данных подходят для представления полных бинарных деревьев или почти полных бинарных деревьев. Почти полное бинарное дерево имеет все не листовые узлы с двумя потомками, за исключением крайнего правого, не листового узла, который может иметь одного левого потомка, как показано на рисунке 2.8.

Реализация на основе указателей подходит для любого дерева. Тем не менее, они не имеют возможности вычислить индекс родительского узла, чтобы пройти обратно. Деревья, реализованные с использованием указателей, являются направленными в природе: деревья могут перемещаться только от родительского узла к дочернему узлу. Чтобы обойти это направленное свойство указателей, нам необходим либо стек для хранения адреса родителей до перехода к потомкам, либо использование обратного указателя, сохраненного в дочернем узле в дополнение к указателям от родительского узла к дочернему узлу. Обе схемы имеют накладные расходы памяти, а также накладные расходы времени выполнения. Тем не менее, в случае стека, пустая трата памяти ограничена только глубиной дерева.

Деревья используются в языках программирования и компиляторах во время фазы синтаксического анализа. Специальный вид дерева, называющийся «дерево типа И - ИЛИ» является основой реализации парадигмы логического программирования и был подробно рассмотрен в главе 10. Деревья также могут быть использованы, чтобы показать структуру вложенности процедур в программе или блоков в процедуре и, таким образом, могут помочь в понимании степени объема переменной, объявленной в процедуре или в блоке. N-арное дерево также используется для реализации структур данных в парадигме логического программирования, как описано в главе 10.

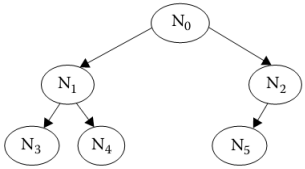


Рисунок 2.8 Почти полное бинарное дерево.

2.3.6 Графы

Граф формально определяется как пара (V, E), где V есть множество вершин, а E является мультимножеством ребер. В неориентированном графе ребра смоделированы как пара (v_i, v_j), где v_i, v_j $\in$ V являются вершинами. Пара (v_i, v_j) $\in$ E показывает связь между вершинами v_i и v_j. Может существовать более одного ребра между двумя узлами. *Путь* - это последовательность ребер, соединяющих *исходный узел* с *узлом назначения*. Может существовать более одного пути между двумя узлами.

В *неориентированном графе* наличие ребра (v_i, v_j) подразумевает наличие симметричного ребра (v_j, v_i). В *ориентированном графе*- графе, где ребра имеют направления, ребра моделируется как упорядоченная пара (v_i, v_j), и наличие (v_i, v_j) не означает наличие ребра (v_j, v_i). Взвешенный граф имеет весовые коэффициенты, связанные с ребрами. Ребра в взвешенном графе моделируются как тройка вида (v_i, v_j, $вес_{ij}$), где вес ij дает вес ребра. Во в*звешенном неориентированном графе* наличие взвешенного ребра (v_i, v_j, *вес ij*) подразумевает наличие *(*v_j, v_i,$вес_{ij}$); веса симметричных ребер остаются прежними. Во *взвешенном ориентированном графе* наличие взвешенного ребра формы (v_i, v_j, $вес_{ij}$) не подразумевает наличие симметричного ребра в другом направлении с тем же

весом. Связный граф - это транзитивное отношение. Если узел v_i связан с v_j , а узел v_j связан с v_k, тогда узел v_i связан с v_k, хотя явное ребро (v_i, v_k) может не существовать. Это свойство имеет важное значение с точки зрения достижимости элементов данных в рекурсивных структурах данных.

Пример 2.12

Рисунок 2.9 иллюстрирует модель связности дорог и расстояний между набором городов в Соединенных Штатах, используя неориентированный взвешенный граф. Города также могут быть соединены посредством железнодорожного сообщения или авиасообщения, чтобы получить несколько ребер между двумя узлами. Тем не менее, мы рассматриваем только дорожные ссылки на этом графе. Граф показывает, что существует шесть городов, представленных в виде узлов (или вершин) графа, а связующие между ними дороги показаны с использованием взвешенных ребер между вершинами, где вес показывает расстояние пути в милях между городами.

Граф *ациклич*- он не имеет цикла; если начать с узла, никто не сможет достичь того же узла с использованием двух различных

последовательностей ребер. Дерево - ацикличный граф. Граф *цикличен*, если начав с, по крайней мере, одного узла, мы можем снова достичь того же узла, используя путь, в котором ребра не повторяются. Например, существует множество циклов в связном графе, данном на Рисунке 2.9, таких как<(*Сиэтл, Кент*), (*Кент, Вашингтон округ Колумбия*),

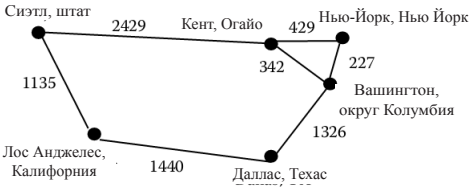


Рисунок 2.9 Пример взвешенного неориентированного графа.

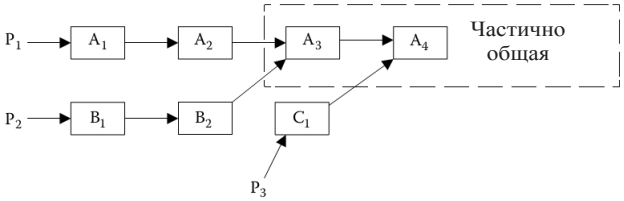


Рисунок 2.10 Моделирование ориентированного ациклического графа с общими информационными узлами.

(*Вашингтон округ Колумбия, Даллас*), (*Даллас, Лос-Анджелес*), (*Лос-Анджелес, Сиэтл*)>. Другой такой цикл это <(*Кент, Нью-Йорк*), (*Нью-Йорк, Вашингтон, округ Колумбия*), (*Вашингтон, округ Колумбия, Кент*)>.

Определение циклов в графах является серьезной проблемой. Для того, чтобы определить циклы, необходимо сохранить ранее посещенные узлы, поэтому проверка принадлежности может быть использована для проверки, пересматривался ли ранее посещенный узел. Проверка принадлежности может быть выполнена неявно, используя хеширование, как описано в Разделе 2.3.9. В языках программирования, графы используются для представления общих структур данных, как описано в примере 2.13. Графы также используются для (1) модели информационного потока между операторами программы; (2), выяснения, какие ячейки данных соединены в хипе во время утилизации памяти; (3) моделирования пространства решений задачи и (4) реализации функциональных языков программирования.

Пример 2.13

На рисунке 2.10 показаны три связных списка, обменивающихся частями своих структурных данных: первый связный список состоит из четырех информационных узлов $A1$, $A2$, $A3$, $A4$; второй связный список состоит из информационных узлов $B1$, $B2$, $A3$, $A4$; третий связный список состоит из информационных узлов $C1$, $A4$. Первая и вторая структуры данных разделяют два информационных узла: $A3$ и $A4$; первая, вторая, и третья структуры данных разделяют общий информационный узел $A4$.

2.3.7 Метод перебора

Вычисление может быть смоделировано как проблема поиска в пространстве состояний. Проблема в пространстве состояний имеет несколько вычислительных состояний, и каждая вычислительная инструкция переносит машину от одного вычислительного состояния в другое вычислительное состояние. Расчет может быть смоделирован как граф в пространстве состояний. Различные методы поиска могут быть классифицированы по двум основным категориям : (1) метода перебора и (2) эвристический поиск. Схемы *метода перебора* потенциально ищут все узлы в пространстве состояний графа до узла, пока цель не будет найдена. Целевой узел - это состояние, которое имеет желаемое последнее условие. Метод перебора гарантирует решение, если поиск не вошел в цикл. Существуют два основных подхода к методу перебора в деревьях или графах: *поиск в глубину* и *поиск в ширину*.

Эвристический поиск использует математические функции, чтобы оценить расстояние и близость текущего состояния к конечному состоянию и целенаправленно двигаться в сторону конечного состояния. Эвристический поиск является более эффективным, чем метод перебора, но не гарантирует решения. Большинство задач языков программирования используют метод перебора, в то время как задачи искусственного интеллекта используют эвристический поиск.

Поиск в глубину

Поиск в глубину обходит дерево по узлам в заданном порядке, обычно в порядке слева направо. Такой поиск использует стек для хранения информации для облегчения перемещения к правым поддеревьям. Поиск начинается с корневого узла и выбирает крайнего левого потомка для обхода. Когда исследуется поддерево с корнем в узле неисследованного

потомка, для указателя нет никакой возможности вернуться к правым потомкам, если указатель снова не посещает текущий узел. Для облегчения обхода к нужным потомкам, адрес текущего узла хранится в стеке. В отсутствие каких-либо неизученных потомков, нет необходимости хранить адреса текущего узла в стеке. Так как стек хранит информацию в режиме LIFO, родитель текущего узла при обходе извлекается в первую очередь. После обхода листового узла (узла, не имеющего потомка), следующий узел извлекается из верхней части стека. Этот процесс извлечения из стека и исследования следующего правого поддерева для извлеченного узла повторяется до тех пор, пока стек не становится пуст, и больше нет поддерева для обхода. Так как более не существует поддерева для обхода, поиск прекращается.

Пример 2.14

Рассмотрим пример на рисунке 2.11. Указатель сначала обходит узел P_0. В это время стек пуст. Однако узел P_0 имеет и левого, и правого потомка. Затем узел P_1-левый дочерний узла P_0-обходится, а адрес узла P_0 хранится в стеке, чтобы облегчить обход узла P_2 позже. Следующий узел P_3-левый потомок узла P_1-обходится, а адрес узла P_1 хранится в стеке, и стек выглядит как <адрес (P_1), адрес (P_0)>. Поскольку узел P_3 - листовой узел , узел P_1 извлекается из стека после обхода узла P_3, и правый потомок - узел P_4-обходится. После извлечения адреса из узла P_1, стек выглядит следующим образом:

<адрес(P_0)>.

Поскольку узел P_1 не имеет больше потомков, никакая информация не помещается в стек во время обхода узла P_4. После обхода узла P_4, адрес узла P_0 извлекается из стека и стек становится пустым. Тем не менее, правый потомок узла P_0 остается для обхода. Стек остается пустым, так как узел P_2 не имеет правого одноуровневого элемента. После обхода узла P_2, адрес узла P_2 хранится в стеке, и узел P_5 обходится. На этом этапе стек выглядит как <адрес P_2)>.

После обхода узла P_5, стек извлекается. Узел P_6, , правый потомок узла P_2, обходится и стек становится пустым. После обхода узла P_6, стек становится пустым. Поскольку ни один дополнительный узел не нуждается в обходе, поиск прекращается.

Основным преимуществом поиска в глубину является то, что размер стека ограничен глубиной дерева, и только на сфокусированной части дерева производится поиск в это время. Тем не менее, примитивный поиск в глубину может застрять на неопределенное время в присутствии

циклов. Обнаружение цикла требует значительного времени выполнения и накладных расходов памяти для хранения всех посещенных узлов и проверки принадлежности узла среди посещаемых узлов. Хэш-таблица может быть использована, чтобы эффективно идентифицировать циклы. Тем не менее, для больших структур данных, накладные расходы памяти для хранения обойденных узлов велики.

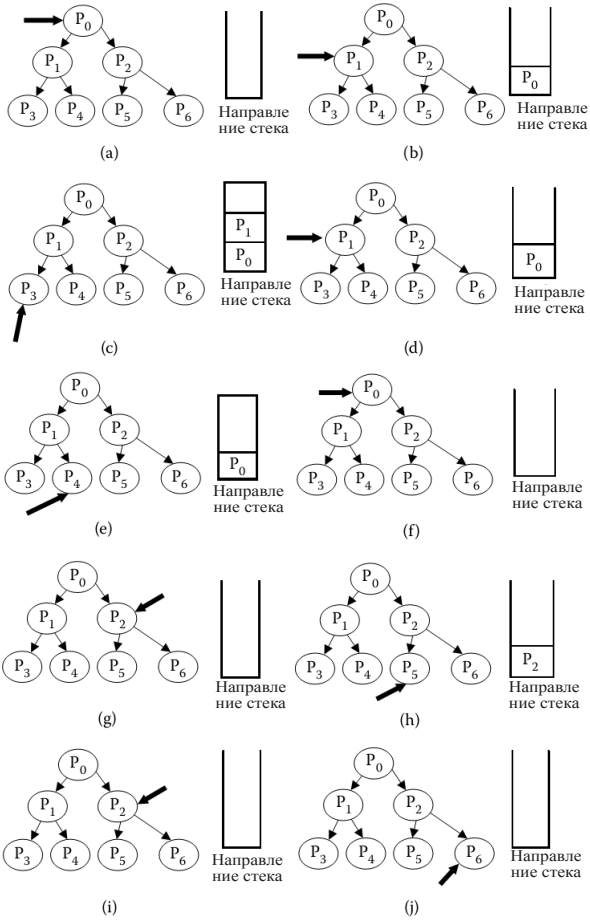


Рисунок 2.11 Пример посика в глубину.

Поиск в ширину

Поиск в ширину (см. Рисунок 2.12) обходит дерево уровень за уровнем в порядке слева направо и использует списки для хранения узлов потомков для посещения. Корневой узел (исходный узел) помещается в очередь на начальном этапе. После этого, каждый раз, когда один элемент удаляется из очереди, все потомки удаленного узла вставляются в заднюю часть очереди. Процесс повторяется до тех пор, пока очередь не опустеет.

Пример 2.15

Рисунок 2.12 иллюстрирует поиск в ширину. Указатель сначала посещает корневой узел P_0 и вставляет адреса потомков P_1 и P_2 в очередь. Затем удаляется

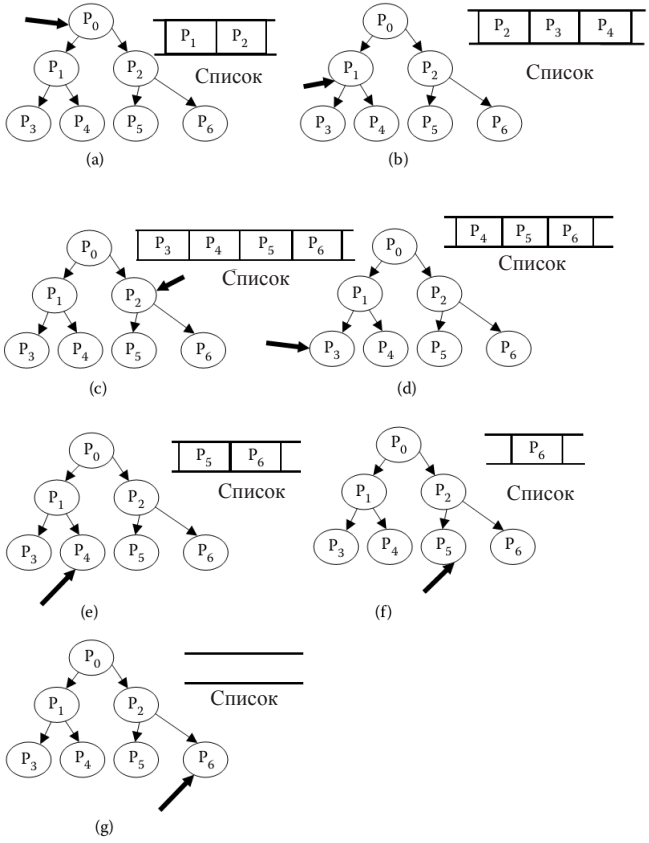


Рисунок 2.12 Пример поиска в ширину.

адрес узла P_1 из очереди и вставляет адреса потомков P_3 и P_4 в очередь, и посещает узел P_1. Затем он удаляет узел P_2 из очереди и вставляет адреса потомков P_5 и P_6 в очередь. Так как узлы P_3, P_4, P_5, и P_6 перечислены в виде листьев, во время их обхода в очередь ничего не вставляется. После того, как последний лист P_6 удален, все узлы обойдены, очередь становится пустой.

Преимуществом поиска в ширину является то, что он обходит дерево на один уровень за один раз.

Существует и множество недостатков поиска в ширину. Накладные памяти при поиске в ширину достаточно велики. Во время обхода дерева, он захватывает столько узлов, сколько существует не терминальных узлов в прежнем уровне. Это число может быть таким большим, как половина числа листов в случае сбалансированного дерева. Однако, если накладным расходом памяти

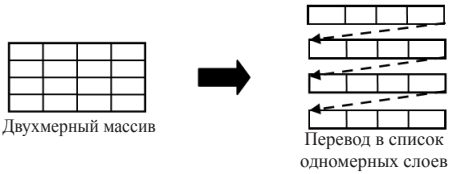


Рисунок 2.13 Перевод двухмерной матрицы в одномерную память.

списка можно эффективно управлять, поиск в ширину можно использовать достаточно эффективно для обхода графика и использовать в некоторых эффективных схемах утилизации памяти в хипе.

2.3.8 Отображение структуры данных в одномерной памяти

Программа может использовать *структуру* (кортеж с множеством полей) для моделирования сложных объектов данных или *многомерных массивов* для моделирования коллекции объектов данных. Массивы могут быть как *статическими*, так и *динамическими. Статические массивы* - это выделенная память - разово во время процедуры вызова - и затем более выделение памяти не требуется. Размер *динамического массива* меняется во время выполнения, основываясь на выполнении программы. В дополнение к массивам и структурам существуют динамические объекты в объектно-ориентированных языках, таких как C++, Java или C # и других современных языках программирования. Все

эти структуры данных должны быть отображены в ОЗУ. ОЗУ может быть абстрагировано как одномерный массив, индексированный по адресам памяти.

Для отображения многомерных массивов на ОЗУ, многомерные массивы должны быть переведены в одномерные массивы, а адреса в ОЗУ должны быть вычислены с помощью уравнения, которое транслирует индекс дата-элемента в относительный адрес, который может быть добавлен к базовому адресу (начальному адресу) структуры данных. Для понимания процесса перевода двухмерных массивов, мы по отдельности возьмем каждую строку (или столбец) одну за одной и поместим их одну за другой в полосе в одном измерении, как показано на рисунке 2.13.

Задача узнать адрес элемента a(*i, j*) для двумерного массива размера *M* × *N* дана в Уравнении 2.1, с условием, что индекс массива начинается с 0. Первое слагаемое в правой части является базовым адресом - адресом [0, 0] - и второе слагаемое производит смещение. Первый элемент второго слагаемого утверждает, что *i*-строки происходят до текущей строки, а второй элемент второго слагаемого показывает смещение элемента в текущей строке.

Адрес = Адрес(a[0, 0]) + (i ∗ N + j) ∗ количество байтов в одном элементе (2.1)

Пример 2.16

Представим двухмерную матрицу целых чисел *m*, имеющую размер 5×6 (5 строк и 6 столбцов). Мы хотим найти адрес элемента, расположенного в области m[2, 4] - строка №2, столбец №4. Предположим также, что индекс начинается с 0, а базовый адрес m [0, 0] - 1000, и 32-битному целому числу необходимы четыре байта. Используя Уравнение 2.1, местонахождением m[2, 4] является 1000 + (2 × 6 + 4) * 4 = 1064. Целое число, хранящееся в элементе m[2, 4] будет в четыре байта, начиная с адреса 1064.

Эта концепция может быть обобщена на карту элементов любой произвольной двухмерной матрицы путем постепенного представления многомерной матрицы в виде последовательности меньшей двухмерной матрицы, пока она не будет сведена к одномерной матрице. Дана *N*-мерная матрица *m*, имеющая значения измерений вида D_1, D_2, ..., D_N, адрес элемента, расположенного в m[i_1, ..., i_N]

$$\text{Address} = \text{Address}(m[0,...,0]) + ((i_1 \cdot D_2 \cdot ... \cdot D_N) + ... + (i_{N-1} \cdot D_N) + i_N) \cdot \text{количество байтов в одном элементе} \quad (2.2)$$

Для получения композиционной структуры, такой как «структура» с несколькими полями вида (поле $_1$, поле $_2$, ..., поле $_N$), смещения каждого поля вычисляются во время компиляции. Это смещение добавляется к базовому адресу для доступа к «*i*»-тому полю

2.3.9 Хэш-таблицы

Важно эффективно извлекать данные во время компиляции и выполнения программы. В то время как статические массивы имеют постоянную временнУю деятельность для извлечения элементов данных, они не подходят в случае, когда даются входные данные для поиска, элемент данных не является индексом, но ключом к записи. Хэш-таблицы поддерживают поиск на основе ключей в почти постоянном времени, используя хэш-функцию *f*, которая отображает ключ *<ключ>* к значению индекса: *f*(*<ключ>*) равен индексу хранения, в котором сохраняется элемент данных . Хэш-таблицы используются в модели реализации языков программирования из-за эффективности поиска.

Главным ограничением при поиске на основе хэширования является *процесс столкновения ключей*-отображения двух ключей на одном и том же значении индекса. Некоторыми из популярных методов для обработки индекса столкновений являются:

(1) (1) выбор размера таблицы в виде простого числа, и использование этого простого числа в хэш-функции для получения индекса. Разделение на простое число в хэш-функции, равномерно распределяет ключи в хэш-таблице, уменьшая вероятность столкновений; (2), использование связного списка, прикрепленного к определенному индексу для обработки нескольких сталкивающихся ключей ; и (3) расширение хэш-таблицы и рехэширование всех элементов, в случае столкновения или скученности элементов в хэш-таблице увеличивается выше порогового значения.

Пример 2.17

Рисунок 2.14 показывает пример хэш-таблицы размером 11, используемый для хранения набора из 6 переменных и их атрибутов. Существует три процедуры: *p0, p1*, и *p2*. Процедура *p0* имеет переменную *x*; Процедура *p1* имеет переменные *s*, *i*, и *j*; Процедура *p2* имеет переменные *a* и *i*. Для того, чтобы избежать конфликта имен, имя процедуры прилагается в качестве префикса к переменным. Например, ключом для переменной *x* в *p0*является *p0x*; ключами для переменных *s*, *i*, и *j* в *p1* являются *p1s, p1i,* и *p1j* соответственно; а ключами для переменных *a* и *i* в *p2* являются *p2a* и *p2i* соответственно. Теперь все

ключи уникальны даже для переменных с конфликтующими именами. Каждый элемент данных - это тройка вида (*идентификатор*, *тип, ячейка памяти*). Шесть троек это: {(*p0x, int, L1*), (*p1s, Bool, L2*), (*p1i, int, L3*), (*p1j, int, L4*), (*p2a, int, L5*), и (*p2i, int, L6*)}.

Функция для отображения ключа нужна для (1) суммирования порядковых значений символов от '*a*' до '*z*' слева направо в ключе; (2) добавления значений цифр в ключе; и (3) для деления суммы на размер таблицы 11, которое является простым числом. Связные списки используются для разрешения столкновений

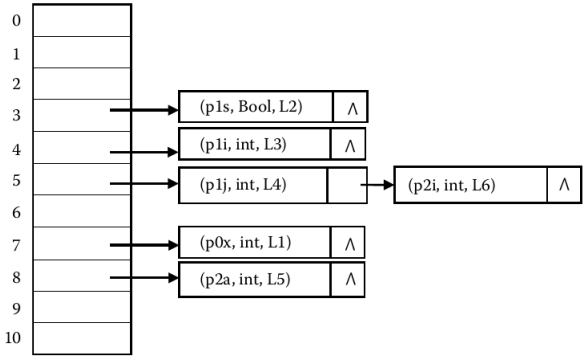


Рисунок 2.14 Пример хэш-таблицы со связным списком для решения конфликтов.

ключей. Например, имя p0x имеет три символа, начиная с символа 'p', имеющего порядковое значение 16, цифра '0', имеющее значение 0, и символ 'x', имеющий порядковое значение 24. Суммируя эти значения, получается значение 40, а разделяя по размеру таблицы 11 дает значение индекса 7. Ключ p1s отображает значение индекса 3. Ключ p1i отображает значение индекса 4. Ключ p1j отображает значение индекса 5. Ключ p2a отображает значение индекса 8, а ключ p2i отображает значение индекса 5. Столкновение ключей p1j и p2i решается использованием связных списков.

2.4 АБСТРАКТНЫЕ ПОНЯТИЯ В ВЫЧИСЛЕНИЯХ

В этом разделе описываются некоторые справочные понятия, используемые в разработке языков программирования. Большая часть этих понятий были вскользь упомянуты в различных контекстах в предыдущем курсе основ и программирования, таком как структуры данных и дискретные структуры. Эти основополагающие понятия

неоднократно использовались на протяжении всего этого курса и нуждаются в более формальном представлении, нежели в разработке языка программ и перспективе реализации.

Программа - это последовательность значимых инструкций. Каждая такая инструкция также называется предложением или выражением. Каждое выражение завершается разделителем на многих языках, либо переводом строк в некоторых языках. Зарезервированное слово является частью языка программирования, похожей на слова на естественных языках. Два зарезервированные слова или две переменные, или любые два объекта, которые имеют самостоятельное значение, отделены друг от друга пустотой, запятой, точкой с запятой, двоеточием, подачей на одну строку, или зарезервированными словами. Эти сепараторы называются разделителями.

В программах могут быть литералы, r-значения, l-значения, идентификаторы, определения, переменные, операторы присваивания, инструкции, команды, выражения, строки символов, объявления типов, операторы, процедуры вызовов, параметры, метки и секвенсоры. Литерал - это элементарная запись, которая не может быть дополнительно разбита на более мелкие части, и которая не может быть пересмотрена. Литералы также известны как нечисленные константы, константы и дескрипторы. Например, число является литералом, символ является литералом, имя ‘Арвинд’ в одинарных кавычках является литералом. Однако, строка не является литералом: строка представляет собой последовательность символов; специальные значения истина и ложь не являются литералами, так как они могут быть пересмотрены. R-значение - это значение, которое разрешается употреблять в правой части инструкции присваивания. L-значение - это область памяти, связанная с переменной или элементом массива или полем внутри структуры. Идентификатор представляет собой символическое имя, используемое в программе, которое может быть связано с другим объектом. Например, имена программы, имена функции, имена переменных, имена констант и пользовательские типы - это идентификаторы. Определение обозначает соответствующее значение в пределах блока или процедуры. Определение заменяется соответствующим значением во время компиляции.

Переменная связана с информационным блоком. Информационный блок может быть простым значением в конкретной области, сложным объектом или значением в абстрактной области, такой как область типа. Конкретное значение представляет собой фактическое значение, с которым программа работает. Переменная типа содержит информацию о

типе. Например, переменная типа может быть связана с целочисленной переменной абстрактного домена. Понятие переменных типа широко используется в полиморфных языках и объясняется в главе 7. С этого момента и далее, мы имеем в виду переменную в качестве носителя значения конкретных значений и переменных типа в качестве носителя значения информации типа. Внутренне переменная отображается в ячейке памяти во время компиляции. Соответствующая ячейка памяти может нести либо конкретное значение, либо указатель на объект в хипе.

Оператор присваивания, такой как x = y + 4 это утверждение программиста, который считывает сохраненное значение из ячейки памяти переменных в выражении на правой стороне утверждения, оценивает алгебраическое выражение, и записывает полученное значение в ячейку памяти переменной на левой стороне утверждения. В приведенном выше операторе, значение переменной y считывается из хранилища, выражение value-of(y) + 4 вычисляется, и результат вычисления выражения сохраняется в ячейке памяти, соответствующий идентификатору x.

Команда - это абстракция управления, которая имеет по крайней мере один оператор присваивания, встроенный в неё. Использование оператора присваивания означает, что новое значение записывается по меньшей мере в одну ячейку памяти в команде. В отличие от команды, представление не включает в себя задание; представление просто считывает значения из ячеек памяти и оценивает их.

Строка представляет собой последовательность символов. Пустая строка не содержит ни одного символа в ней. Объявление типа используется для создания определенной пользователем абстракции данных таким образом, что данные могут быть сохранены в указанном пользователем способом, чтобы смоделировать объекты в реальных проблемах. Тем не менее, объявление типа само по себе не создает ячейку памяти. Ячейки памяти создаются только тогда, когда переменная объявлена.

Оператор может быть арифметическим или логическим. Оператор может быть унарным (одноместным) или бинарным (двуместным). Унарные операторы имеют один операнд, а бинарные операторы - два операнда. Арифметические операторы могут быть ‘<’ (меньше чем), ‘>’ (больше чем), ‘= <’ (меньше или равно), ‘> =’ (больше или равно), ‘==’ (равно), ‘<>’ (не равно), ‘= / =’ (не равно как в Прологе). Кроме того, существуют унарные операторы, такие как «+» и «-», чтобы обеспечить знак числа . Бинарный логический оператор, такой как логическое И (∧),

логическое ИЛИ (∨), исключающее ИЛИ (⊕), и импликация соединяют два Булевых выражения с использованием значений истинности, как было описано ранее, а унарный оператор ‘не’ принимает одно Булево выражение в качестве аргумента и возвращает утверждение истина или ложь.

И арифметический и Булев операторы имеют *приоритет оператора*: получив выражение , некоторые операторы упрощаются первыми. Унарные операторы имеют более высокий приоритет, чем бинарные операторы; в бинарных операторах умножение и деление имеют более высокий приоритет, чем сложение и вычитание. Точно так же, Булев унарный оператор «не» (¬) имеет наивысший приоритет, за которым следует логическое И (∧), а затем логического ИЛИ (∨), и затем следствие (→).

2.4.1 Изменяемые переменные против неизменяемых переменных

В парадигме императивного программирования переменные являются изменяемыми: пользователь может деструктивно обновить содержимое ячейки памяти, соответствующей переменной, по желанию, используя операции присваивания, например, x = y + 4. В отличие от парадигмы императивного программирования, парадигма декларативного программирования не допускает изменение ячеек памяти на пользовательском уровне; переменная является просто неизменяемым значением, которое может быть связано со значением только один раз.

Существуют преимущества и недостатки обоих подходов обработки переменных. Основным преимуществом деструктивного обновления ячейки памяти является повторное использование памяти, например, в повторном использовании высказываний, включающем индексные переменные цикла FOR. Основными недостатками деструктивного обновления переменной являются (1) потери прошлых значений и (2) побочные эффекты в результате нежелательного поведения программы, вызванные нарушением фундаментальных математических свойств программ, как описано в главе 4. Если прошлые значения не могут быть восстановлены, то любой метод решения проблем не сможет выполнить возврат (возвращение), чтобы опробовать альтернативные возможности получения решение, даже если решение может существовать.

Основные преимущества неизменяемых переменных с назначенным один раз свойством заключаются в: (1) сокращении возможности нежелательного поведения программ, вызванного побочными

эффектами и (2) использование последнего значения для использования альтернативных решений, таких как логическое программирование, которое рассматривается в главе 9. Основным недостатком является то, что память не может быть повторно использована в результате взрыва памяти при обработке элементов данных с участием больших структур данных, особенно итеративным способом. Многие современные языки, поддерживающие несколько парадигм, поддерживают как изменяемые, так и неизменяемые определения переменных, чтобы максимизировать преимущества в обоих подходах.

2.4.2 Связывание и правила области видимости

Связывание ассоциирует два элемента или элемент с соответствующими атрибутами. Например, имя переменной может быть привязано к ячейке памяти, ячейка памяти может быть связана со значением, идентификатор может быть связан со значением, а переменная может быть привязана к типу. Аналогичным образом, группа операторов может быть связана с идентификатором и называться функцией или процедурой. Правило области видимости определяет использование переменной и степень ее видимости. Переменная видна и может быть использована в пределах своей области; она не играет никакой роли за ее областью. Связывание может быть временным или стойким, в зависимости от области.

Существует два типа правил области видимости: статические и динамические. Как следует из названия, статическая область видимости основывается на структуре программы, такой как границы блока, границы подпрограммы или функции (или метода, в объектно-ориентированных языках программирования) и границы класса; она не меняется с вызывающим шаблоном подпрограмм. Динамическая область видимости изменяет масштаб переменной, основанной на вызывающих шаблонах подпрограмм. Пример статической области видимости приведен в примере 2.18. Пример динамической области видимости приведен в примере 2.19.

```
integer x, y, z;
main ( )
{x = 4; y = 10; z = 12;
  {integer temp, z;
       temp = x; x = y; y = temp; z = 5;}
print(x, y, z);
}
```

Рисунок 2.15 Пример статической области видимости переменных.

Пример 2.18

Программа на рисунке 2.15 имеет три глобальные переменные x, y, и z во внешнем блоке, а также две локальные переменные temp и z во внутреннем блоке. Глобальные переменные видимы для всей программе, в то время как область видимости локальных переменных ограничена блоком, в котором они были объявлены.

Конфликт имен переменной z решается путем отдачи предпочтения переменной, которая ближе всего к использованию. Переменная z в операторе печати относится к глобальной переменной z, в то время как переменная z во внутреннем блоке относится к локальной переменной z. Обе эти переменные являются отдельными объектами данных, отображаются в разных местах памяти и имеют различные сроки службы. Когда переменная z печатается вне области внутреннего блока, значение глобальной переменной z = 12 печатается. Информация об области видимости этих переменных может быть определена при просмотре структуры программы, и она не имеет ничего общего с вызывающим шаблоном программы.

Пример 2.19

Программа на рисунке 2.16 использует правило динамической области. Программа имеет функцию sum, которая имеет две переменные: переменная x объявлена в пределах функции sum, а переменная y - это свободное вхождение; то есть, переменная y не была объявлена в функции sum. Переменная x забирает значение из механизма передачи параметров. Тем не менее, переменная y ищет такое же имя в области памяти, выделенной для цепочки процедур вызова, чтобы выяснить соответствующее значение связывания.

Основная программа состоит из двух блоков. Первый блок имеет две локальные переменные y и z, и вызывает функцию sum с аргументом, используя переменную y. Второй блок состоит из трех локальных переменных w, y, и z, и вызывает функцию sum с фактическим параметром z.

```
integer sum(integer x);
return (x + y);
main ( )
{ {integer y, z; y = 4; z = 5; sum(y);}
  {integer w, y, z; w = 4, y = 5; z = 6; sum(z);}
}
```

Рисунок 2.16 Пример динамической области видимости переменных.

Первый вызов функции *sum(y)* будет возвращать значение 8, так как формальный параметр *x* получает значение фактического параметра *y*, а свободная переменная *y* в функции *sum* динамически связана с переменной *y*, объявленной в первом блоке основной программы. Второй вызов функции *sum* возвращает значение 11, так как формальный параметр *x* в функции *sum* получает значение фактического параметра *z* в основной программе, а свободная переменная *y* в функции *sum* получает значение 5, связанной с переменной *y*, объявленной во втором блоке *основной* программы. Это связывание свободной переменной и ее привязывание к последнему объявлению в цепи вызывающего шаблона из подпрограмм, что делает область видимости переменных динамичной.

2.4.3 Типы переменных

Переменные были классифицированы в зависимости от их видимости, на основе правил видимости и срока жизни. *Переменная*, которая имеет срок службы программы называется *глобальной переменной*; переменная, которая имеет срок службы и область действия процедуры называется *локальной переменной*; а переменная, которая может быть объявлена во внешнем уровне вложенности вложенной процедурной структуры, но используется в одной из вложенных процедур называется *нелокальной переменной.* В блочно-структурированных языках - языках, поддерживающих требования программы, организованной в виде блоков операторов -переменные также могут быть локальными по отношению к блоку. Область действия переменных ограничена в пределах блока, в котором они были объявлены.

Существует возможность конфликта имен между нелокальными и локальными переменными или глобальными и локальными переменными. Конфликт разрешается путем *затенения* (становления невидимым) нелокальных или глобальных переменных, если локальная переменная объявлена с тем же именем. Концепция затенения также относится к обработке видимости нелокальных переменных, если они были объявлены на разных уровнях вложенности. В таких случаях переменная, которая объявлена на уровне вложенности, ближайшем текущей процедуре, использующей то же имя переменной, видна, а другие переменные с одинаковыми именами затеняются.

Переменные могут быть *статическими* или *динамическими. Статические переменные* выделяются в фиксированной памяти во время компиляции, и эта ячейка памяти не изменяется во время выполнения программы. Преимущество статического распределения

памяти, как описано в главе 5, это более быстрый прямой доступ к памяти без использования какого-либо указателя. *Статическая переменная* может быть *статической глобальной переменной*, когда областью видимости статической переменной является вся программа, или она может быть *статической локальной переменной*, когда область видимости программы ограничена в рамках процедуры, в которой она была объявлена. Тем не менее, из-за локальной области видимости переменной, значение недоступно вне рамок процедуры и может быть доступно только в течение последующих вызовов процедуры, в которой она объявлена. *Динамические переменные* -переменные, место под которые выделяется во время выполнения программы и помещаются в стек управления. Большинство объявлений локальных переменных являются динамическими локальными переменными и размещаются в стеке в реализации языков на его основе, как описано в главе 5.

Глобальные переменные также распределены как статические переменные для эффективного доступа к памяти. Все остальные переменные имеют ограниченную видимость, что зависит как от вложенных массивов программных блоков, так и от блоков. Локальные переменные видны только в пределах блока, в котором они объявлены. Локальные переменные на уровне процедуры имеют видимость в пределах процедуры.

В объектно-ориентированных языках переменные могут быть объявлены в классе, и они распределяются между всеми экземплярами класса. Такие переменные называются *переменными класса*. В отличие от переменных класса, мы можем иметь *переменные экземпляра* , чьи возможности ограничены для созданного объекта.

Другая классификация переменных основана на *изменяемости*, как обсуждалось ранее. *Изменяемая переменная* может быть деструктивно обновлена любое количество раз, в то время как *неизменная переменная* имеет *назначенное один раз* свойство и не может быть изменена или отвязана путем программирования, раз связана со значением. Тем не менее, неизменные переменные могут быть отвязаны механизмом реализации, таким как поиск с возвратами, как в реализации парадигмы логического программирования.

2.4.4 Оболочка и Хранилище

Выполнение программы использует два абстрактных компонента: *оболочка* и *хранилище. Оболочка* - это окружение, где происходит вычисление. Оболочка определяется как набор совокупности пар вида

(имена переменных ↦ ячейки памяти или атрибуты переменных) или (идентификатор имени ↦ значения), как и в случае объявления констант. Новые объявления изменяют оболочку путем (1) создания новых привязок идентификатора вида ↦ ячейки памяти или идентификатор ↦ значения и (2) затенение привязок нелокальных или глобальных переменных, имеющих с локальными переменными конфликтные имена.

После вызова подпрограммы, оболочка меняется. Как изображено на Рисунке 2.17, текущая оболочка подпрограммы состоит из (1) привязки локальных переменных к ячейкам памяти, (2) привязки нелокальных переменных к ячейкам памяти, (3) привязки глобальных переменных к ячейкам памяти, и (4) ячеек памяти в последовательности с доступными вызывающими программами, с использованием параметров ссылки. После того, как вызываемая подпрограмма закончена, оболочка возвращается обратно в среду, доступную для вызова подпрограммы, дополненную оболочкой, созданной динамическими объектами и рекурсивными структурами данных, которые живут за пределами жизни вызываемой подпрограммы, создавшей их.

Хранилище - это множество привязок вида (ячейка памяти ↦ значение) и оно изменяется каждый раз, когда оператор присваивания записывает значение в ячейку памяти, значение инициализируется,

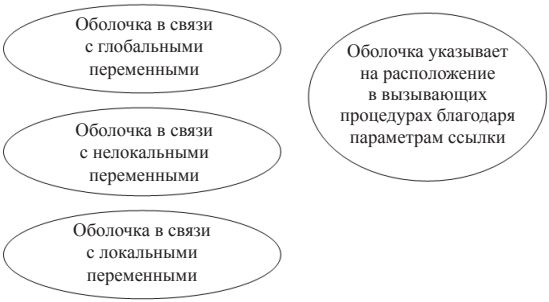


Рисунок 2.17 ОБолочка для выполнения в подпрограмме.

или значение изменяется из-за передачи параметров. Операции присваивания могут обновлять значения, хранящиеся в следующем:

1. 1. Ячейки памяти, связанные с локальными, нелокальными или глобальными переменными.

2. 2. Ячейки памяти переменных в вызывающих подпрограммах, когда параметр передается в качестве параметра ссылки.

3. 3. Ячейки памяти формальных параметров в вызываемой инициализацией программе.

4. 4. Ячейка памяти актуальных параметров, путем передачи результатов, вычисленных в вызываемых подпрограммах.

5. 5. Ячейки памяти рекурсивной структуры данных или динамических объектов данных.

После того, как процедура вызвана, и оболочка, и хранилище получают изменения внутри вызываемой процедуры. Хранилище меняется из-за (1) прохождения значений параметров или (2) инициализации локальных переменных. Во время выполнения команд в подпрограмме, хранилище продолжает получать изменения, как описано выше. После того, как вызываемая процедура закончена, среда, созданная локальными переменными (в вызываемой подпрограмме) теряется; архивная среда вызывающей подпрограммы восстанавливается и дополняется оболочкой глобальных переменных и новой оболочкой, созданной для рекурсивных структур данных и динамических объектов, которые имеют срок службы за пределами вызываемой подпрограммы.

Хранилище не может оставаться таким же после возвращения из вызываемой процедуры из-за разрушительных обновлений глобальных переменных, нелокальных переменных и переменных в процедурах вызова, при передаче в качестве параметров ссылки. Присвоение новых значений этим переменным будет обновлять хранилище вызывающей процедуры, и эти изменения будут сохранены даже после того, как вызванная процедура будет закончена. В дальнейшем мы будем изучать, что такого рода изменения в хранилище могут вызвать непредусмотренное нарушение поведения программы.

2.4.5 Функции и Процедуры

Далее мы обсудим разницу между выражениями и командами, а также функциями и процедурами. Выражение читает одно или несколько значений из памяти и оценивает их. Тем не менее, оно не записывает результат вычисления обратно в память. Это означает, что *хранилище* остается неизменным во время вычисления выражения. В отличие от выражения , команда записывает данные в ячейку памяти, изменяя хранилище. Например, $x + y + 4$ является выражением, в то время как

инструкция присваивания $z = x + y + 4$ является командой. В выражении $x + y + 4$, процессор считывает значение переменных x и y из памяти, а затем вычисляет выражение. Для сравнения, инструкция присваивания $z = x + y + 4$ записывает результат вычисления выражения $x + y + 4$ в ячейку памяти, соответствующую переменной z.

Функция имеет четыре компонента: (1) имя функции, (2) входные параметры,

(3) объявление переменных и (4) набор связных выражений для отображения входных значений на значение выходного сигнала. Функция не имеет инструкции «присваивания», чтобы выполнять обратную запись в хранилище программы. Для сравнения, процедура имеет одну или несколько инструкций присваивания.

В парадигме императивного программирования немыслимо проводить процесс программирования без использования операторов присваивания. Функция с оператором присваивания - это фактически процедура, которая имитирует эффект функции.

2.4.6 Абстрагирование выполнения программы

Выполнение программы может быть абстрагировано, как переход от одного вычислительного состояния в другое до тех пор, пока программа не достигнет конечного состояния, которое удовлетворяет последнему условию. Есть два подхода к модели вычислительного состояния. Первый подход модели вычислительного состояния σ выглядит как тройка вида (σ^E, σ^S, σ^D), где символ σ^E является *оболочкой*, символ σ^S является *хранилищем*, а символ σ^D является стеком из пар (часть *оболочки*, в вызывающей процедуры не требуется [или затеняется] в вызываемой процедуре, часть хранилища в вызывающей подпрограмме не разделяют вызываемой процедуры) цепочки вызова подпрограмм вызовов в порядке, обратном порядку поступления . Второй подход модели вычислительного состояния выглядит как булевое выражение, соединяющее логические аксиомы, используя логические операторы, такие как логическое ИЛИ, логическое И, отрицания и импликации. Булевое выражение изменяется каждый раз, когда выполняется оператор присваивания. Второй подход не зависит от машины фон Неймана и используется для рассуждения о правильности программы, как описано в главе 3, и поэтапой реконструкции программы, начиная с конечного состояния, как описано в главе 4.

2.4.6.1 Вычислительные Состояния как Тройка (Оболочка, Хранилище, Дамп)

В первом подходе вычислительные состояния σ изменяются после выполнения инструкции из-за (1) изменения оболочки σ^{E}, вызванной объявлениями; (2) изменений в хранилище, обозначенном σ^{S}, вызванных оператором присваивания или инициализацией; или (3) изменения в дампе, обозначенного σ^{D}, из-за вызова функции или процедуры. После оператора объявления, текущая оболочка σ^{E} становится σ'^{E}, а вычислительное состояние изменяется. После нового оператора присваивания, хранилище σ^{S} изменяется в новое хранилище σ'^{S}, и вычислительное состояние изменяется.

После вызова процедуры, пара вида (часть оболочки вызывающей процедуры, и часть хранилища в вызывающей подпрограмме не разделяют вызываемой процедуры) выталкивается в дамп σ^{D}, чтобы дать новое вычислительное состояние (σ'^{E}, σ'^{S}, σ'^{D}). После возврата из функции или процедуры, оболочка отбрасывается путем объявления локальных переменных в вызываемой процедуре, сохраненная часть оболочки вызывающей подпрограммы выталкивается, а объединение остальной части оболочки и вытолкнутой части оболочки дает новую оболочку σ^{Enew}. Новая оболочка σ^{Enew} процедуры вызова будет такой же, как старая σ^{E}, если нет структуры данных с временем жизни больше, чем время жизни вызываемой подпрограммы. Аналогичным образом, в случае с хранилищем, хранилище, связанное с локальными переменными в вызываемой процедуре отбрасывается, а объединение оставшейся части хранилища в вызываемой процедуре и часть хранилища вызывающей процедуры извлеченная из дампа становится новым хранилищем σ^{Snew} процедуры вызова. Обратите внимание, что σ^{Snew} не может быть таким же, как хранилище σ^{S}. Конечное состояние достигается после успешного выполнения программы.

Пример 2.20

Рассмотрим пример, приведенный на Рисунке 2.18. Программа имеет три переменные: *x, y, z*. Переменные *x* и *y* имеют глобальную область видимости, а переменная *z* является локальной по отношению к основной части. Основная часть присваивает значения всех трех переменных и вызывает функцию Swap (обмен) путем передачи ссылки переменных *x* и *y*. Дескриптор '*' в теле замены разыменовывает адрес и читает (или записывает) в ячейки памяти, обеспечивающие соответствующие глобальные переменные *x* и *y*. Процедура *замены* меняет местами

значения глобальных переменных x и y.

Начальное вычислительное состояние {< >, < >, < >}: оболочка, хранилище и дамп пустые. После объявления глобальных переменных x и y, идентификатор x отображается на адрес *l -значение1* и идентификатор y отображается на адрес *l-значение2*, а оболочка становится <$x \mapsto$ *l-значение1*, $y \mapsto$ *l-значение2*>. Новое вычислительное состояние

{<$x \mapsto$ *l-значение1*, $y \mapsto$ *l-значение2*>, < >, < >}. Внутри основной части объявление

переменной целого числа z изменяет оболочку на <($x \mapsto$ *l-значение1*, $y \mapsto$ *l-значение2*, $z \mapsto$ *l-значение3*> Новое вычислительное состояние: {<($x \mapsto$ *l-значение1*, $y \mapsto$ *l-значение2*, $z \mapsto$ *l-значение3*>, < >, < >}. После выполнения каждого из трех операторов присваивания, хранилище меняет изменения вычислительного состояния. После выполнения оператора присваивания $x = 4$, новое хранилище становится <*l-значение1* $\mapsto$ *4*>, а новое вычислительное состояние становится {<($x \mapsto$ *l-значение1*, $y \mapsto$ *l-значение2*, $z \mapsto$ *l-значение3*>, <*l-значение1* $\mapsto$ *4*>, < >}. После выполнения операторов $y = 10$ и $z = 12$, новое вычислительное состояние будет {<($x \mapsto$ *l-значение1*, $y \mapsto$ *l-значение2*, $z \mapsto$ *l-значение3*>, <*l-значение1* $\mapsto$ *4, l-значение2* $\mapsto$ *10,*

l-значение 3 $\mapsto$ *12*>, < >}.

Вызов процедуры *обмена* толкает локальную часть основной программы на дамп, а идентификаторы m и n отображаются на место нахождения *l-значения4* и *l-значения 5* соответственно. Место расположения *l-значения4* отображается на место расположения *l-значения1* согласно ссылке; а место расположения *l-значения5* отображается на место расположения *l-значения2* согласно ссылке. Оболочка и хранилище для глобальных переменных x и y сохраняются, в то время как пары (*оболочка для локальных переменных*, соответствующее *хранилище*) основной части помещаются в *дамп*. Новое вычислительное состояние перед началом объявлений в процедуре *swap* становится {<$x \mapsto$ *l-значение1,*

```
integer x, y;
procedure swap( integer x, y);
main ( )
{ integer z;
x = 4; y = 10; z = 12;
swap(ref x, ref y)
print( x, y, z);
}
procedure swap(integer *m, *n)
{ integer temp;
temp = *m; *m = *n; *n = temp;
}
```

Рисунок 2.18 Программа иллюстрации вычислительных состояний.

y ↦ *l-значение*$_2$, *m* ↦ *l-значение*$_4$, *n* ↦ *l-значение*$_5$>, <*l-значение*$_1$ ↦ *4, l-значение*$_2$ ↦ *10, l-значение*$_4$ ↦ *l-значение*$_1$, *l-значение*$_5$ ↦ *l-значение*$_2$>, < (<*z* ↦ *l-значение*$_3$>, <*l-значение*$_3$ ↦ *12*>)>}. Объявление «`integer temp` « меняет оболочку, и новая оболочка становится следующей: <*x* ↦ *l-значение*$_1$, *y* ↦ *l-значение*$_2$, *m* ↦ *l-значение*$_4$, *n* ↦ *l-значение*$_5$, *шаблон* ↦ *l-значение*$_6$> Вычислительное состояние изменяется соответствующим образом. Оператор присваивания `temp` =*m включает разыменовывание адреса: ячейка памяти *l-значение* $_6$ сопоставляется со значением, хранящемся в ячейке памяти *l-значение* $_1$, и новое хранилище становится вида <*l-значение*$_1$ ↦ *4, l-значение*$_2$ ↦ *10, l-значение*$_4$ ↦l*л-значение*$_1$, *l-значение*$_5$↦*l-значение*$_2$, *l-значение*$_6$↦*4*>. *Разыменовывание* использует транзитивность функции отображения между l-значением 4↦ l-значение 1 и l-значение 1 ↦ 4 для доступа к значению 4. Оператор присваивания*m = *n записывает разыменованное значение *n* в место расположения l-значения 1, и новое хранилище становится вида <*l-значение*$_1$ ↦ *10, l-значение*$_2$ ↦ *10, l-значение*$_4$ ↦ *l-значение*$_1$, *l-значение*$_5$ ↦ *l-значение*$_2$, *l-значение*$_6$ ↦ *4*>. Оператор присваивания *n = temp записывает значение шаблона в разыменованное место нахождения *l-значения* $_2$. Новое хранилище становится вида <*l-значение*$_1$ ↦ *10, l-значение*$_2$ ↦ *4, l-значение*$_4$ ↦ *l-значение*$_1$, *l-значение*$_5$ ↦ l-*значение*$_2$, *l-значение*$_6$ ↦ *4*>. Вычислительное состояние до возвращения из подпрограммы обмена: {<*x* ↦ *l-значение*$_1$, *y* ↦ *l-значение*$_2$, *m* ↦ *l-значение*$_4$, *n* ↦ *l-значение*$_5$, *temp* ↦ *l-значение*$_6$>, <*l-значение*$_1$ ↦ *10, l- значение*$_2$ ↦ *4l-значение*$_4$ ↦ *l-значение*$_1$, *l-значение*$_5$ ↦ *l-значение*$_2$, *l-значение*$_6$ ↦ *4*>, < (<*z* ↦ *l-значение*$_3$>, <*l-значение*$_3$ ↦ *12*>)>}. После возвращения из процедуры `swap` , локальная оболочка и соответствующее хранилище для процедуры *swap* удаляются из оболочки и хранилища, локальная

оболочка и хранилище основной части извлекаются из дампа, и дамп снова становится пустым. Новым вычислительным состоянием после возврата из процедуры обмена становится $\{<x \mapsto l\text{-значение}_1, y \mapsto l\text{-значение}_2, z \mapsto l\text{-значение}_3>, <l\text{-значение}_1 \mapsto 10, l\text{-значение}_2 \mapsto 4, l\text{-значение}_3 \mapsto 12>, < >\}$.

Вычислительное состояние как Булев предикат

Во втором подходе, логическое выражение, определяющее вычислительное состояние изменяется после выполнения оператора присваивания. Оператор присваивания $x = 5$ делает предикат $x == 5$ истинным. После выполнения оператора $y = 6$, конъюнктивный предикат $x == 5 \wedge y == 6$ становится истинным. Этот абстрактный подход нужен, чтобы смотреть на выполнение программы, освободив ее от какой-либо конкретной архитектуры, и был использован для анализа программ для их правильности.

2.4.7 Процессы и Потоки

Современные языки программирования, которые поддерживают парадигму параллельного программирования и языки сценариев, используют понятия *процессов* и *потоков*. Активная часть программы или подпрограммы называется *процессом*. Процесс имеет свой *собственный идентификатор*, *хип*, *стек* и *область памяти* для взаимодействия с другими процессами и булевыми флагами для сохранения состояния выполнения процесса. Процесс хранится в блоках памяти. Блоки памяти, соответствующие процессу загружаются в ОЗУ с жесткого диска для выполнения процесса. Как показано на рисунке 2.19, процесс может быть в пяти состояниях: *создан, готов, активен, приостановлен* и *прекращен. Активный процесс* приостанавливается для того, чтобы (1) дать процессу шанс, основываясь на стратегии планирования; (2) ждать, пока ввод/вывод будет завершен; или (3) позволить процессу приостановиться в течение определенного времени, на основании команды во время выполнения. *Приостановленный процесс* активизируется после того, как данные доступны от операций ввода/вывода, или планировщик дает еще один поворот к процессу.

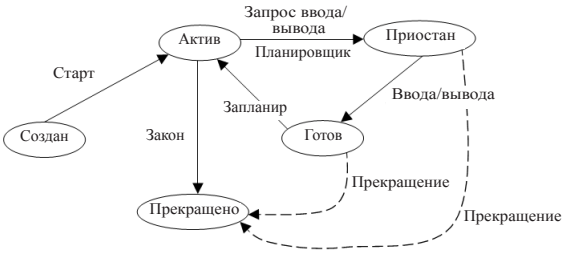


Рисунок 2.19 Диаграмма состояний процесса с обозначенными переходами.

Поток представляет собой последовательность действий. Разница между потоком и процессом заключается в том, что поток имеет меньшие накладные расходы на выполнение и реактивацию, поскольку он разделяет бОльшую часть области данных существующего процесса, который создает этот поток. Процесс может создавать несколько потоков, которые возвращаются обратно к созданному процессу после того, как подзадачи выполнены. Во время выполнения потока процесс может быть запущен или приостановлен в ожидании сигнала, что поток прекращен. Поток прекращается либо после завершения подзадачи, либо прерывается действиями операционной системы.

2.4.8 Буферные области

Когда два процесса или потока взаимодействуют друг с другом, или передают данные друг другу, они могут сделать это в асинхронном режиме при разной скорости. многократно используемый объем памяти необходим для облегчения асинхронной передачи данных. Такой объем памяти называется *буфером*. Существуют две основные операции в буфере: (1) внесение данных и (2) получение данных.

Буферы реализуются с использованием *кольцевой очереди*. Как показано на Рисунке 2.20, кольцевые очереди непрерывно многократно используют память, освобожденную удаленными элементами данных. Кольцевые очереди используют арифметические операции по модулю на линейном массиве, чтобы вернуться к началу массива после обнаружения последнего элемента памяти в массиве.

Существуют четыре операции в буфере: (1)проверить, пуст ли буфер, (2)проверить, заполнен ли буфер, (3)удалить элемент из непустого буфера, и (4)вставить элемент

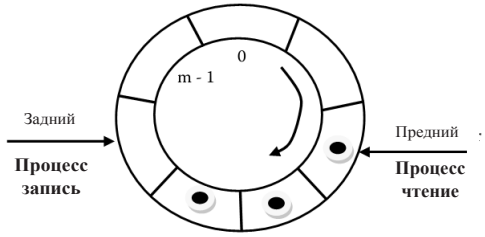


Рисунок 2.20 Схематическое изображение буфера как кольцевой очереди.

в не полный буфер. Предположим, что буфер реализован в виде массива размером m; тогда операции на буфере будут выглядеть следующим образом:

```
is_empty_buffer(buffer) ::    if (front-pointer == rear-pointer) return(true)
                              else return(false);
is_full_buffer(buffer) ::     if (((rear-pointer + 1) modulo m)
                              == front-pointer) return(true)
                              else return(false);
insert(buffer, element) ::    if not (is_full_buffer(buffer)) {
                              buffer[rear-pointer] = element;
                              rear-pointer = (rear-pointer + 1)
                              modulo m;}
                              else raise_exception('buffer_full');
remove(buffer)::              if not (is_empty_buffer(buffer)) {
                              element = buffer[front-pointer];
                              front-pointer = (front-pointer + 1)
                              modulo m; return(element);}
                              else raise_exception('buffer_empty')
```

Операция *is_empty*(*buffer*)возвращает значение *истина*, если значение заднего указателя равно значению переднего указателя. Операция *is_full*(*buffer*) проверяет, даст ли увеличивающийся задний указатель на 1 с использованием арифметических операций по модулю значение переднего указателя. Это означает, что задний указатель догнал передний указатель, и не существует никаких свободных клеток. Операция *insert*(*buffer, element*) сначала проверяет, не заполнен ли буфер. Элемент данных вставляется только если буфер не заполнен. Элемент данных

вставляется в ячейку, индексированную задним указателем. После вставки элемента в ячейку, задний указатель увеличивается на 1 с использованием арифметических операций по модулю. Операция *remove*(*buffer*)удаляет элемент данных из местоположения, индексированного передним указателем, если буфер не пуст, а передний указатель увеличивается на единицу с использованием арифметических операций по модулю.

2.5 КРАТКИЕ ВЫВОДЫ

В разделе были описаны различные основные понятия, используемые в разработке и реализации языков программирования. Эти понятия играют важную роль в разработке и реализации языков программирования и станут актуальным в следующих главах.

Концепции дискретных структур, необходимые для изучения языков программирования включают в себя множества и мультимножества, логические операторы, функции и отношения, конечный автомат, и множество операций. Теория типа широко использует теории множеств, и многие языки программирования поддерживают объявления множеств и операций на основе множеств. Многие операции над множествами, такие как множества всех подмножеств, декартово произведение, конечное отображение и несвязное объединение были обсуждены.

Булева логика является одной из основ программирования. Существует класс функций (или высказываний в логических программах), которые называются *предикатами*, которые возвращают значение *истина* или *ложь*, в зависимости от значений переменных. Проверка на сложные логические условия используется во многих абстракциях управления, таких как извлечение (условный оператор, оператор выбор и т.д.) и неопределенные итерации (цикл WHILE DO, цикл DO WHILE), обработка исключений и условные переходы, такие как выход из вложенных блоков или постепенное прекращение выполнения, если событие уже произошло. Булевы условия формируют ядро логического программирования, основанное на соединении логических предикатов, используя логическое И, логическое ИЛИ, импликации и отрицания.

Функция представляет собой отображение между двумя множествами: *доменом* и *областью изменения* таким образом, что каждый элемент домена отображает только один из области изменения. Отображение должно быть выполнено через ассоциацию «one-to-one» или «many-to-one». Тем не менее, оно не всегда может быть таковым. Парадигма функционального программирования всецело использует функции, в то время как другие парадигмы программирования, такие как парадигма

императивного программирования, используют функции наряду с утверждениями. Функциональная программа использует только вычисления выражений. В парадигме императивного программирования функции также используют утверждения в дополнение к вычислениям выражений.

Отношение определяет свойство между двумя элементами (или элементом и атрибутом) *x* и *y*. Отношение может быть комбинацией *возврата, четности* (или *антисимметричности*), или *перехода*. Свойства отношений используются в программировании, таком как логическое программирование, замещении программных переменных и анализе зависимости данных для перевода программы для одновременного запуска.

Рекурсия характеризуется наличием двух типов определений: *базовый случай* и *рекурсивные определения.* В языках программирования, рекурсия происходит как в определениях процедур, так и в определениях структуры данных. Рекурсивные функции играют важную роль в программировании. *Функции хвостовой рекурсивности* могут быть смоделированы с использованием итеративных программ. Класс *линейных рекурсивных* программ может быть реализован при использовании понятия *накопителей* и итерации. Обработка рекурсии нуждается в предназначении для роста памяти во время выполнения, так как требования к памяти рекурсивных процедур и рекурсивных структур данных не могут быть определены во время компиляции. Некоторые из накладных расходов рекурсивного программирования могут быть облегчены путем преобразования программ хвостовой и линейной рекурсии для соответствующих итерационных программ.

Конечные автоматы - это модельные переходы между различными ситуациями выражающимися в состояниях, которые использовались в языках программирования во время лексического анализа- фазы во время компиляции, которая обсуждается в главе 3.

Стеки и очереди (списки) - это абстрактные типы данных с различными свойствами: стек поддерживает свойство LIFO (в порядке, обратном порядку поступления), а очередь поддерживает свойство FIFO (в прямом порядке). Элементы выталкиваются и извлекаются из только с одного конца в стеке, в то время как элементы данных вводятся с одного конца в очереди и выводятся из другого конца. Стеки играют важную роль в обходе дерева поиском в глубину, выполнении рекурсивных процедур и реализации языков программирования. Очередь используется при обходе дерева поиском в ширину и там, где требуется свойство FIFO. Обход

поиском в глубину проходит глубже в дерево, так как поиск достигает конечных узлов в левом поддереве перед обходом в правом поддереве. Поиск в ширину обходит дерево уровень за уровнем. Поиск в ширину использовался в некоторых эффективных методах управления хипом, как обсуждалось в главе 6.

Хэш-таблицы используются для эффективного ввода и поиска динамических данных путем сопоставления первичного ключа к индексу, используя хэш-функцию. Использование функции хеширования гарантирует, что индекс идентифицируется в ближайшее постоянное время. Хэш-таблицы играют важную роль в реализации языков программирования и в управлении процессами, так как информация может быть получена в почти постоянном времени.

Переменная является основным держателем информации в программе. Переменная содержит шесть основных атрибутов: *имя*, *тип*, *объем*, *срок жизни*, *местоположение* в *памяти*, а также *присвоенное значение*. *Область видимости* является частью программы, где переменная может быть прочитана или записана, а *время жизни* -это период, когда переменная может быть использована. *Глобальные переменные* имеют один и тот же объем и время жизни, как и программы, в то время как *локальные переменные* ограничены объемом процедуры, в которой они объявлены. *Статические переменные* распределяются по фиксированным адресам во время компиляции и доступны с помощью прямого доступа к памяти. *Изменяемые переменные* могут быть деструктивно обновлены и использованы в парадигме императивного программирования для операторов присваивания.

Во время выполнения программы, есть три важных понятия: *оболочка*, которая изменяется за счет объявлений; *хранилище*, которое изменяется за счет операторов присваивания; и *дамп* -стек оболочки и хранилище последовательности вызывающих процедур. Выполнение программы может быть смоделировано как последовательность переходов между вычислительными состояниями, описываемыми отвлеченно либо как тройка вида (*оболочка, хранилище* и *дамп*) или в виде *логических выражений,* включающих конъюнкции, дизъюнкции и отрицания булевых предикатов. Второй подход освобождает определение вычислительного состояния машины фон Неймана и используется, чтобы рассуждать о корректности программ.

Процесс является активной частью программы или подпрограммы, которая запущена на компьютере и занимает память в ОЗУ. *Поток* представляет собой последовательность действий, которые действуют

как процесс, за исключением того, что поток имеет меньше накладных расходов распределения памяти, чем процесс и использует пространство памяти родительского процесса. Буферы представляют собой пространства памяти, используемые для передачи данных между двумя процессами, потоками, между двумя устройствами ввода/вывода, или между процессором и устройствами ввода/вывода. Буфер моделируется как кольцевой буфер, который использует арифметические операции по модулю, чтобы повторно использовать пространство памяти, освобожденное удалением элементов данных из буфера.

2.6 ОЦЕНКА

2.6.1 Концепции и Определения

Накопитель; ациклический граф; антисимметричность; связывание; Булева логика; поиск в ширину; буфер; Декартово произведение; кольцевая очередь; команда; вычислительное состояние; цикл; циклический граф; поиск в глубину; направленный ациклический граф; ориентированный граф; дамп; правило динамической области; оболочка; использование кванторов существования; выражение; конечное отображение; конечный автомат; исчисление предикатов первого порядка; функция; граф; хэш-функция; хэш-таблица; неизменный; следствие; индексация; линейная рекурсия; логическое И; логическое ИЛИ; отображение; мультимножества; изменяемое; отрицание; одноадресная вычислительная машина; упорядоченное множество; указатель; показательное множество; исчисление предикатов; процесс; исчисление высказываний; кванторы; очередь (списки), рекурсии; рекурсивная структура данных; рекурсивная функция; ссылка; возвратность; связь; правила видимости; последовательность; стек; статическая область видимости; хранилище; строка; симметричность; хвостовая рекурсия; поток; трехадресная машина; транзитивность, дерево; кортеж; двуадресная машина; тип переменной; использование кванторов всеобщности; переменная; видимость; машина фон Неймана; взвешенный граф; машина с нулевым адресом.

Решение Задач

1. Записать декартово произведение двух множеств: {«утро», «вечер», «ночь», «день»} и { «солнце», «свет», «дождь»}.

2. Записать показательное множество множества, которое дано как

{солнце, свет, дождь}. Объяснить размер показательного множества.

3. Дана функция «вычитание»-5, которая вычитает число пять из области чисел натурального ряда и распределяет элементы в расширенной области натуральных чисел, показывая отображение элементов 10, 0, 3, 7 и 8. Обратите внимание, что расширенная область также содержит нижний символ ⊥. Все, что не отображается для регулярных элементов в области значений, отображается для нижнего символа.

4. Записать представление решения исчисления предикатов для следующего утверждения: «для каждого числа *N* такого, что *N*>1, существует число, которое на 1 меньше, чем это число.»

5. Записать представление решения исчисления предикатов для следующего утверждения: «Для каждого человека в этом мире, существует связь, которая соединяет этого человека с, по крайней мере, еще одним человеком в этом мире.»

6. Дан трехмерный массив, имеющий измерение размера [0..4, 0..7, 0..9] и базовый адрес 10,000, дайте начальный адрес элемента, расположенного в индексе (3, 4, 2), при условии, что каждый элемент занимает 2 байта памяти. Поясните свои вычисления.

7. Дана четверка (4-кортеж) вида (целое число, символ, плавающая точка, целое число), дайте смещение третьего поля, учитывая, что стартовый адрес 5000. Предположим, что целое число занимает четыре байта, число с плавающей точкой занимает восемь байт, а символ занимает один байт.

8. Показать снимок состояния стека после каждой операции, начиная с пустого стека, для следующей последовательности операций: push (stack, 4), push (stack, 5), pop (stack); push (stack, 6). Представьте стек в виде последовательности вида <элемент$_N$, **...**, элемент$_1$>, где элемент $_N$, ..., элемент $_1$ - это те элементы, которые были протолкнуты в стеке в порядке LIFO, и элементы добавлены и удалены от переднего конца.

9. Показать снимок состояния очереди после каждой операции, начиная с пустой очереди, для следующей последовательности операций: insert (queue, 4), insert (queue, 5), remove (queue); insert (queue, 6). Представьте очередь, как последовательность вида <элемент$_1$, …, элемент$_N$>, где элемент$_1$, …, элемент$_N$ - это элементы в очереди. Элементы вставляются с заднего конца последовательности и удаляются

с переднего конца последовательности.

10. Напишите программу для поиска в ширину, используя очереди и для поиска в глубину, используя стек, и сравнить среднее число элементов данных, сохраненных в очереди против стека. Реализуйте очередь с использованием вектора. Создайте как минимум 10 деревьев во время выполнения с использованием генераторов случайных чисел для статистического анализа.

11. Напишите простую программу с использованием, по меньшей мере, пяти переменных и напишите простой код с пятью простыми операторами, и покажите, как оболочка и хранилище изменяются после каждого утверждения и после выполнения каждого оператора.

Возьмите программу с, по крайней мере, пятью различными процедурами, с, по крайней мере, двумя процедурами, которые были вызваны несколько раз из различных процедур. Представьте каждую процедуру, отметив узел в графе. Нарисуйте ориентированное ребро от вызывающей к вызываемой процедуре. Вес ребра определяется числом вызовов процедур. Если процедура вызывает саму себя, тогда нарисуйте ребро от узла к самому себе, показывая цикл. Опишите получившийся граф.

2.6.3 Развернутый ответ

13. Что вы понимаете под исчислением предикатов? Объясните. Каковы различные компоненты исчисления предикатов? Объясните каждый из них с помощью простого примера.

14. В чем разница между исчислением предиката первого порядка и высшего порядка? Объясните.

15. Объясните разницу между стеком и очередью.

16. Что вы понимаете под направленным ациклическим графом? Чем они отличаются от деревьев? Дайте реальный пример задачи, которая может быть смоделирована при помощи направленных ациклических графов, но не при помощи деревьев.

17. Каковы преимущества и недостатки использования указателей в языках программирования? Объясните. Обсудите три проектных решения, которые позволили бы уменьшить недостатки использования указателей в языках программирования.

18. Объясните поиск в глубину, ширину и их различия. Объясните, используя наглядный пример сбалансированного N-арного дерева ($n \geq 2$).

19. Объясните статическую и динамическую область видимости, используя простой пример, не приведенный в книге.

20. Объясните разницу между оболочкой и хранилищем. Покажите, как оболочка и хранилище изменяются при использовании объявлений, операторов присваивания, вызова процедуры и вычисления выражения.

21. Объясните, каким образом механизм хэширования работает для эффективного поиска и извлечения данных из архивных данных, используя простой, но полный пример, не приведенный в книге.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Давендер С. Малик и Мридул К. Сен. *Дискретные математические структуры: Теория и Практика*.

Thomson Course Technology. 2004.

Давендер С. Малик. *Структуры данных*, *использующие C++*, издание второе. Course Technology Cengage Learning.

2009.

Дэвид А. Паттерсон и Джон Л. Хеннесси *Архитектура компьютера и проектирование компьютерных систем, издание пятое*. Morgan Kaufmann. 2012.

Зильбершатц, Абрахам, Галвин, Петер Б., и Гайне, Грег. *Концепции операционных систем*, издание девятое.

John Wiley and Sons. 2010.

Глава 3

Синтаксис и Семантика

БАЗОВЫЕ КОНЦЕПЦИИ

Абстрактные понятия в вычислениях (раздел 2.4); Булева логика (раздел 2.2.2); Диаграммы потоков управления (глава 1); Дискретные структуры (раздел 2.2); Оболочка и хранилище (раздел 2.4.4); Конечные автоматы (раздел 2.2.5); Рекурсия (раздел 2.2.4); Деревья (раздел 2.3.5); Графы (Раздел 2.3.6); Машина фон Неймана (раздел 2.1).

Существует два основных компонента для понимания конструкции языков программирования: *синтаксис* и *семантика. Синтаксис* заключается в проверке допустимости структур оператора в языках программирования, а семантика заключается в понимании точных значений предложений в языках программирования. Обе эти концепции имеют важное значение для понимания программы, разработки компилятора, и обслуживания программного обеспечения. Синтаксис имеет важное значение, потому что, если структура предложения не будет подтверждена должным образом, то она не будет являться частью языка программирования и не сможет быть связана с правильным значением. *Семантика* имеет важное значение, потому что пока мы не сопоставляем и не передаем однозначный смысл для каждого слова, используемого в программах и конструкциях языков программирования, предложение в программе не может быть понято и правильно переведено в код промежуточного уровня для выполнения.

3.1. ВВЕДЕНИЕ В СИНТАКСИС И СЕМАНТИКУВ большинстве естественных языков, за исключением графических языков, таких как китайский, содержатся четыре основных компонента синтаксиса: *конечный набор символов, слов, фраз* и *грамматические правила* (также известные как продукционные правила), чтобы построить и

обосновать предложения. Символы делятся на *гласные* и *согласные*. Гласные используются для связывания согласных, и чтобы помогать произносить слова. Слова являются основной единицей для связывания значения; никакого значения не привязано на уровне символов. Эти слова дополнительно соединены с помощью грамматических правил (продукционных правил), чтобы сформировать предложения. Если значения отдельных слов являются уникальными, то может быть получен и уникальный смысл для данного предложения.

Для того, чтобы получить смысл данного предложения, во-первых, мы должны применять грамматические правила несколько раз, чтобы согласовать структуру предложения. Процесс подтверждения грамматической правильности предложения, используя продукционные правила, в грамматике называется *синтаксическим анализом*. В процессе подтверждения, предложение все больше сокращается до множества *промежуточных*

Рисунок 3.1 Простая английская грамматика и соответствующее дерево синтаксического разбора.

форм, используя продукционные правила. Процесс завершается после того, как промежуточная форма становится лишь начальным символом. Графическое представление этого прогрессивного применения продукционных правил помогает добраться до стартового символа, начиная с предложения, является деревом и называется *деревом синтаксического разбора*. Подробности синтаксического анализа и дерева синтаксического разбора объясняются в разделе 3.2.

Пример 3.1

Рассмотрим пример английского предложения: «Я играю в баскетбол.» Каждое слово имеет значение. Перед тем, как проверить, является ли предложение грамматически правильным, должна быть проверена орфография. После того, как орфография проверена, применяется

простое подмножество грамматических правил английского языка (см. Рисунок 3.1). Предложение «Я играю в баскетбол» трансформируется в промежуточную форму

<*субъект*> <*глагол*> <*объект*>, который, в свою очередь, трансформируется в другую промежуточную форму <*субъект*> <*предикат*>. Применяя еще одно продукционное правило, промежуточная форма <*субъект*> <*предикат*> сводится к начального символа <*предложение*>.

Семантика получает значение синтаксически правильного предложения в конкретной *области*. Например, строка «214» означает: (1) десятичное число 214_{10} ($2 * 10^2 + 1 * 10^1 + 4 * 10^0$) в

10-ной системе исчисления, (2) 214_8 в восьмеричной системе ($2 * 8^2 + 2 * 8^1 + 4 * 8^0$) = 148_{10} (*148* в *10-ной системе исчисления*),

и (3) «этаж номер два, площадка номер 1, и четвертая комната на площадке» в области строительства. Значение зависит от семантической области, в которой была использована фраза. Для того, чтобы иметь уникальное значение, семантическая область должна быть четко зафиксирована. Например, значение двоичного числа «1011» в 10-ной системе исчисления это $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11_{10}$, *13* в восьмеричной (семантическая область по основанию 8) и «D» в шестнадцатеричной (семантическая область по основанию 16).

Значение десятичного числа <*целая часть*>‘**.**’<*плавающая часть*> в области чисел - это *значение* (<*целая часть*>) + *значение* (<*плавающая часть*>. Если значения нетерминальных символов <*целая часть*> и <*плавающая часть*> известны в индивидуальном порядке, а смысл бинарного оператора ‘+’ понятен, то значение десятичного числа может быть получено. Операции, которые используются для получения значения предложения в *семантической области* называются *семантической алгеброй*. Нам нужна семантическая область и соответствующая семантическая алгебра, чтобы понять смысл предложений в языках программирования.

ГРАММАТИКА

Языки программирования не так сложны как английский язык для того, чтобы избежать неоднозначности при переводе инструкции высокого уровня в эквивалентные инструкции низкого уровня. Многие усилия были предприняты, чтобы сохранить грамматику языков программирования простой:

1. 1. Основные единицы языков программирования больше похожи на слова на английском языке; нет 26 букв, которые составляют слова. Эти основные смысловые единицы называются *ключевыми (зарезервированными) словами* и они связаны с уникальным значением. Набор ключевых (зарезервированных) слов в языках программирования намного меньше, чем в английском языке.

2. 2. Языки программирования избегают нескольких значений для одного слова, которое нуждается в устранении неоднозначности, основываясь на контексте. Использование контекстно-зависимых многосмысловых слов сделает разбор очень времязатратным и склонным к ошибкам при выводе точного значения для предложения на языке. Поскольку значение предложения связано с генерацией кода низкого уровня, ошибка в значении приведет к ошибочному коду низкого уровня. Тем не менее, ограниченное количество словарного запаса имеет несколько значений, и их точные значения получены на основе контекста. Это называется *перегрузкой* и обсуждается в Главе 7.

Грамматика используются для получения *конечной длины* предложения в языке. Грамматика состоит из четырех компонентов: *начальный символ*, *множество терминальных символов (зарезервированные слова), множество нетерминальных символов*, а также *множество продукционных правил. Нетерминальные символы* расширяются до комбинации терминальных и нетерминальных символов с использованием набора продукционных правил. Нетерминальные символы являются лишь частью грамматики, а не частью предложений в языках программирования. Множество всех *терминальных символов* (зарезервированных слов) в языке программирования называется *алфавитом* соответствующей грамматики. Предложения на языке программирования выводятся из алфавита с использованием продукционных правил, начиная с *начального символа. Алфавит* традиционно обозначается как греческий символ Σ. Формально грамматика определяется как 4-ка вида (*начальный символ S*, *набор продукционных правил P, множество нетерминальных символов N*, *алфавит* Σ). Принимая во внимание предложение на языке программирования, продукционные правила в грамматике также используются для проверки структуры предложения. Этот процесс многократного применения продукционных правил в грамматике языка программирования с целью сократить данное предложение до *начального символа* называется

синтаксическим анализом. Правостороннее выражение продукционных правил сопоставляется с частью предложения или промежуточной формой и замещается левосторонним выражением продукционных правил. Этот процесс восстановления с использованием продукционных правил создает структуру дерева под названием *дерево разбора*. Предложение расположено в конечных узлах синтаксического дерева, нетерминальные символы расположены в неконцевых узлах дерева синтаксического разбора, а начальный символ расположен в корневом узле дерева.

Пример 3.2

На рисунке 3.1 показан простой пример использования простого подмножества правил английской грамматики, которые могут передать фразу «Я играю в баскетбол» на английский язык. Грамматика состоит из пяти продукционных правил. Начальный символ это <*предложение*>. Каждое продукционное правило имеет левостороннее и правостороннее выражение, разделенное символом '::='. Левостороннее выражение правила имеет только один символ, заключенный в угловые скобки '<' и '>', а правостороннее выражение правила представляет собой комбинацию двух типов символов: (1) символы, заключенные в угловые скобки и (2) символы без угловых скобок. Символы, заключенные в угловые скобки , называются *нетерминальными символами*, а символы без угловых скобках называются *терминальными символами (зарезервированнми словами)*. Множество нетерминальных символов

{<*предложение*>, <*предикат*>, <*субъект*>, <*глагол*>, <*объект*>}, а алфавит (набор нетерминальных символов) {**Я, Мы, Ты, играть, танцевать, футбол, баскетбол**}.

Дано предложение «Я играю в баскетбол,» правила 3, 4 и 5 применяются соответственно к различным частям предложения путем сопоставления правостороннего выражения продукционных правил с остальными частями в предложении и подставления части предложения соответствующим левосторонним выражениям продукционных правил: правило 3 переводит слово «Я» в нетерминальный символ <*субъект*>, правило 4 переводит слово «играть» в нетерминальный символ <*глагол*>, а правило 5 переводит слово «баскетбол» в нетерминальный символ <*объект*>. Новая промежуточная форма выглядит как <*субъект*><*глагол*><*объект*>. Во втором шаге нетерминальная пара вида

<глагол><объект> сопоставляется с правосторонним выражением правила 2 и выводит нетерминальный символ *<предикат>* (левостороннее выражение правила 2). Новая переведенная промежуточная форма становится вида *<субъект><предикат>*, которая сопоставляется с правосторонним выражением правила 1, и вводит начальный символ *<предложение>*. Таким образом, применение продукционных правил уменьшает предложение «Я играю в баскетбол» до стартового символа *<предложение>* во время проверки структуры. Успешная проверка грамматической структуры не всегда означает, что предложение будет иметь смысл. Например, фраза «*Я танцую* в *футбол*» по своей структуре проверяется с помощью продукционных правил. Тем не менее, предложение не несет смысловую нагрузку.

3.2.1 Типы Грамматики

Есть три типа грамматик, которые могут быть использованы в определении и реализации языков программирования: *регулярная грамматика*, *бесконтекстная грамматика*, и *контекстная грамматика*.

Регулярная грамматика также носит название *грамматика типа 3*, это простейший тип и он представляет собой *недетерминированный конечный автомат*. *Недетерминированный конечный автомат* это конечный автомат, который имеет несколько возможных переходов из одного состояния в другое состояние для одного и того же входного символа. В разделе 2.2.4 главы 2 мы обсуждали конечный автомат, который принимает переменную. Конечный автомат может быть выражен при помощи обычной грамматики:

```
<S1> → <letter> <S2>
<S2> → <letter> <S2> | <digit> <S2> | '_' <S2> | ε
```

Формально, *регулярная грамматика* представлена в виде четверки (*N, Σ, P, S*), где *N*- множество нетерминальных символов, Σ - множество терминальных символов, *P* является набором продукционных правил, а *S* -это начальный символ. В примере, приведенном выше, *N* - это {*<S1>*, *<S2>*}, Σ - это {*любая буква, любая цифра, '_'*}, *P* является набором продукционных правил, а *S* - начальный символ *<S1 >*. *Регулярные грамматики* используются для того, чтобы принять предложение и выдать сигнал в виде маркера при приемке. Учитывая последовательность символов, регулярная грамматика будет принимать последовательность, если она является собственной переменной.

Регулярные грамматики широко используются на лексическом этапе

анализа во время процесса компиляции, генерирующего потоки лексем - усвоенные представления для элементов языка программирования, таких как переменные, литералы, идентификаторы и зарезервированные слова. Лексемы необходимы для облегчения синтаксического анализа.

Бесконтекстная (контекстно-свободная) грамматика, которая также называется *грамматикой 2-го типа*, представлена в виде четверки вида (*N, Σ, P, S)*, где *N* - множество нетерминальных символов, Σ - множество терминальных символов, *P* является набором продукционных правил, а *S* является начальным символом. Основнаой характеристикой контекстно-свободной грамматики является то, что левая часть представляет собой единый нетерминальный символ. Грамматика на рисунке 3.1 является примером контекстно-свободной грамматики. Это ограничение имеет преимущества во времени и пространстве сложности при разборе предложения. Так как левостороннее выражение продукционных правил представляет собой единый нетерминальный символ, то он может быть расширен без присутствия любого терминального символа. Контекстно-свободная грамматика является более выразительной, чем обычные грамматики. Например, не требуется регулярной грамматики, чтобы принять строку $a^n b^n$. Тем не менее, мы можем написать контекстно-свободную грамматику, чтобы принять строку $a^n b^n$, как указано ниже

```
<S> ::= a <S> b | ε
```

Вышеупомянутая грамматика является контекстно-свободной по определению, так как она имеет начальный символ <*S*>, набор нетерминальных символов {<*S*>}, набор терминальных символов {a, Ь}, и одно продукционное правило. Правило имеет только один нетерминальный символ в левостороннем выражении. Греческая буква ε обозначает нулевой символ.

Контекстно-зависимая грамматика, также известная как *грамматика 1-го типа,* может иметь дополнительные терминальные символы на левой стороне выражения в дополнение к нетерминальным символам . Единственное ограничение состоит в том, что длина строки из левой части должна быть меньше или равна числу символов в правой части. Рассмотрим следующие правила грамматики:

```
<S> ::= a<S>c | ε
<S> c ::= b<S>cc
```

Приведенный выше пример иллюстрирует контекстно-зависимую грамматику, поскольку второе продукционное правило имеет терминальный символ ‘c’ в дополнение к нетерминальному символу

<*S*> на левой стороне продукционного правила. Грамматика принимает строку вида $a^n b^m c^{m+n}$ ($m \geq 0$, $n > 1$). Тем не менее, второе правило раскрывается только при наличии терминального символа ‘c’. Контекстно-зависимые грамматики являются более мощными, чем контекстно-свободные грамматики. Например, контекстно-зависимая грамматика может быть записана в форме, принимающей строку $a^n b^n c^n$ ($n > 0$). В то время как контекстно-независимая грамматика может создать строку вида $a^n b^n c^n$. Тем не менее, существует серьезный недостаток использования контекстно-зависимой грамматики: чувствительность контекста значительно увеличивает количество продукционных правил, что замедляет процесс разбора предложения. Из-за соображений эффективности во время синтаксического анализа, определения языка программирования

используют контекстно-свободные грамматики.

3.2.2 Представление грамматики, используя форму Бэкуса-Наура

В первые годы развития компиляторов, двое ученых, работающих в области компьютерных наук, Джон Бэкус и Петер Наур, предложили форму для описания контекстно-свободных грамматик для языков программирования. Предложенная ими форма сегодня известна как *Форма Бэкуса — Наура* или *БНФ*.

В БНФ левые и правые стороны продукции разделяются символом ‘:: =,’ нетерминальные символы заключены в угловые скобки, а множество определений одних и тех же нетерминальных символов отделены друг от друга вертикальной чертой ‘|. Кроме того, грамматика использует греческий символ ‘ε’, чтобы описать нулевое значение, и использует хвостовую рекурсию (см. главу 2, подраздел 2.2.3), чтобы определить любое количество повторений типов объекта. В хвостовой рекурсии, рекурсивная часть приходит в конце определения.

Пример 3.3

Нетерминальный символ <*последовательность-операторов*> определяет хвост рекурсивно. Базовым случаем является нулевой оператор, обозначающийся символом ‘ε,’ а часть хвостовой рекурсии определяется как нетерминальный символ <*оператор*>, за которым следует точка с запятой, а далее

<*последовательность операторов*> следующим образом:

<последовательность операторов> ::= *<оператор>* **‘;’**
<последовательность-операторов> | **ε**

Пример 3.4

Рисунок 3.2 иллюстрирует грамматику БНФ для упрощенного определения арифметического выражения. Контекстно-свободная грамматика имеет начальный символ *<выражение>*, 13 продукционных правил, набор нетерминальных символов, таких как {*<выражение>*, *<А-выражение>*, *<L-выражение>*,

<сравнение>, *<идентификатор>*, *<символы>*, *<цифра или буква>*, *<число>*, *<цифра>*,

<А-оператор>, *<L-оператор>*, *<оператор сравнения>*}, и алфавит {**0** ... **9**, ‘**a**’ … ‘**z**’, ‘**A**’ … ‘**Z**’, ‘**+**’, ‘**–**’,

‘*****’, ‘/’, ‘**&&**’, ‘||’, ‘**>**’, ‘**<**’, ‘**>=**’, ‘**=<**’, ‘**==**’}. Запись ‘0’ | ‘1’ | ... | ‘8’ | ‘9’ дает подгруппу, включающую все цифры от 0 до 9.

Выражение *(<выражение>)* является многопрофильным определением. Это может быть либо арифметическое выражение (*<А-выражение>*) или логическое выражение (*<L-выражение>*) или идентификатор (*<идентификатор>*). Арифметическое выражение (*<а-выражение>*) определяется как число (*<число>*) или арифметическое выражение (*<А-выражение>*) с последующим арифметическим оператором (*<А-оператор>*), а затем еще арифметическое выражение (*<А-выражение>*). Логическое выражение (*<L-выражение>*) является *истиной ;*

<выражение> **::=** *<А-выражение>* | *<Л-выражение>*
<L-выражение> **::= истина** | **ложь** | *<идентификатор>* | **не** *<L-выражение>* |
<сравнение> | *<L-выражение>* *<L-оператор>* *<L-выражение>*
<сравнение> **::=** *<А-выражение>* *<оператор сравнения>* *<А-выражение>*
<А-выражение> **::=** *<число>* | *<идентификатор>* | *<А-выражение>*
<А-оператор> *<А-выражение>*
<идентификатор> **::=** *<буква><символ>* | *<буква>*
<символы> **::= ε** | *<цифра или буква><символы>*
<цифра или буква> **::=** *<цифра>* | *<буква>*
<число> **::=** *<цифра>* | *<число><цифра>*
<цифра> **::= ‘0’** | **‘1’** | **… |’8’** | **‘9’**
<буква> **::= ‘a’**| **‘b’** | **… |** ‘**z**’| ‘**A**’ | ‘**B**’| **…**| ‘**Z**’
<А-оператор> **::=** ‘ **+** ’ | ‘**–**‘ | ‘/’ | ‘*****’
<Л-оператор> **::=** ‘**&&**’ | ‘||’
<оператор-сравнения> **::= ‘>’ | ‘<’** | ‘>=’ | ‘=<’ | ‘ ==’

Рисунок 3.2 БНФ грамматика для определения выражения.

ложью, сравнением двух арифметических выражений, двух логических выражений (<*L-выражение*>), соединенных логическим оператором (<*L-оператор*>), или отрицание логического выражения. Сравнение *(*<*сравнение*>) сравнивает два арифметических выражения с использованием операторов сравнения (<*Оператор сравнения*>). Идентификатор (<*идентификатор*>) является буквой, следующей за нетерминальным символом

<*символ*>), который равен любому количеству повторений <*цифра*> или <*буква*>. Число (<*число*>) определяется рекурсивно как число (<*число*>) с последующей цифрой (<*цифра*>). Леворекурсивное определение <*числа*> будет использоваться в следующем разделе для объяснения денотационной семантики в разделе 3.5. Цифра (<*цифра*>) может быть любым элементом от 0 до 9. Арифметический оператор (<*A-оп*>) является '+', '-', '*', или '/' и логический оператор может быть логическим И ('&&') или логическим ИЛИ ('||'). Грамматика неоднозначна из-за отсутствия приоритета операций, и может дать два или более деревьев разбора для одной и той же фразы, как показано на рисунках

3.12 и 3.13.

3.2.3 Расширенная форма Бэкуса — Наура (РБНФ)

Несмотря на свою выразительность , БНФ не может определить некоторые ситуации легкие для человеческого понимания, как то:

1. 1. БНФ использует несколько определений даже если изменяются небольшие части продукционного правила. Это приводит к ненужному развертыванию правил. Например, мы можем определить арифметическое выражение как <*A-выр*> ('+'|'−'|'*'|'/') <*A-выр*>,, где конструкции ('+'|'−'|'*'|'/') показывают одну из возможностей и объединяют два правила в одно правило для лучшего понимания.

2. 2. БНФ использует хвостовую рекурсию для обработки любого количества повторений символа. Хвостовую рекурсию проще выразить, используя итерацию. В примере 3.1, вместо определения <*последовательность операторов*> , она может быть выражено как {<*оператор*> ';'}* , где обозначение '*' обозначает любое количество повторений нетерминального символа <*оператор*> , следующего за терминальным символом';'.

3. 3. БНФ не определяет дополнительное повторение подраздела

продукционного правила и описывает их как определение или нулевой символом ε.

РБНФ использовались для снятия данного ограничения, присутствующего в БНФ. Таблица 3.1 показывает замены, используемые в РБНФ. Группировка нескольких определений использует круглые скобки и вертикальную черту вида (вариант 1 | вариант 2). Дополнительные возможности в расширении правостороннего выражения продукционного правила выражаются, используя квадратные скобки вида [дополнительная функция]. Повторение использует фигурные скобки вокруг повторяющегося понятия, дополненного различными

Рисунок 3.1 Преобразование грамматики БНФ в грамматику РБНФ

Представление БНФ	**Представление РБНФ**
<*NT*> **::=** Вариант$_1$ | Вариант$_2$	<*NT*> **::=** (Вариант$_1$ Вариант$_2$)
<*NT*> **::= ε** | Дополнительная функция	[Дополнительная функция]
<*NT*> **::=** Символ<*NT*> | Символ	<*NT*> **::=** {Символ}$^+$
<*NT*> **::=** Символ <*NT*> | **ε**	<*NT*> **::=** {Символ}*
<*NT*> **::=** '**0**'|'**1**'|'**2**'|'**3**'	<*NT*> **::=** '**0**'–'**3**'

символами, такими как '*', '+' или цифрой, чтобы описать степень повторения. Например, число определяется как {<*цифра*>}$^+$, означая один или несколько повторов <*цифра*>; а <*идентификатор*> определяется как <*буква*> {(< *буква* > | < *цифра* >)}^{254} означает буквы, за которым следует максимум 254 символа (буквы или цифры). Благодаря использованию скобок, квадратных скобки, угловых скобок, вертикальных полос, '+' и '*' со специальными значениями, эти символы помещаются в кавычки при использовании в качестве терминальных символов в языке программирования. Диапазон описывается в сокращенном виде с помощью дефиса между нижней границей и верхней границей диапазона.

Пример 3.5

Рисунок 3.3 показывает РБНФ версию грамматики, описанной на рисунке 3.2. Определения нетерминальных символов <*A-выр*>, <*L-выр*>, и <*сравнение*> были изменены путем включения «групповых» особенностей РБНФ, как описано в таблице 3.1. Определение

нетерминального символа *<L-выр>* использует «дополнительную» особенность РБНФ, в то время как унарный оператор '**не**' может повториться при необходимости перед *<L-выр>*. Соответствующие множества определений в *<L-выр>* были объединены. Нетерминальный символ *<номер>* использует комбинацию «группировка» и «повторение» и определяется как один или несколько повторений группы цифр от 1 до 9. Некоторые из продукционных правил, которые были использованы для изменения других продукционных правил верхнего уровня, не требуются больше, и они были объединены с правилами верхнего уровня.

Грамматика РБНФ далее может быть расширена путем определения *<A-выр>* итеративно как *<число>(*или*<идентификатор>),* следующим за нулевым или большим количеством повторений сгруппированных операторов ('+' | '–' | '*' | '/') и группой *(<число>* | *<идентификатор>*). Нетерминальный символ *<L-выр>* итеративно определяется как группировка (**истина** | **ложь** | *<идентификатор>* |

<сравнение>), следующая за повторением группы операторов сравнений и группировки (**истина** | **ложь** | *<идентификатор>* | *<сравнение>*). Правосторонее выражение [**не**] *<L-выр>* трансформировалось в группу [**не**] (**истина**| **ложь**| *<идентификатор>* |*<сравнение>*) путем откидывания "[**не**]." Модифицированная РБНФ грамматика приведена на рисунке 3.4.

Пример 3.6

Рисунок 3.5 иллюстрирует еще один реалистичный пример определения грамматики для *итеративных утверждений* для изучения представлений синтаксиса, используя представление РБНФ. Названия говорят сами за себя. *<итерация>* имеет множество определений: *<цикл for>*, *<цикл while>*,*<цикл do_while>*, и *<итератор>*. Нетерминальный символ (*<цикл for>*)

<expression> **::=** *<A-expr>* | *<L-expr>*
<L-expr> **::= true** | **false** | *<identifier>* | *<comparison>* |
[**not**] *<L-expr>* ('**&&**' | '||') [**not**] *<L-expr>*
<comparison> **::=** *<A-expr>* ('**>**' | '**<**' | '**>=**' | '**=<**' | '**==**') *<A-expr>*
<A-expr> **::=** *<number>* | *<A-expr>* (' **+** ' | '**–**' | '*' | '/') *<A-expr>*
<identifier> **::=** *<letter>* {(*<letter>* | *<digit>*)}*
<number> **::=** {*<digit>* }+

Рисунок3.3 Расширенная БНФ грамматика для определения выражения.

```
<expression> ::= <A-expr> | <L-expr>
<L-expr> ::= [not] (true | false | <identifier> | <comparison>)
            { ('&&' | '||') [not] (true | false | <identifier> | <comparison>)}*
<comparison> ::= <A-expr> ('>' | '<' | '>=' | '=<' | '==' ) <A-expr>
<A-expr> ::= (<number>|<identifier>) {(' + ' | '–' | '*' | '/' ) (<number> | <identifier>)}*
<number> ::= {<digit>}+
<identifier> ::= <letter> {(<letter> | <digit>)}*
<letter> ::= ( 'a' – 'z' | 'A' – 'Z')
<digit> ::= '0' – '9'
```

Рисунок 3.4 РБНФ для ределения выражения

```
<iteration> ::= <for-loop> | <while-loop> | <do-while-loop> | <iterators>
<for-loop> ::= for '('<l-value> '=' <expressions> ';' <expressions>; <expressions> ')' <block>
<while-loop> ::= while '(' <L-expr> ' )' <block>
<do-while-loop> ::= do <block> while '(' <L-expr> ' )'
<iterator> ::= foreach '(' <identifier> in (<identifier> '|' <enumeration>) ')' <block>
<block> ::= '{' {<statement> ';'}* '}' | <statement> ';' | '{' '}'
<statement> ::= <assignment> | <if-then-else> | <iteration>
<assignment> ::= <identifier> '=' <expression>
<enumeration> ::= '{' <entity> { ',' <entity> }* '}' | <identifier>
<entity> ::= <integer> | <float> | <string>
<identifier> ::= <alphabet> {(<alphabet> | <digit>)}*
<string> ::= ""{(<alphabet> | <digit>)}* ""
<integer> ::= [(' + ' | '–')] {<digit>}*
<alphabet> ::= 'A' – 'Z' | 'a' – 'z'
<digit> ::= '0' – '9'
```

Рисунок 3.5 Грамматика для итерационных операторов, написанных на РФБН

(*<идентификатор>*) устанавливается для вычисленного значения выражения (*<выражение>*), следующего за терминальным символом '**;**' следующего за нетерминальным символом *<идентификатор>*, следующего за нетерминальным символом *<опер>*, следующего за нетерминальным символом *<выражение>*, показывая конечное состояние, следующее за разделителем '**;**', следующее за выражением *<выражение>*, чтобы показать размер шага индексной переменной, следующей за правой круглой скобки, следующий за нетерминальным символом *<блок>*, обозначающим блок операторов. Остальные продукционные правила могут быть считаны аналогично, используя знания РФБН, полученные в предыдущих примерах.

3.2.4 Атрибутивные грамматики

Продукционные правила в грамматике могут быть связаны с различными атрибутами для эффективной обработки ошибок, оценки, спецификации ограничений и генерации кода низкого уровня для эффективного выполнения на конкретном типе архитектуры. Различные

архитектуры накладывают различные ограничения на продукционные правила, такие как размер слов, допускаемый размер строк, максимальное количество символов в идентификаторе, и так далее. Кроме того, каждое продукционное правило имеет какое-то значение, связанное с ним, которое используется для генерации низкоуровневого кода из дерева синтаксического разбора. Атрибут продукционного правила отражает ограничение архитектуры, или ограничения, наложенные разработчиком языка, и значение продукционного правила, необходимого для генерации кода низкого уровня из дерева разбора, генерируемого в процессе анализа.

Например, язык может быть скомпилирован для выполнения на 16-битной, 32-битной, или 64-битной машине. По прагматическим причинам и из соображений эффективности, разработчики могут ограничить количество допустимых символов в переменной. Аналогичным образом, на основании размера слова, значение целого числа или числа с плавающей точкой должно быть ограничено максимальным значением, которое зависит от архитектуры. В определениях синтаксиса, эти ограничения (или атрибуты), также являются частью грамматики и определяются вместе с продукционными правилами.

Во время разбора, атрибуты, связанные с продукционным правилом, должны двигаться вверх и вниз по дереву синтаксического разбора, чтобы убедиться, что ограничения выполняются должным образом. Распространение атрибутов также полезно в промежуточной генерации кода и кода для обработки исключений в случае, если нарушаются атрибуты.

Пример 3.7

Таблица 3.2 показывает два продукционных правила и соответствующие введенные продукционные правила. Дескриптор <*целое число*>1 обозначает левостороннее целое число, а дескриптор

<*целое число*>2 обозначает правостороннее целое число. Функция *значение* дает значение целого числа

по основанию 10, функция *длина* обозначает число цифр в целом числе, а функция *размер* дает число знаков в строке.

Атрибуты для правила 1 говорят, что значение целого числа <*целое число*> ограничено в пределах между

от $-\ 2^{31}$ до $+\ 2^{31} - 1$; количество цифр в целом числе на левой

стороне, обозначенное *длиной* (<*целое число*>1), на единицу больше, чем число цифр в целом числе на правой стороне; а значение целого числа <*целое число*>*1* на левой стороне является *10 * значение целого числа <целое число> 2 на правой стороне + значение цифры <цифра> на правой стороне*. В качестве альтернативы, если <*целое число*> определяется как <*цифра*>, то *значение* <*целое число*> будет таким же, как *значение* параметра <*цифра*>.

Правило 2 приписывает наложение ограничения на размер элемента <*идентификатор*> из соображений эффективности. Ограничение заключается в том, что размер идентификатора должен быть меньше или равен 255, а размер левой части идентификатора на единицу больше, чем размер в правой части строки.

Таблица 3.2 Примеры продукционных правил в атрибутивной грамматике

Продукционное правило	**Продукционное правило с атрибутами**
1 <int> ::= <int><digit> | <digit>	<int> ::= <int><digit> **Attributes:** value(<int>) > -2**31; value(<int>) < 2**31 – 1 value(<int>$_1$) = 10 * value(<int>$_2$) + value(<digit>) length(<int>$_1$) = length(<int>$_2$) + 1 <int> ::= <digit> **Attribute:** value(<int>) = value(<digit>)
2 <identifier> ::= <letter>{(<digit>|<letter>)}*	<identifier> ::= <letter> {<digit> |<letter>)}* **Attribute:** size-of(<identifier>) =< 255 size-of(<identifier>) >= 1

3.2.5 Гиперправила и Мета-определения

Правила синтаксиса можно сделать более выразительным, собрав общую картину в продукционных правилах. Используются два дополнительных типа правил: (1) *гиперправила* и (2) *мета-определения. Гиперправила* абстрагируют несколько продукционных правил по общей схеме, а *мета-определения* указывают несколько определений, которые будут замещены в гиперправилах. Подставив мета-определения в гиперправила, можно получить несколько продукционных правил, имеющих аналогичные модели. Например, <*последовательность*> является общим шаблоном по многим продукционным правилам в

грамматике языков программирования и может быть выражена как гиперправило следующим образом:

<последовательность> : *<определение>* ';' *<последовательность>* | ε **-гиперправило**

Мета-определение для определения нетерминального символа *<определение>* приведено ниже. Имена на правой стороне выражения самоочевидны.

<определение> :: *<формальный параметр>* | *<актуальный параметр>*|
<объявление> | *<оператор>* **-мета-определение**

Применяя одно мета-определение за раз в гиперправиле, создаются четыре продукционных правила. Только одно мета-определение применяется за раз, параллельно всем определениям, содержащимся в гиперправиле для генерации продукционных правил; два или более мета-определения не могут быть применены одновременно в том же самом гиперправиле. Четыре правила производства:

<последовательность-формального-параметра> **::=** *<формальный-параметр> ';'*
<последовательность-формального-параметра> | ε
<последовательность-фактического-параметра> **::=***<фактический-параметр>* ';'
<последовательность-фактического-параметра> | ε
<последовательность-объявлений> **::** = *<объявление>* '; '
<последовательность-объявлений> | ε
<последовательноть-операторов> **::=** *<оператор>* ';'
<последовательность-операторов> | ε

В грамматике необходимо отличать продукционные правила от гиперправил и метаправил. В гиперправилах слева и справа используется знак разделения ':', в метаправилах слева и справа используется знак разделения '::', а продукционные правила в БНФ и ее вариациях обладают знаком '::=', отделяющим левую часть от правой части.

3.2.6 Абстрактный синтаксис

В целях понимания свойств абстракций данных и абстракций

управления для класса языков программирования, правила синтаксиса упрощаются за счет использования абстракций данных и управления. Так, например, мы можем абстрагировать конструкцию языка программирования посредством использования *программ, блоков, итерации, операторов выбора, операторов присваивания, команд, выражений, объявления, формальных параметров, фактических параметров, идентификаторов, определений, литералов* и *секвенсоров - операторов перехода*, *выражений, описывающих тип данных* и т. д. Эти абстракции определяются с помощью правил абстрактного синтаксиса в сочетании со значимыми зарезервированными словами в языках программирования. *Правила абстрактного синтаксиса* отличаются от набора продукционных правил для грамматики языка программирования. Некоторые определения более низкого уровня, такие как определения идентификаторов, чисел, целых чисел, цифр, выражений, строк и приоритета оператора- которые являются универсальными для всех языков программирования игнорируются, так как они не добавляют информации к пониманию языков программирования. По той же самой причине игнорируются литералы более низкого уровня, разделители и пробелы. Абстрактный синтаксис лаконичен и объясняет конструкции посредством имеющихся знаний программистов об абстракциях. В правилах абстрактного синтаксиса имеются свойственные им неопределенности, возникшие в результате пропуска особенностей низкого уровня. Тем не менее, он лаконичен, связан с абстракциями управления и данных в языке программирования в форме нетерминальных символов и содержит основы языка программирования.

Пример 3.8

Правила абстрактного синтаксиса для абстракций управления l-значений, объявлений, выражений и команд могут быть определены для основного класса императивных языков программирования, как показано на Рисунке 3.6.

Правила абстрактного синтаксиса показывают, что абстракция l-значения может быть идентификатором, конкретным полем структуры или переменной с индексом (записывается *<l-значение>* ‘[‘*<выражение>*’]’). Абстрактное выражение может быть литералом, идентификатором, l-значением, выражением в скобках, двумя выражениями, соединенными бинарным оператором или унарным

оператором, за которым следует выражение.

Абстракция команды в данном языке программирования может быть блоком в фигурных скобках, оператором присваивания, последовательностью команд, условным оператором

```
<l-value> ::= <identifier> | <identifier>.<l-value> | <l-value>' ['<expression>']'

<declarations> ::= variable <identifier> <type-expression> |
                   <type-expression>[<numeral>] |
                   structure {<type-expression>} <identifier> |
                   void <identifier> ( <formal-parameters> ) |
                   <identifier> function <identifier> ( <formal-parameters> )

<expressions> ::= <literal> | <identifier> | <l-value> | (<expressions> | <op> <expressions> |
                  <expressions> <op> <expressions>
<actual-parameters> ::= <identifier> ',' <actual-parameters>

<formal-parameters> ::= <identifier> ',' <identifier-sequence> ';' <formal-parameters> | ∈

<commands> ::= { <commands> } | <l-value> '=' <expressions> | <command> ';' <commands> |
                   if <expressions> then <commands> else <commands> |
                   if <expressions> then <commands> |
                   while '(' <expressions> ')' <commands> |
                   do <commands> while '(' <expressions> ')' |
                   for '(<l-value>'=' <expressions> ';' <expressions>';' <expressions> ')' <commands> |
                   <identifier> '(<formal-parameters> ')'

<sequencer> ::= goto <numeral>
<program> ::= main <identifier>';' <declarations> ';' <commands>
```

Рисунок 3.6 Пример правил абстрактного синтаксиса.

циклом WHILE, циклом DO-WHILE, циклом FOR или вызовом процедуры. Кроме того, она сообщает зарезервированные слова {**if, then, else, do, while, '{,' and '},' for**} и другие, являющиеся частью языковых конструкций. Тем не менее, другие особенности низкого уровня были убраны. Например, определение оператора в определении правил абстрактного синтаксиса для

<*выражение*> не различается между различными приоритетами операторов и разными типами выражений: логический по сравнению с арифметическим.

3.3 СИНТАКСИЧЕСКИЕ ДИАГРАММЫ

Текст подходит для обработки грамматики на компьютере. Тем не менее, люди лучше воспринимают визуализированное и простое для понимания графическое изображение. Разработчики языка используют графические модели продукционных правил, чтобы программисты могли визуализировать, понять и использовать синтаксис языков

программирования. Такие графические модели называются *синтаксическими диаграммами*.

Понимание взаимосвязи между синтаксической грамматикой и синтаксической диаграммой необычайно важно, потому что разработчики языка должны

(1) составлять синтаксические диаграммы, которые будут просты для понимания программистами и другими разработчиками языка и (2) воспроизводить текстовую форму для разрабатывающихся синтаксических анализаторов и генераторов кода. Программисты также должны преобразовывать синтаксические сведения, полученные при изучении синтаксических диаграмм в текстовую форму при разработке программы.

Формально, *синтаксическая диаграмма* - это *направленный циклический граф* с терминальными и нетерминальными символами, представленными узлами, и взаимосвязями между терминальными и нетерминальными символами в правой части продукционного правила, представленными дугами. Циклы моделируют определения хвостовой рекурсии (или повторения в РБНФ). Левая часть синтаксической диаграммы описывает нетерминальные символы в продукционном правиле, расположенные слева, остальная часть графа является моделью правой части продукционного правила.

Синтаксические диаграммы включают три основных компонента: *нетерминальные символы*, *терминальные символы* и *направленные дуги*. Для удобства нетерминальные символы заключаются в овалы, таким образом их четко можно отличить от терминальных символов. Стрелка показывает направление движения по диаграмме. Крайний слева символ показывает нетерминальный символ в левой части продукционного правила, который необходимо определить. Различные преобразования при переходе от синтаксических правил к синтаксическим диаграммам представлены на Рисунках 3.7 и 3.8.

Повторные определения представлены в виде нескольких направлений движения в синтаксической диаграмме. Нулевой символ представлен в виде направленной дуги между источником и блоком назначения, в котором не содержится символ. Объединение нескольких символов в правой части продукционного правила представлено в виде участка, соединяющего несколько узлов. Определения хвостовой рекурсии представлены в виде контура с обратной связью (цикла), для того чтобы показать неоднократность вхождения определения. Контур с обратной связью с определением на выходящей дуге описывает

одно или несколько вхождений в РБНФ. У узла назначения две опции: использование идущей обратно дуги для повторного прохождения цикла или выход. Интересный маршрут заключается в моделировании нулевого или большего количества вхождений символа. Он представляет собой контур с обратной связью с изменениями, происходящими в результате того, что на дуге обратного направления появляются символы, а на дуге, направленной вперед, символов нет.

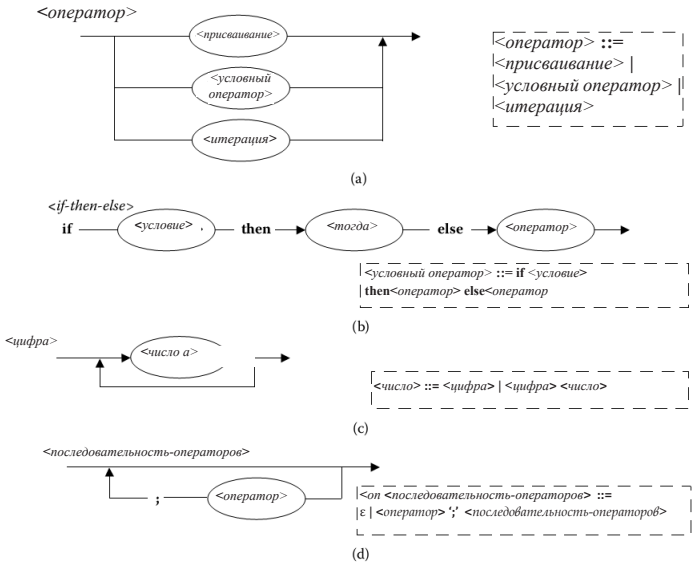
Для боле ясного представления информации и восприятия, несколько продукционных правил объединяются в одну синтаксическую диаграмму. Объединение используется для отражения абстрактных элементов в языках программирования, таких как *идентификаторы; переменные; целые числа; десятичные числа; цифры; арифметические выражения; логические выражения; фактические параметры; формальные параметры;* различные типы операторов, *таких как условный оператор, цикл WHILE, цикл DO-WHILE, итераторы и операторы выбора; блок операторов; объявления типа данных; программа;* и т.д.

Пример 3.9

На Рисунке 3.7 представлены синтаксическая диаграмма для повторных определений (Рисунок 3.7a), синтаксическая диаграмма взаимосвязи символов (Рисунок 3.7b), синтаксическая диаграмма определений хвостовой рекурсии, показывающей одно или несколько вхождений символа(ов) (Рисунок 3.7c), и синтаксическая диаграмма для нулевого или большего количества вхождений символа(ов) (Рисунок 3.7d).

Как показано на Рисунке 3.7a, разветвление на несколько маршрутов между двумя узлами, начинающееся с одной стороны и объединяющееся обратно с другой стороны, представляет собой повторные определения. Синтаксическая диаграмма для нетерминального символа <*оператор*> имеет несколько ветвей: (1) первая ветвь содержит нетерминальный символ <*присваивание*>; (2) вторая ветвь содержит нетерминальный символ <*условный оператор*>; и (3) третья ветвь содержит нетерминальный символ <*итерация*>. Она эквивалентна синтаксическому правилу вида

<*оператор*> **::=** <*присваивание*> | <*условный оператор*> | <*итерация*>



На рисунке 3.7b представлено синтаксическое правило <условный оператор> ::= if <условие> then <оператор> else

<оператор>. Правая часть правила представляет собой соединение терминальных символов {if, then, else} и нетерминальных символов {<условие>, <оператор>} в определенном порядке. Синтаксическая диаграмма сохраняет порядок соединения в синтаксическом правиле.

На Рисунке 3.7c представлена синтаксическая диаграмма для продукционного правила <число> ::= <цифра>

| <цифра><число>. Определение имеет базовый случай<число> ::= <цифра> и определение хвостовой рекурсии<число> ::= <цифра> <число>. Это отображается в виде одного или нескольких вхождений <цифра>, потому что определение хвостовой рекурсии можно использовать для добавления еще одной

<цифры>, а определение нетерминального символа<число> всегда можно завершить применением базового случая, который добавляет еще одну <цифру> перед завершением расширения. Определение представляет одно или несколько вхождений <цифры>. РБНФ вариант синтаксического правила записывается как <число> ::= {<цифра>}$^{+}$.

Соответствующая синтаксическая диаграмма представляет собой контур с обратной связью, в котором дуга в прямом направлении содержит нетерминальный символ

<цифра>, а от узла назначения идет дуга в обратном направлении к узлу-источнику. Движение в прямом направлении можно проходить несколько раз. Каждый раз, когда повторяется движение в прямом направлении, прибавляется дополнительная <цифра> для формирования последовательности <цифра>.

На Рисунке 3.7d представлена синтаксическая диаграмма, моделирующая синтаксическое правило для нулевого или большего количества вхождений операторов. Соответствующее продукционное правило в БНФ состоит из базовой части и части хвостовой рекурсии. Базовая состоит из пустого символа ε, а часть хвостовой рекурсии включает нетерминальный символ <оператор>, сопровождаемый символом ';' как показано ниже

<последовательноть-операторов> ::= <оператор> ';'
<последовательность-операторов> | ε

В РБНФ нулевое или большее количество вхождений символов записываются как символы, помещенные в круглые скобки, и ставится знак "*" сразу после правой круглой скобки, как показано ниже:

<последовательность-операторов> ::= {<оператор>';'}*

Нулевое или большее количество вхождений символов изображается в синтаксической диаграмме в виде контура с обратной связью, так что дуга в прямом направлении изображается в виде прямой стрелки без символов, а на дуге обратного направления присутствуют символы. На Рисунке 3.7d на дуге обратного направления находится нетерминальный символ

<предложение> , за которым следует терминальный символ ';.'

3.3.1 Перевод синтаксических правил в синтаксические диаграммы

Синтаксические правила переводятся в соответствующие синтаксические диаграммы согласно следующим правилам:

1. Терминальные и нетерминальные символы в синтаксических диаграммах четко различаются.

2. *Продукционное правило с повторными определениями* в БНФ или с *группой*в РБНФ можно представить как параллельные ветви между двумя узлами, и выглядит это как многополосная дорога.

3. *Объединение* нескольких символов с правой стороны изображается как несколько символов на одной ветви.

4. Определение хвостовой рекурсии изображается как контур с обратной связью, в котором правый край соединен с началом определения.

5. Пустой символ, обозначается ε, изображается между двумя узлами в виде прямой линии со стрелкой.

6. Необязательное определение изображается в виде двух ветвей, причем одна ветвь представляет собой прямую линию со стрелкой от одного узла к другому, а вторая ветвь содержит определение.

На Рисунке 3.8 изображена синтаксическая диаграмма для соответствующих компонентов синтаксических правил в БНФ и РБНФ. Эллипс обозначает символ (терминальный или нетерминальный), представленный в форме текста.

На первой строке представлена синтаксическая диаграмма, соответствующая продукционному правилу, в котором три символа находятся в правой части. Три дуги, соединяющие эллипсы, для удобства были объединены в одну. На второй строке представлено продукционное правило, в котором одновременно содержатся четыре разных определения:

1. Первое определение, расположенное на верхней дуге, содержит два символа в правой части.

2. Второе и третье определения содержат по одному символу в правой части.

3. Четвертое определение, расположенное на нижней дуге, содержит два символа в правой части.

На третьей строке представлено определение хвостовой

рекурсии одного или нескольких вхождений символа в правой части продукционного правила. Одно или несколько вхождений равнозначны

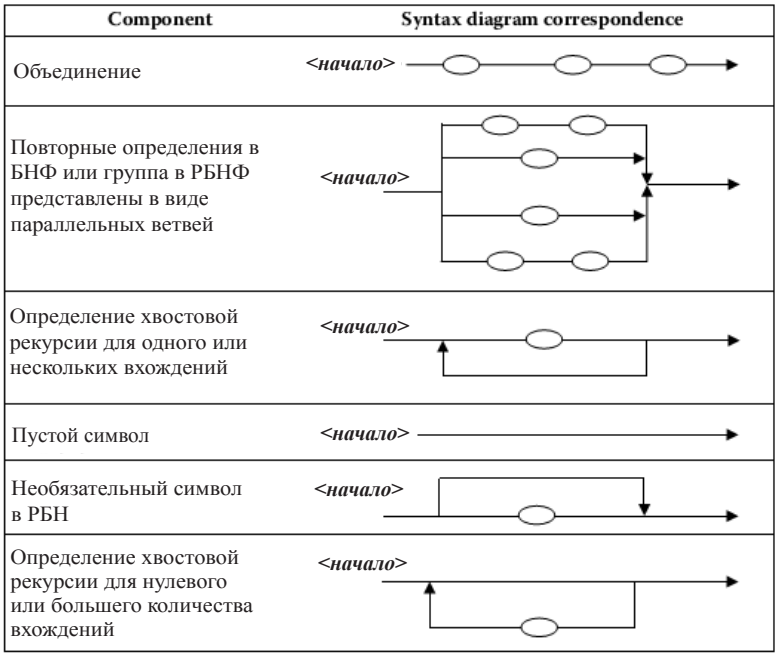


РИСУНОК 3.8 Соответствие между синтаксическими диаграммами и синтаксическими правилами.

$\{$<*символ*>$\}^{+}$ в РБНФ и определению хвостовой рекурсии <*определение*> **::=** <*символ*>

<*определение*> | <*символ*> в БНФ, где <*символ*> может быть выражен сочетанием терминального и нетерминального символа. На четвертой строке представлена синтаксическая диаграмма для пустого символа ‘ε.’ Так как в этом случае символ отсутствует, то соответствующая синтаксическая диаграмма представляет собой прямую стрелку.

На пятой строке представлена синтаксическая диаграмма необязательного символа. Необязательный символ, записанный как [<*символ*>] в РБНФ соответствует группе (<*символ*> | **ε**), состоящей из двух маршрутов прямого направления: на одном маршруте стоит <*символ*>, как встроенный узел, а на другом маршруте нет встроенного узла. На шестой строке представлено нулевое или большее количество

вхождений символа. В синтаксической диаграмме нулевого или большего количества вхождений также есть обратная дуга, тождественно равная одному или нескольким вхождениям. Тем не менее, в синтаксической диаграмме для нулевого или большего количества вхождений символ присутствует не на прямом, а на обратном маршруте. Определение нулевого или большего числа вхождений <*символ*> записывается как {<*символ*>} * в РБНФ и как определение хвостовой рекурсии

<*определение*> **::=** <*символ*><*определение*> | **ε** в БНФ, где <*символ*> может быть терминальным или нетерминальным символом.

Для перевода из текста в синтаксическую диаграмму грамматику группируют в значимые функциональные единицы, обладающие некоторым значением на уровне абстракции данных или управления. Грамматику разделяют на группы, состоящие из подгрупп продукционных правил, в которых каждая подгруппа соответствует при создании программы функциональной единице, такой как *переменные, формальные параметры, блок* и т.д. Затем правила высокого уровня итеративно преобразуются в синтаксические диаграммы, а нетерминальные символы внутри преобразуемых синтаксических диаграмм расширяются до других синтаксических диаграмм, пока добавление синтаксических диаграмм более низкого уровня не облегчит дальнейшее понимание конкретного функционального блока.

Пример 3.10

В определении <*идентификатор*> присутствуют три продукционных правила, использующие группу из трех нетерминальных символов: {<*идентификатор*>, <*буква*>, <*цифра*>}. При этом <*буква*> и<*цифра*> используются для определения нетерминального символа <*идентификатор*>. Таким образом, три продукционных правила объединяются в одну синтаксическую диаграмму.

```
<identifier>, <letter> and <digit> as shown below:
<identifier> ::= <letter> {(<letter> | <digit>)}*
<letter> ::= 'A' - 'Z' | 'a' - 'z'
<digit> ::= '0' - '9'
```

Сначала строится синтаксическая диаграмма для продукционного правила, определяющего нетерминальный символ <*идентификатор*>. Продукционное правило, определяющее <*идентификатор*> выбирают первым, так как определение

<идентификатора> включает в себя определение других нетерминальных символов, находящихся в правой части, и эти нетерминальные символы можно раскрыть позже. Нетерминальные символы*<буква>* и*<цифра>* раскрывают для уточнения синтаксической диаграммы. Продукционные правила конвертируются в синтаксическую диаграмму благодаря соответствию между классами продукционных правил. Итоговая синтаксическая диаграмма представлена на Рисунке 3.9.

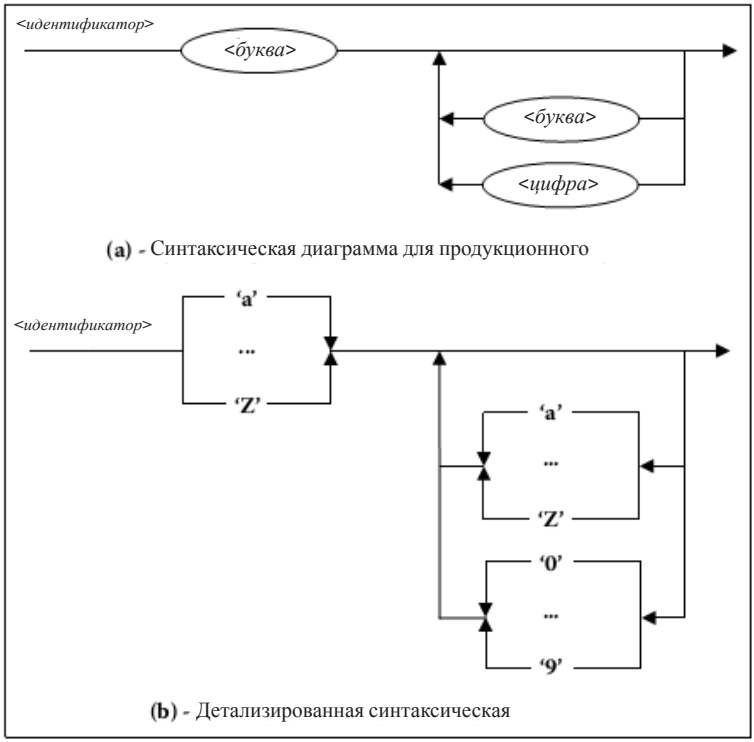


РИСУНОК 3.9 Построение синтаксической диаграммы с использованием нескольких продукционных правил.

3.3.2 Перевод синтаксических диаграмм в синтаксические правила

Синтаксические диаграммы необходимо преобразовывать в форму текста для их использования в генераторе правил в синтаксическом

анализаторе. Таким образом, разработчик языка или программист сможет преобразовать синтаксическую диаграмму в соответствующий набор синтаксических правил.

Перевод синтаксической диаграммы в продукционное правило представляет собой процесс , обратный созданию синтаксической диаграммы из продукционных правил. Те части синтаксической диаграммы, которые встраиваются в синтаксическую диаграмму и которые определяются только через терминальные символы и уже определенные нетерминальные символы, переводятся в продукционные правила с помощью соответствий между продукционными правилами и синтаксическими диаграммами, которые были представлены на Рисунке 3.8. Новые значимые нетерминальные символы создают для встроенных синтаксических диаграмм.

После создания продукционного правила, соответствующие части синтаксической диаграммы заменяют заново определенным нетерминальным символом, и процесс повторяется. Процесс останавливается, когда синтаксическая диаграмма полностью превращается в один нетерминальный символ.

Пример 3.11

Для создания набора правил для синтаксической диаграммы, представленной на Рисунке 3.7, большинство встроенных частей - несколько параллельных маршрутов , определяющих группу (‘**0**’ | ‘**1**’ | ... | ‘**9**’) и несколько параллельных маршрутов, определяющих группу *(‘***a***’* | ... | *‘***z***’)* - преобразуются в <*буква*> **::=** *‘***a***’* | ... | *‘***z***’*и<*цифра*> **::=** *‘***0***’* | ... | *‘***9***’*. Нетерминальные символы <*буква*> и <*цифра*> теперь подставляют на место нескольких параллельных маршрутов, и синтаксическая диаграмма сокращается до диаграммы, представленной на Рисунке 3.7а. На Рисунке 3.7а, представлено объединение нетерминального символа <*буква*> и определения хвостовой рекурсии, показывающей нулевое или большее количество вхождений альтернативных символов (<*буква*> | <*цифра*>). Таким образом, новое продукционное правило имеет вид <*идентификатор*> **::=** <*буква*> {(<*буква*> | <*цифра*>)}*.

3.4 ПРОВЕРКА СТРУКТУРЫ ПРЕДЛОЖЕНИЯ

Прооцесс преобразования программы начинается с проверки структуры предложения. Проверка структуры предложения состоит из двух этапов: (1) преобразование символов предложения во внутренний

формат с помощью «токенов» и (2) проверка этого внутреннего формата предложений с помощью грамматических правил языка. Первый этап называется *лексический анализ*, а второй этап называется *синтаксический анализ*. Результатом второго этапа является *дерево разбора*. *Дерево разбора* - это дерево, сформированное повторяющимся сопоставлением правой части грамматических правил и замещением сопоставленной части промежуточной формы (или предложения) соответствующей левой частью нетерминального символа сопоставляющегося продукционного правила. Такая замена создает новую промежуточную форму, в которой часть, сопоставленная с правой частью продукционного правила, заменяется на нетерминальный символ левой части продукционного правила. Процесс повторяется до тех пор, пока не будет достигнут начальный символ. Последовательность полученных промежуточных форм образует дерево разбора. Дерево разбора является исходной информацией для следующего уровня генерации кода. И лексический, и грамматический анализ автоматизированы, при этом вспомогательные программы для создания автоматизированного грамматического анализатора с использованием *генератора правил для синтаксического анализатора* разрабатывались долгое время. Входящим звеном в генератор правил для синтаксического анализатора служит грамматика языка. Автоматизированный грамматический анализатор использует методы «сверху-вниз» или «снизу-вверх» в сочетании с предварительным просмотром символов, чтобы однозначно определить продукционное правило с целью сокращения промежуточной формы. Автоматизированные грамматические анализаторы рассматриваются более подробно

в Разделе 3.4.5.

В этом разделе рассматриваются основы лексического анализа и грамматического анализа. Мы используем наши интеллектуальные способности для установления подходящего правила, которое в большей степени сократит промежуточные формы до соответствующих нетерминальных символов, то есть, таким образом мы можем сформировать оптимальное дерево разбора. Изучение генераторов правил для синтаксического анализатора и других методов автоматизации синтаксического анализа входит в курс изучения компиляторов.

3.4.1 Лексический анализ

Лексический анализ - это первый этап трансляции программы (компиляции), на котором распознаются и преобразуются в токены

зарезервированные слова, идентификаторы и литералы. Литералы - это постоянные величины в языке программирования, поэтому они не могут быть изменены. Например, *10* и *4,7* являются литералами. Наименьшая синтаксическая единица, из которой образовано зарезервированное слово, идентификатор или литерал, называется *лексемой*. *Лексема* представляет собой последовательность символов, которая формирует в языке программирования синтаксическую единицу самого низкого уровня. При лексическом анализе пробелы удаляются; создаются токены; и поток токенов поступает в синтаксический анализатор для проверки предложения. Например, символ оператора присваивания '=' переходит в токен *присвоить*; символ '<' переходит в токен *меньше-чем*.

Конечный автомат используется для распознавания слов, идентификаторов и литералов и для создания токенов. Конечный автомат определяется соответствующей регулярной грамматикой, которая затем разделяется на более мелкие взаимосвязанные конечные автоматы, вырабатывающие токены, как показано на

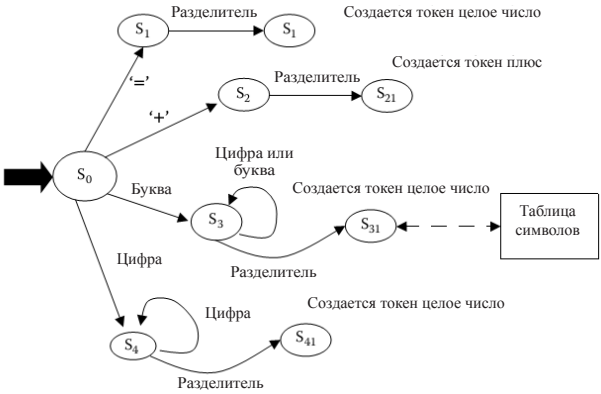


РИСУНОК 3.10 Упрощенная схема конечного автомата для лексического анализа.

Рисунке 3.10. Данные, входящие в конечный автомат, представляют собой последовательность символов в программе, а выходные данные - токенизированная версия программы.

Процесс распознавания лексемы начинается с исходного состояния S_0, при этом конечный автомат пропускает различные состояния в процессе распознавания лексемы. После определения разделителя или предварительного просмотра символа, в ходе которого устраняется

неоднозначность преобразования в следующее состояние, создается соответсвующий токен. После создания токена, конечный автомат возвращается в исходное состояние S_0. Возможными разделителями могут быть пробелы, символы окончания предложения, например, точка с запятой, окончание блока предложений, например, фигурные скобки, символ конца строки в некоторых языках или символ конца строки. Определение разделителей зависит от языка. После создания токена пара (*лексема, созданный токен*) архивируется в таблицу символов. В следующий раз, когда в программе будет определяться такая же лексема, при просмотре таблицы символов будет создан такой же токен.

Пример 3.12

Конечный автомат на Рисунке 3.10 может вводить идентификаторы, целые числа без знака и зарезервированные слова ‘=’ и ‘+’. Он состоит из четырех подчастей, каждая из которых создает соответствующие токены для зарезервированных слов‘=’ и ‘+’, идентификатора и внутреннее числовое представление для числа. Исходное состояние конечного автомата - S_0. При обработке зарезервированного слова ‘=’ конечный автомат переходит в состояние S_1, при обработке зарезервированного слова ‘+’ конечный автомат переходит в состояние S_2, при обработке буквы конечный автомат переходит в состояние S_3, при обработке любой цифры конечный автомат переходит в состояние S_4. Конечный автомат выполняет цикл в состоянии S_3, если следующим символом является буква или цифра, пока не будет найден разделитель, который преобразует идентификатор в токен, и переходит обратно в состояние S_0. Если в качестве входного данного поступает цифра или буква, то конечный автомат выполняет цикл в состоянии S_4 до тех пор, пока не найдется разделитель, который создаст токен, являющийся внутренним представлением целого числа без знака, состоящего из последовательности цифр. Затем переходит обратно в состояние S_0.

Давайте рассмотрим предложение *sum* = *sum* + 10. Символ *sum* является идентификатором, символы ‘=’ и ‘+’ - зарезервированными словами, символ 10 - число, символ ‘;’ - разделитель, показывающий конец предложения . Символ пробела выступает как разделитель идентификатора и зарезервированных слов. Во входной строке находятся десять символов “*sum* = *sum* + 10”, включая пробелы.

Лексический анализатор за один раз обрабатывает один входной символ. Изначально, конечный автомат находится в состоянии S_0.

Первая буква ‘s’. Конечная автомат переходит в состояние S_3, которое распознает идентификаторы. Следующие два символа - буквы ‘u’ и ‘m’. При состоянии S_3 автомат выполняет цикл. Следующий символ - пробел, выполняющий роль разделителя идентификаторов. Конечный автомат переходит в новое состояние S_{31}, при этом идентификатор *sum* интернализируется. Конечный автомат обращается к таблице и берет из нее значение токена ‘_1’. При последующих вхождениях идентификатора конечный автомат будет выполнять поиск в таблице поиска и брать из нее интернализованный токен ‘_1’. После создания токена ‘_1’ конечный автомат возвращается в исходное состояние S_0. Следующий символ - зарезервированное слово
=’. Конечный автомат переходит в состояние S_1. Также предварительно просматривается последующий символ - пробел, который выполняет роль разделителя. Конечный автомат переходит в следующее состояние S_{11}, и создает соответствующий токен *присвоено*. После создания токена *присвоено*, конечный автомат возвращается в исходное состояние S_0. Со следующей буквой ‘s’ конечный автомат переходит в состояние S_3, и процесс повторяется с единственный отличием:

в поисковой таблице уже содержится запись об идентификаторе *sum,* , поэтому снова создается токен ‘_1’. После создания токена ‘_1’ конечный автомат возвращается в исходное состояние S_0. После просмотра следующего символа ‘+’ конечный автомат переходит в состояние S_2. После просмотра разделительного пробела конечный автомат переходит в состояние S_{21}, создает токен *плюс* и возвращается в исходное состояние S_0. Следующий символ - цифра 1. Автомат переходит в состояние S_4. Следующий

символ - цифра 0. Конечный автомат циклически возвращается в состояние S_4. После просмотра разделителя ‘;’ конечный автомат выводит токен - интернализованный вариант целого числа 10 без знака и возвращается в исходное состояние S_0. Выходной информацией из лексического анизатора будет последовательность ‘_1’ *присвоено* ‘_1’ *плюс* 10. Такая токенизированная последовательность будет служить входными данными для синтаксического анализа.

Во многих случаях невозможно определить лексему только просмотрев символ, из-за неоднозначностей при переходе в более чем одно состояние. Тогда просматриваются несколько символов, чтобы устранить неоднозначность. Например, три различных символа ‘=’ и ‘==’ и ‘=<’ с тремя различными токенами: ‘=’ получат токен *присвоено*, ‘==’ получит токен *равно*, а последовательность символов ‘=<’ получит

токен *меньше_или_равно*. Если смотреть только на первый символ '=', то может возникнуть неоднозначность. Как только мы рассмотрим следующий знак, то неоднозначность исчезнет: при наличии пробела будет создан токен *присвоено*, если в качестве последующего символа стоит '=', то будет создан токен *равно*, а если последующий символ '<', то будет создан токен *меньше_или_равно*

Для лексического анализатора конечный автомат моделируется в виде двумерной таблицы перехода с размерами *M* строк и *X N* столбцов, где *M* - это количество состояний в конечном автомате, а *N* - количество возможных входных символов. Каждая ячейка таблицы содержит состояние (или набор состояний), в которое перейдет конечный автомат при рассмотрении следующего входного символа. В случае неоднозначности в ячейке может находится несколько состояний. Решение находится с помощью стека. В нем хранятся пары формы (*позиция предварительно-просмотренного-символа*, *набор остальных состояний, которые необходимо рассмотреть*), которые перемещаются вперед во входном потоке вместе с последующим предварительно просматриваемым символом. Если следующий предварительно просматриваемый символ образует уникальную лексему, то стек повторно инициализируется, и образуется токен. Иначе, стек выталкивается, и изучаются остальные возможные состояния.

3.4.2 Синтаксический анализ

Синтаксический анализатор принимает токены, сформированные в предложение, и грамматику языка в качестве входных данных и создает однозначное дерево разбора. В процессе синтаксического анализа циклически применяется одно из правил грамматики для подпоследовательности *сокращенно-промежуточной-формы* предложения, синтаксический анализ которого необходимо провести. *Сокращенно-промежуточная-форма* состоит из сочетания терминальных и нетерминальных символов. Следующее правило выбирается сопоставлением *сокращенно-промежуточной-формы* с правой частью продукционного правила и замещением подобранной части нетреминальным символом из левой части вплоть до получения первоначального символа. Если цикличное применение продукционного правила не приводит к получению первоначального символа или нет последовательности сокращенной-промежуточной-формы, к которой можно применить продукционное правило, то провести синтаксический анализ нельзя, и это приводит к получению ошибки. Упрощенная схема

синтаксического анализа представлена в виде алгоритма на Рисунке 3.11.

Входными данными для алгоритма являются набор продукционных правил {p_1, **…** p_n}, предложение S, представляющее собой последовательность символов s_0, **…**, s_m и начальный символ *root*. Выходными данными алгоритма является дерево разбора T. Сначала сокращенная-форма представляет собой исходную последовательность S. Пока сокращенная-форма не станет эквивалентна начальному символу и не произойдет ошибки синтаксического анализа, будет выполняться итерация цикла WHILE. Внутри цикла итерации, продукционным правилом p_i в форме<*не*

терминальный-символ> **::=** s_i **…** s_j проверяется, содержится ли в текущей сокращенной форме подпоследовательность

si … sj. После сопоставления подпоследовательность *si* **…** *sj* замещается на левую часть

(<*продукционное-правило* p_k>), и изменяемая *сокращенная-форма* обновляется. Дерево T также обновляется в соответствии с добавлением новых дуг в форму (s_i **…** s_j → *левая-часть*(p_k)), и процесс повторяется до тех пор, пока не будет найден корневой символ или не обнаружится ошибка анализа.

Ошибка анализа логической переменной возникает в том случае, если невозможно подобрать соответствующее правило.

Алгоритм синтаксического анализа предложения типа снизу-вверх

Вход:
1. Набор продукционных правил грамматики **R** = {p_i, … p_n};
2. Предложение в виде последовательности S = s_0, …, s_m;
3. Начальный символ *корень*;

Выход: Дерево синтаксического анализа T;

```
{   reduced-form = S;
    parsing-error = false;
    T = null-tree;
    while ((reduced-form ≠ root ) && not(parsing-error) )
            { If there exists a subsequence s_i .. s_j in reduced form, such that
              s_i... s_j == right-hand-side(p_k ∈ R) where 1 =< k =< n {
              nonterminal = left-hand-side(p_k);
              reduced-form = substitute(reduced-form, s_i... s_j, nonterminal);
              T = T + edge(s_i ..s_j → left-hand-side(p_k));
              }
              else parsing-error = true;
            }
    If not(parsing-error) return(T); else print('parsing-error');
}
```

РИСУНОК 3.11 Упрощенная схема синтаксического анализа предложения типа снизу-вверх. Процесс нахождения подходящей

подпоследовательности, которая соответствует продукционному правилу, - сложная задача, которая решается за счет применения различных автоматизированных методов синтаксического анализа, таких как LL(K)- и LR-анализаторы. Автоматизированные анализаторы кратко описаны в подразделе 3.4.5.

Пример 3.13

Давайте проанализируем предложение “x + 3 * 4,” используя грамматику, представленную на Рисунке 3.2. Предложение содержит пять символов: ‘x’, ‘+’, ‘3’, ‘*’, и ‘4’. Эти символы будут преобразованы лексическим анализатором в соответствующие токены. Тем не менее, для удобства, вместо токенов мы будем использовать символы в их первоначальной форме. Любой символ или подпоследовательность из 5 символов может претендовать на соответствие с правой частью продукционного правила. Используя грамматику, представленную на Рисунке 3.3, получаем два альтернативных дерева разбора, показанных на Рисунках 3.12 и 3.13. *Грамматика, дающая больше одного дерева разбора для одного предложения*, *называется неоднозначной*, *и* ее *необходимо избегать*.

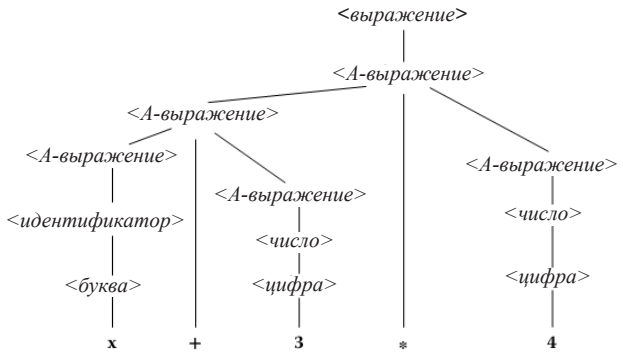


РИСУНОК 3.12 Неправильное дерево разбора, полученное в связи с неоднозначностью в грамматике.

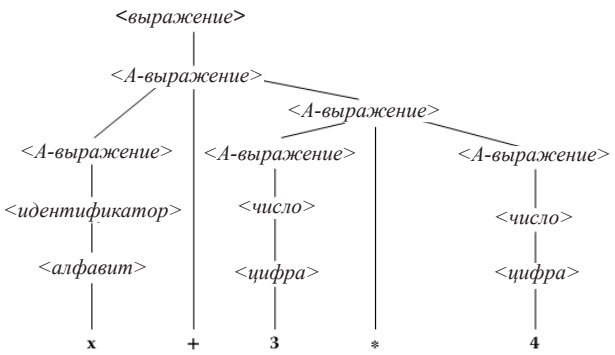


РИСУНОК 3.13 Правильное дерево разбора для выражения x + 3 * 4.

3.4.3 Устранение грамматических неоднозначностей

Неоднозначная грамматика нарушает один из основополагающих принципов языка программирования, который заключается в том, что предложение должно обладать индивидуальным значением. Два различных дерева разбора для одного предложения могут быть преобразованы в два различных набора указаний нижнего уровня, что приведет к двум различным вычислениям. Существует два основных класса вычислений, которыми вызваны грамматические неоднозначности: (1) в продукционном правиле не учитывается приоритет операторов, и (2) сопоставление конструкций во вложенных структурах, в которых уровень вложения разделен зарезервированными словами неоднозначно.

Пример 3.14

Грамматика для определения *<выражение>* на Рисунке 3.2 посредством 3.5 неоднозначна из-за использования нескольких определений арифметических выражений и логических выражений при группировке операторов в одном продукционном правиле. В арифметических правилах бинарные операторы '*' и '/' имеют более высокий приоритет чем бинарные операторы '+' и '−'. Аналогично, в логических выражениях, унарный оператор 'not' имеет более высокий приоритет чем '&&' (логическое И), а '&&' (логическое И) имеет более высокий уровень приоритет чем '||' (логическое ИЛИ).

Один из способов справиться с неоднозначностями в выражениях – это четко разделить подвыражения, используя круглый скобки. Тем не менее,

очередность приоритета может быть закодирована в грамматических правилах посредством разделения продукционного правила на несколько продукционных правил. Продукционное правило со множеством определений делится на продукционные правила при введении новых нетерминальных символов, которые последовательно определяются в среде других нетерминальных символов с увеличивающимся приоритетом операторов, и последнее продукционное правило использует операторы с наивысшим приоритетом. Сначала анализируются операторы с более высоким приоритетом, так как правила, содержащие операторы более высокого приоритета, применяются в первую очередь при синтаксическом анализе типа снизу-вверх. Соответствующая неоднозначная грамматика выражений представлена на Рисунке 3.14.

Продукционное правило для арифметического выражения *<A-выражение>* было разделено на три продукционных правила с использованием двух дополнительных нетерминальных символов *<выражение-многоопред.>* и

<A-условие>. Аналогично, продукционное правило для логического выражения *<L-выражение>*

<выражение> **::=** *<A-выражение>* | *<L-выражение>* (1)
<A-выражение> **::=** *<A-выражение>* (‘ + ’ | ‘-‘)*<выражение-многоопред.>* | *<выражение-многоопред.>* (2)
<выражение-многоопред.> **::=** *<выражение-многоопред.>* (‘*’ | ‘/’) *<A-условие>* | *<A-условие>* (3)
<A-условие> **::=** ‘**(**‘ *<A-выражение>* ‘**)**’ | *<идентификатор>* | *<число>* (4)
<L-выражение> *::=* *<L-выражение>* ‘**||**’*<выражение-И>* (5)
<выражение-И> **::=** *<выражение-И>* ‘**&&**’ *<L-условие>* (6)
<L-условие> *::=* [**not**] (‘**(**‘ *<сравнить>* ‘**)**’ | ‘(‘ *<L-выражение>* ’)’ | *<идентификатор>* | **true** | **false** (7)
<сравнить> *::=* *<A-выражение>* (‘>’ | ‘<’ | ‘>=’ | ‘=<’ | ‘==’) *<A-выражение>* (8)
<идентификатор> *::=* *<буква>*{(*<буква>*|*<цифра>*)}* (9)
<число> **::=** [(‘ + ’|’−‘)] {*<цифра>*} + (10)
<буква> **::=** ‘**a**’|‘**b**’|...|‘**z**’|‘**A**’|‘**B**’|...|‘**Z**’ (11)
<цифра> **::=** ‘**0**’|‘**1**’|...|‘**9**’ (12)

РИСУНОК 3.14 Неоднозначная грамматика выражений.

было разделено на три продукционных правила с использованием двух дополнительных нетерминальных символов:

<выражение-И> и *<L-условие >*. Нетерминальный символ *<A-выражение>* был определен посредством *<выражение-многоопред.>*. Нетерминальный символ *<выражение-многоопред.>* был определен относительно *<A-условие>*. Нетерминальный символ *<A-условие>* определен относительно (*<A-выражение>*), или *<идентификатор>*, или *<число>*. Нетерминальный символ *<L-выражение>* определяется

нетерминальным символом *<выражение-И>*. Нетерминальный символ*<выражение-И>* определяется нетерминальным символом *<L-условие>*. Нетерминальный символ

<L-условие> определяется относительно необязательного терминального символа '**not**',за которым следует группа логических выражений в скобках, или сравнение в фигурных скобках, или нетерминальный символ*<идентификатор>*, или терминальный символ **true** , или терминальный символ **false**.

Пример 3.15

Давайтепроанализируемпредложение"x+3*4",используяграмматику, представленную на Рисунке 3.14. Литералы '3' и '4' сокращаются до нетерминального символа *<A-условие>* при помощи последовательности правил: правило 12, правило 10, правило 9 и правило 4. Сокращенная форма *"<A-условие>* − *<A-условие>*" сокращается до нетерминального символа *<выражение-многоопред.>* с помощью правила 3, и новая сокращенная форма имеет вид "x + *<выражение-многоопред.>."* Символ x сокращается до нетерминального символа*<идентификатор>* с помощью последовательности правил: правило 11 и правило 9, и новая промежуточная форма имеет вид "*<идентификатор>* + *<выражение-многоопред.>*." Нетерминальный символ *<идентификатор>* в дальнейшем будет сокращен до нетерминального символа

<A-выражение> с помощью последовательности правил: правило 4, правило 3, и правило 2. Новая сокращенная форма имеет вид "*<A-выражение>* + *<выражение-многоопред.>*", которое сокращается до нетерминального символа

<A-выражение> с помощью правила 2. Нетерминальный символ*<A-выражение>*сокращается до первоначального символа *<выражение>* с помощью правила 1. Полученное уникальное и правильное дерево разбора представлено на Рисунке 3.15.

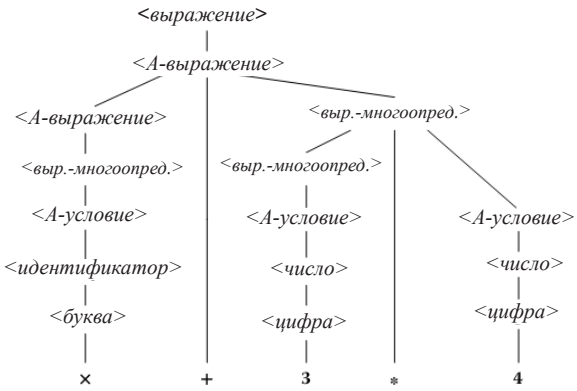


РИСУНОК 3.15 Синтаксический анализ выражения “x + 3 * 4”, выполненный с помощью однозначной грамматики.

ТАБЛИЦА 3.3 Возможные интерпретации не имеющего соответствия вложенного условного оператора

Incorrect Interpretation	Correct Interpretation

`if (x > 4) then`
`  if (y > 0) then return(1);`
`else return(0)`
(a) `else` part is matched with `outer if`
	`if (x > 4) then`
	`  if (y > 0) then return(1);`
	`  else return(0);`
	(b) `else` part matched with the `nearest if`

3.4.3.1 Неоднозначности вложенных структур

Условные оператор имеет два варианта: (1) **if** <*оператор*> **then** <*оператор*> или (2) **if** <*оператор*> **then** <*оператор*> **else** <*оператор*>. Первый вариант не сопоставляемый, так как отсутствует часть else, а второй вариант сопоставляемый, так как присутствует часть else. Сочетание сопоставляемого и несопоставляемого варианта для создания вложенного условного оператора приводит к неоднозначности, так как становится непонятно сопоставляется часть else с внешним if или внутренним if, как показано в Таблице 3.3.

У обоих операторов есть два вхождения “if” и одно вхождение “else.” Правило программирования заключается в том, что else связывается с ближайшем вхождением if. Для правильной интерпретации существует два варианта действий: (1) записать внутреннюю сопоставляемую часть с помощью разделителей, чтобы четко отделить ее от внешних блоков, или (2) записать однозначную грамматику для обработки вложенного

условного оператора. Пример однозначной грамматики приведен ниже

<условный-оператор> **::=** <сопоставляемый-условный-оператор > |
<несопоставляемый-условный-оператор>

<сопоставляемый-условный-оператор> **::= if** <условие> **then**
<сопоставляемый-условный-оператор>
else <сопоставляемый-условный-оператор>|
<другие-операторы>

<несопоставляемый-условный-оператор> **::= if** <условие> **then**
<условный-оператор> |
if <условие> **then**
<сопоставляемый-условный-оператор>
else <несопоставляемый-условный-оператор>

3.4.4 Абстрактное синтаксическое дерево

Дерево разбора, составленное из правил абстрактного синтаксиса, называется *абстрактным синтаксическим деревом*. В *абстрактное синтаксическое дерево* не входят все нетерминальные символы из конкретных синтаксических правил, которых нет в правилах абстрактного синтаксиса. Выражения связываются с операторами. Например, абстрактное синтаксическое дерево для выражения “x + 3 * 4” представлено на Рисунке 3.16.

Конкретные синтаксические деревья (деревья, созданные из конкретных синтаксических правил) сокращаются до абстрактных синтаксических деревьев с помощью удаления излишних нетерминальных символов и нетерминальных символов более низкого уровня, которые напрямую не добавляют значение в предложение. Например, конкретное синтаксическое дерево для выражения “x + 3 * 4”, представленное на Рисунке 3.15, содержит много нетерминальных символов низкого-уровня, таких как {<*A-условие*>, <*выражение-многоопред.*>, <*цифра*>}, которые напрямую не относятся к семантике и будут удалены из бестактного дерева, представленного на Рисунке 3.16.

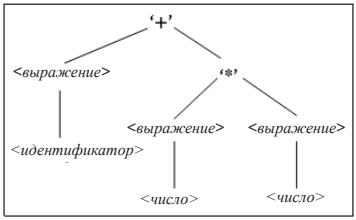


РИСУНОК 3.16 Абстрактное синтаксическое дерево для выражения

“x + 3 * 4.”

Абстрактные синтаксические деревья полезны для понимания языковых конструкций языков программирования, так как они скрывают несущественные детали конкретных синтаксических деревьев. Абстрактные синтаксические деревья также используются в синтаксически управляемой генерации кода программ: после синтаксического анализа конкретное синтаксическое дерево сокращается до соответствующего абстрактного синтаксического дерева, а затем проводится семантический анализ абстрактного синтаксического дерева для создания кода промежуточного уровня. Абстрактные синтаксические деревья также используются в анализе программы и толковании абстрактных свойств программ.

3.4.5 Автоматизированный синтаксический анализ

В предыдущем разделе мы предположили, что при построении дерева разбора автоматически будет определяться подходящая последовательность, которая будет сопоставлять правую часть продукционного правила, и последняя будет сокращаться до нетерминального символа из левой части соответствующего продукционного правила. Для определения таких подпоследовательностей люди используют свои интеллектуальные способности, компьютерам же необходимо автоматизировать программы синтаксического анализа для однозначного определения таких подпоследовательностей, которые могут соответствовать правой части продукционного правила.

Автоматизированные методы синтаксического анализа можно разделить на две основные категории: синтаксический анализ типа *сверху-вниз* и синтаксический анализ типа *снизу-вверх*. При выполнении *синтаксического анализ* типа *сверху-вниз*, также известного как *метод рекурсивного спуска*, крайний слева нетерминальный символ расширяется до правой части продукционного правила и сопоставляется с соответствующей частью анализируемого предложения. Программа синтаксического анализа *возвращается к исходной позиции*— идет в обратном направлении, чтобы найти альтернативные решения — и подставляет альтернативные правила, если сопоставление оказалось неуспешным.

Возвращение к исходной позиции - вычислительно неэффективная методика. Для повышения эффективности и устранения необходимости возвращения к исходной позиции используется *предварительный просмотр терминального символа*. *Предсказывающие синтаксические*

анализаторы используют таблицу синтаксического анализа размером $M \times N$, где M - количество нетерминальных символов и N - количество терминальных символов (использующиеся для предварительного просмотра) в грамматике. Каждая ячейка таблицы синтаксического анализа содержит ссылку на одно или несколько продукционных правил, показывающих определение. Если ячейка ссылается на уникальное продукционное правило, то продукционное правило считается подобранным и процесс предварительного просмотра завершается. В противном случае будут предварительно просматриваться больше символов, и будут изучаться соответствующие ячейки таблицы до тех пор, пока не определится единственное продукционное правило.

Синтаксический анализ типа снизу-вверх, также имеющий название *восходящий синтаксический анализ*, начинается с анализируемого предложения и движется наверх. Синтаксический анализатор определяет соответствующую подпоследовательность в промежуточных сокращенных формах и сокращает их до нетерминального символа в левой части продукционного правила. Основным классом восходящего синтаксического *анализа является LR(k) синтаксический анализ*: *k* означает максимальное количество символов, которые необходимо предварительно просмотреть, чтобы принять однозначное решение при выполнении синтаксического анализа. LR(k) синтаксический анализ строит дерево разбора в обратном порядке, начиная с крайней правой части предложения. LR синтаксический анализ обладает рядом преимуществ: (1) он не является рекурсивным, (2) он не возвращается к исходной позиции, (3) с его помощью можно проводить синтаксический анализ всех программных конструкций, и (4) он может распознавать неоднозначности в грамматике. Особый подкласс LR(K) синтаксического анализатора, имеющий название LALR (синтаксический анализатор LR с предпросмотром), широко используется для программ синтаксического анализа в языках программирования и создается с помощью автоматизированных генераторов синтаксического анализатора. Более детальное изучение автоматизированных синтаксических анализаторов входит в курс изучения компиляторов.

3.5 СЕМАНТИКА

В языках программирования используются пять основных видов семантики: *операционная семантика, аксиоматическая семантика, денотационная семантика, семантика действий* и *поведенческая семантика*. В следующих подразделах объясняются особенности

каждого вида семантики.

3.5.1 Операционная семантика

Операционная семантика, впервые была описана как часть ALGOL 68 и уже позже была сформирована Плоткиным. Ее основной задачей является присвоение значения предложению через описание его влияния на определение абстрактного состояния абстрактной машиной с помощью соответствующего набора абстрактных команд. Значение абстракции управления высокого уровня представляет собой последовательность *небольших пошаговых абстрактных команд*, в результате выполнения которых вычислительное состояние переходит в новое вычислительное состояние. Вычислительные состояния зависят от абстрактной машины, которая в свою очередь основывается на парадигмах программирования и языках программирования. Изменения в вычислительных состояниях при выполнении различных абстрактных команд описывают операционную семантику. Операционная семантика, содержащая *небольшие пошаговые абстрактные команды*, называется *операционной семантикой с малым шагом*. Операционная семантика, которая занимается вопросами перехода вычислительных состояний с использованием абстракций контроля высокого уровня и данных, такими как *цикл FOR, условный оператор* или *цикл WHILE*, называется *операционной семантикой с большим шагом*.

В нашем случае для понимания принципа машинного перевода программ высокого уровня в абстрактные команды низкого уровня мы решили взять машину, в основу которой заложена машина фон Неймана, описанная в Разделе 2.1. Абстрактная машина может смоделировать парадигму императивного программирования. Вычислительное состояние представляет собой тройную форму (*среда, блок памяти, дамп*), а программа - это последовательность *объявления переменных, выражений* и *команд. Объявление переменных* преобразовывает вычислительное состояние путем изменения среды, *команда* изменяет вычислительное состояние путем изменения блока памяти, выражение читает блок памяти без изменения вычислительного состояния. Вызов подпрограммы изменяет как среду, так и блок памяти. Оператор может быть *составным* и может изменять как среду, так и блок памяти. Например, оператор *integer* $x = 10$ использует как объявление переменных, так и оператор присваивания: объявление переменных *integer* x изменяет среду, а оператор присваивания $x = 10$ изменяет блок памяти. Как описано в Разделе 2.4, вычислительное состояние обозначается греческой буквой

σ, а оператор обозначается символом *S* для моделирования перехода между вычислительными

состояниями. Операционная семантика оператора определяется как $(S, \sigma_0) \rightarrow \sigma_1$. *Составной оператор* имеет вид <*S*; *Ss*>, где символ *S* обозначает первый оператор, а

(пустая команда, σ) → σ
(литерал, σ) → литерал
(идентификатор, σ) → r-значение(идентификатор) (σ) если
((идентификатор ↦ l-значение) ∈ σ^E и (l-значение ↦ r-значение) ∈ σ^S
(***новый*** идентификатор, <σ^E, σ^S, σ^D>) → <$\sigma^E \oplus$ (идентификатор ↦
l-значение), $\sigma^S \oplus$ (l-значение ↦ неопределенное), σ^D> (exp_1 op exp_2, σ)
→ значение$_1$ op значение$_2$ иσ не изменяется
где (exp_1, σ) → значение$_1$ и (exp_2, σ) → значение$_2$, и
op ∈
{сложение, вычитание, умножение, деление}
(идентификатор = exp, <σ^E, σ^S, σ^D>) → <σ^E, $\sigma^S \oplus$
(l-значение(идентификатор) ↦ значение, σ^D>
где (exp, <σ^E, σ^S, σ^D>) → значение

РИСУНОК 3.17 Операционная семантика простых абстрактных команд.

символ *Ss* остальные операторы. Для понимания значения составных операторов, нам необходимо понять операционную семантику последовательности операторов. Операционная семантика последовательности операторов имеет вид $(<S; Ss>, \sigma_0) \rightarrow (Ss, \sigma_1) \rightarrow^* \sigma_{final}$, где $(S, \sigma_0) \rightarrow \sigma_1$. Символ "→*" обозначает переходы, количество которых равно количеству операторов в оставшейся последовательности операторов, а σ_{final} обозначает конечное вычислительное состояние.

На Рисунке 3.17 представлена операционная семантика для некоторых наиболее распространенных абстрактных команд с малым шагом, таких как *пустая команда, оценка литерала*, *просмотр* идентификатора, *изменение значения идентификатора*, *оценка простого выражения*, *оценка составного выражения*, *объявление нового идентификатора* и *оператор присваивания*. Для объяснения действия оператора мы обозначаем вычислительное состояние σ в виде формы трех операторов <σ^E, σ^S, σ^D>, где σ^E - среда - преобразование набора идентификаторов в набор l-значений, σ^S обозначает блок памяти— преобразование набора l-значений в набор r-значений, а σ^D обозначает дамп— последовательность частных сред и блоков памяти в цепочке вызывающих процедур, скрытой (невидимой) или заархивированной во время выполнения вызываемой процедуры.

Вычислительное состояние абстрактной машины сохраняется после выполнения пустой команды, оценки литерала, просмотра идентификатора и оценки выражения. При обнаружении идентификатора в среде σ^E просматривается его l-значение, а соответствующее значение l-значения просматривается в блоке памяти σ^S для нахождения соответствующего r-значения.

Оценка составного выражения не изменяет вычислительного состояния. Относительно текущего вычислительного состояния σ измеряются два выражения, а для получения результата для двух значений применяется бинарный оператор. Объявление нового идентификатора <*ident*> изменяет среду: в среду σ^E добавляется новое связывание (<*ident*> $\mapsto$ l-значение). К тому же, к соответствующему блоку памяти σ^S добавляется новое связывание (l-значение $\mapsto$ неопределенное).

Оператор присваивания - это составной оператор. Значение оператора присваивания описывается как: (1) оценивание правой части выражения с помощью текущего вычислительного состояния σ_0 и (2) обновление текущего блока памяти σ^S за счет изменения связывания l-значения идентификатора с вычисленным значением правой части выражения.

Как только значения абстрактных команд с малым шагом будут хорошо распознаны в абстрактной машине, так составное значение абстракций высокого уровня станут ясными для нашего понимания последовательности абстрактных команд с малым шагом, эквивалентных абстрактным командам высокого уровня. Абстракции управления высокого уровня переводятся в последовательность абстрактных команд с малым шагом методом представления абстракций управления высокого уровня в виде диаграммы потоков управления и последующим преобразованием диаграммы потоков управления в последовательность эквивалентных абстрактных команд с малым шагом, которые воздействуют на вычислительное состояние таким же образом, как и абстракции управления высокого уровня. Подробности преобразования абстракций управления высокого уровня, таких как условный оператор, цикл WHILE, цикл FOR, вызов процедуры и их сочетаний, в абстрактные команды низкого уровня описываются в Главе 5.

3.5.2 Аксиоматическая семантика

Программы могут быть разработаны на основе математической логики, такой как исчисление предикатов, независимо от базовой архитектуры и абстрактной машины. Значение программы воспринимается в виде логических выражений, а не как последовательность преобразований

вычислительных состояний в абстрактной машине, основанной на модели фон Неймана. *Аксиоматическая семантика,* основанная на исчислении предикатов, впервые была описана Ч. Э. Р. Хоаром с помощью логических выражений для определения вычислительного состояния в момент вычисления. Выполнение оператора изменяет вычислительное состояние из-за изменения логического выражения. Значение оператора выводится из разности между *постусловием* и *предусловием*. *Предусловие* - это логическое выражение сразу перед выполнением оператора, а *постусловие* - это логическое выражение, которое становится истинным сразу после выполнения оператора. В случае противоречия аксиом, вызванных операторами вследствие перехода к новому условию, как в операторе присваивания, аксиома в предусловии опускается в пользу новой полученной аксиомы. Предположим, что начальные условия до выполнения команды - $\{\mathbf{P}\}$, а постусловия после выполнения оператора - $\{\mathbf{Q}\}$. Тогда символическая запись, описывающая аксиоматическую семантику, имеет вид

$\{\mathbf{P}\}\ S\ \{\mathbf{Q}\}$, где S - оператор.

Пример 3.16

Например, после выполнения оператора присваивания $x = 4$, логическое условие $x == 4$ становится *истинным*. Если предыдущее условие $x == 10 \wedge Y == 9$ было *истинным* до выполнения оператора, то после выполнения оператора присваивания $x = 4$, замещающего логического подвыражение $x == 10$ на логическое подвыражение $x == 4$, логическое выражение $x == 4 \wedge y == 9$ становится *истинным*. Выполнение оператора $y = 5$ приводит к созданию *истинного* постусловия $x == 4 \wedge y == 5$. Значение оператора $y = 5$ эквивалентно разности между *истинным* постусловием $x == 4 \wedge y == 5$ и *истинным* предусловием $x == 4 \wedge y == 9$.

В обычном языке программирования есть абстракции управления, такие как *команда*, *последовательность команд*, *условные операторы*, например, *оператор if*, *итерационные конструкции*, например, *цикл WHILE*, и *вызовы процедур*. Для понимания значения команды и получения конечных условий с помощью аксиоматической семантики мы должны получить постусловия после выполнения этих абстракций управления. Влияние оператора присваивания уже было разобрано в Примере 3.15. При выполнении последовательности операторов постусловие после

выполнения предыдущего оператора становится предусловием для текущего оператора, а постусловие после выполнения текущего оператора становится предусловием для последующего оператора *(объявления, команды или выражения). В следующем правиле символ {P} обозначает предусловие, символ {Q} обозначает постусловие, символ S обозначает текущий оператор, а символ Ss обозначает последовательность операторов после выполнения текущего оператора. Правило гласит, что после выполнения оператора S, новое предусловие для оставшихся операторов Ss - {Q}, где {Q} - это постусловие после выполнения оператора S, учитывая, что предусловие до выполнения оператора S было {P}.*

{P} (S; Ss) → {Q} (Ss) где {P} S {Q}

Аксиоматическая семантика абстракции управления if B then S_1 else S_2 представляется в виде составного постусловия $\{Q_1\}$ V $\{Q_2\}$. Постусловие $\{Q_1\}$ получается после выполнения оператора S_1, если логическое выражение B истинно в предусловии {P}, а постусловие$\{Q_2\}$ получается, если логическое выражение B ложно в предусловии {P}. Мы можем записать правило аксиоматической семантики для условного оператора как:

{P} if B then S_1 else S_2 $\{Q_1 \vee Q_2\}$ где {P} S_1 $\{Q_1\}$, если B истина, или {P} S_2 $\{Q_2\}$, если B ложь

Дизъюнкция (логическое ИЛИ) объединяет then-часть и else-часть. Постусловие будет реализовано или истинной then-частью, или истинной else-частью, поскольку эти части взаимо исключают друг друга.

Пример 3.17

Разберем следующий программный код:

x = 10; z = 4; if (x > 4) then value = x; else value = z;

Примем {P} за начальное условие. После выполнения оператора x = 10, постусловие $\{P \wedge x == 10\}$ становится истиной. Это постусловие становится предусловием для следующего оператора присваивания z =

4. После выполнения оператора z = 4 постусловие {P ∧ x == 10 ∧ z == 4} становится истиной и служит предусловием для условного оператора. Постусловие условного оператора {P ∧ x == 10 ∧ z == 4 ∧ (((x > 4) ∧ (value == x)) ∨ (x ≤ 4) ∧ (value == z)))}.

Аксиоматическая семантика абстракции управления "while B S" является сложной. Правило гласит, что предусловие {I ∧ B} перед выполнением цикла WHILE делится на две части: инвариантную часть I и оставшееся логическое выражение B, а постусловие {I ∧ ¬ B} состоит из той же инвариантной части I и отрицания ¬ B. Инвариантное условие остается неизменным во время выполнения итерации. Инвариантное условие I нельзя вычислить непосредственно. Аксиоматическая семантика для цикла WHILE имеет вид:

{**I** ∧ **B**} while B S; {**I** ∧ ¬ **B**} где {**I**} S {**I**}% **I** - инвариантное условие

Основные преимущества аксиоматической семантики заключаются в следующем:

1. Она может применяться для получения конечного условия после выполнения программы без действительного ее выполнения. Это свойство, в ограниченной степени, может помочь проверить правильность программы без выполнения программы. Идея заключается в том, чтобы получить конечное условие F^C, произведенное с помощью аксиоматической семантики, и сравнить его с имеющимся условием F^I. Если конечное полученное условие $F^C \subseteq F^I$, тогда программа *не содержит шибок* (смотри Приложение VII), а если конечное выполненное условие $F^C \subseteq F^I$ и $F^I \subseteq F^C$, тогда программа и выполнена, и не содержит ошибок. Если схема работает для маленьких программ, то она становится вычислительно неподходящей для больших программ в связи с (1) вычислительными затратами на сопоставление эквивалентных логических выражений, (2) с затратами на установление того, что условие, описанное одним логическим выражением относится к другому логическому выражению, и (3) затраты на обнаружение инвариантых условий.

2. Зная предполагаемое конечное условие F^I, программа может быть создана в виде нескольких этапов с помощью обратного логического вывода: постусловия и воздействие конструкций программы используются для постепенного получения истинных предусловий. Предусловия становятся постусловиями для предыдущего оператора. Повторением такого обратного эвристического рассуждения программа

может быть постепенно создана так, как описано в Главе 4.

3.5.3 Денотационная семантика

Денотационная семантика получает значения абстракций управления или данных преобразованием синтаксического правила в *область семантики* при помощи *математических функций* и *семантической алгебры* в области семантики.

Для получения смысла предложения строится такое же дерево разбора, как и в синтаксическом анализе с тем лишь различием, что на дугах дерева разбора применяются семантические правила, соответствующие синтаксическим правилам, а значения частей предложения получают в виде внутренних узлов. Полное значение предложения получают в исходном узле дерева разбора. Значение сложного предложения получают функциональным составлением отдельных сегментов.

Различие операционной семантики и денотационной семантики заключается в том, что операционная семантика описывает изменения в вычислительном состоянии как результат выполнения абстрактных команд в абстрактной машине, а денотационная семантика использует абстрактные синтаксические деревья и сочетание семантических правил, выраженных математическими формулами, для получения значения предложения. В денотационной семантике нет понятий вычислительного состояния и абстрактной машины.

Денотационная семантика и операционная семантика обладают несколькими сходствами:

1. Денотационная семантика, как и операционная семантика использует среду и блок памяти.

2. И операционная семантика, и денотационная семантика используют абстракции в своих определениях.

Пример 3.18

Давайте разберем понятие денотационной семантики на примере простой грамматики, которая грамматически представляет собой только целые числа без знака в системе с основанием 10. Грамматика, соответствующая целым числам с основанием 10, приведена ниже

Синтаксическое правило #1: <*целое число*> **::=** <*целое*

число><*цифра*> | <*цифра*>

Синтаксическое правило #2: <*цифра*> ::= ‘**0**’ | ‘**1**’ | ‘**2**’ | **…** | ‘**9**’

Область семантики для получения значения целого числа - это целые числа с основанием системы счисления 10, обозначаются как $\mathbb{Z}_{10}$, где индекс указывает основание системы счисления 10. Для определения значения необходимы операции умножения и сложения. Таким образом, семантическая алгебра для области семантики представлена знаками *плюс*, *умножение*: $\mathbb{Z}_{10} \times \mathbb{Z}_{10} \rightarrow \mathbb{Z}_{10}$ это говорит о том, что входными данными является пара целых чисел, при выполнении сложения и умножения также получается целое число с основанием системы счисления 10. Символ “×” обозначает декартово произведение. Семантические правила используют функцию ‘ĭ’ для преобразования синтаксического правила в соответствующее значение области семантики $\mathbb{Z}_{10}$.

Область семантики: Z_{10}

Семантическая алгебра: сложение, умножение: $Z_{10} \times Z_{10} \rightarrow Z_{10}$

Семантическое правило # 1: ĭ (<*целое число*>$_{\text{лч}}$) : = ĭ (<*целое число*>$_{\text{пч}}$) *умножить*

десять *плюс* ĭ(<*цифра*>) | ĭ (<*цифра*>)

Семантическое правило # 2: ĭ (<*цифра*>) : = **ноль** | **один**| **два**| **…** | **девять**

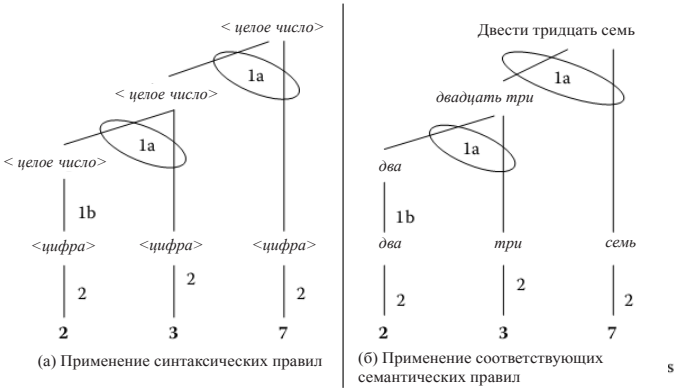
Это скрытое представление того, что мы понимаем под значениями десять, сто и т.д. Предполагая это, мы можем получить значение любой последовательности цифр в 10-ной системе счисления. Давайте разберем с помощью грамматики последовательность из трех цифр 237, и затем, пользуясь таким же деревом разбора для применения соответствующих семантических правил, дадим определение *двумстам тридцати семи* в области семантики $\mathbb{Z}_{10}$. Составление дерева разбора и получение значения посредством соответствующих семантических правил представлены на

Рисунке 3.18.

Как показано на дереве разбора (Рисунок 3.18a), каждый символ разбивается на <*цифра*> согласно синтаксическому правилу 2. Согласно синтаксическому правилу 1b мы можем представить крайний левый символ <*цифра*> как <*целое число*>, а затем, используя синтаксическое правило 1a, дважды связать две другие цифры с определением <*целое*

число>. В основании дерева разбора будет находится начальный символ <*целое число*>.

На Рисунке 3.18b представлено применение соответствующих семантических правил в дереве разбора для определения значения поддерева во внутренних узлах и значения предложения в корневом узле. Семантическое правило 2 придает значения *два*, *три* и *семь* символам ‘*2,*’ ‘*3,*’ и ‘*7*’, соответственно. Семантическое правило 1b преобразует цифру в целое число *два*. Применяя семантическое правило 1a, получаем значение *два умножить на десять плюс три = двадцать три*. Снова применяя семантическое правило 1a, получаем значение *двадцать три умножить на десять плюс семь = двести тридцать семь*.



(а) Применение синтаксических правил

(б) Применение соответствующих семантических правил

s

РИСУНОК 3.18 Получение значения с помощью применения семантических правил в дереве разбора.

Пример 3.19

Для понимания денотационной семантики разберем пример определения чисел в языках программирования. Домен натурального числа - это составной домен, включающий несвязное объединение домена целого числа и домена действительного числа. Тем не менее, домен целого числа можно привести к домену действительного числа, в случае если арифметическое действие включает как целое число, так и действительное число. Синтаксические правила для чисел представлены на Рисунке 3.19. Синтаксические правила легко читаются, за исключением одной формулировки: определения нетерминальных

символов *<целые неотрицательные числа>* и *<число с плавающей точкой>* образуют последовательность *<цифр>*. Тем не менее, для улучшения семантической интерпретации их можно записать в другом виде: пока

<целое неотрицательное число> создает последовательность цифр посредством леворекурсивного определения, определение *<число с плавающей точкой>* использует определение хвостовой рекурсии. Хотя конечный результат является одинаковым, способ передачи значения, связанный с двумя синтаксическими правилами, различен. Значение нетерминального символа*<целое неотрицательное число>* при сдвиге влево *умножается на десять*, а значение нетерминального символа *<число с плавающей точкой>* при сдвиге вправо *делится плавающей точкой на* 10.

Значение *<целое неотрицательное число>* отличается от домена целого числа и домена действительного числа: в домене целого числа оно рассматривается как целое число; а в домене действительных чисел значение числа *приводится* к

Правило 1: *<число>* **::=** *<целое число>* | *<действительное число>*
Правило 2: *<целое число>* **::=** *<знак>* *<целое неотрицательное число>* | *<целое неотрицательное число>*
Правило 3: *<действительное число>* **::=** *<знак>* *<действительное число без знака>* | *<действительное число без знака>*
Правило 4: *<действительное число без знака>* **::=** *<целое неотрицательное число>* '.' *<число с плавающей точкой>* |*<целое неотрицательное число>* '.' *<число с плавающей точкой>* '**E**' *<целое число>*
Правило 5: *<целое неотрицательное число>* **::=** *<целое неотрицательное число>* *<цифра>*|*<цифра>*
Правило 6: *<число с плавающей точкой>* **::=** *<цифра><число с плавающей точкой>* | *<цифра>*
Правило 7: *<цифра>* **::=** '**0**' | '**1**' | '**2**' | … | '**9**'
Правило 8: *<знак>* **::=** ' + ' | '−'

РИСУНОК 3.19 Синтаксические правила для проверки чисел.

соответствующему числу в домене действительных чисел. Область семантики для правил интерпретации - это *домен натуральных чисел с основанием* 10, обозначается N. Необходимо отметить, что область натуральных чисел N - это несвязное объединение домена целых чисел Z и домена действительных чисел R. Тем не менее, каждый элемент $z \in Z$ обладает уникальным образом $r \in R$, имеющим такое же значение. Преобразование из домена целых чисел в домен действительных

чисел - это пример *приведения типа* - вида преобразования без потери информации, который обсуждается в Главе 7. Семантическая алгебра включает *сложение целых чисел* (‘$+^{I}$’), *умножение целых чисел* (‘$\times^{I}$’), *сложение действительных чисел* (‘$+^{R}$’), *умножение действительных чисел* (‘$\times^{R}$’), *деление действительных чисел* (‘$/^{R}$’) и *возведение в степень* (‘^’). Операции *сложения* и *умножения* имеют дополнительную функцию: сложение становится сложением целых чисел в домене целых чисел и сложением чисел с плавающей точкой в домене действительных чисел. Аналогично, операция умножения становится умножением в домене целых чисел и умножением чисел с плавающей точкой в домене действительных чисел.

На Рисунке 3.20 представлены семантические правила в том же порядке, что и синтаксические правила. Функция ň обозначает интерпретацию синтаксического правила в домене натуральных чисел N, функция ř обозначает интерпретацию синтаксического правила в домене действительных чисел R, а функция ĭ обозначает интерпретацию в домене целых чисел ′.

Давайте изучим каждое семантическое правило. Правило 1 утверждает, что значение числа, обозначаемого ň(<*число*>), эквивалентно значению действительного числа, обозначаемого ř(<*действительное число*>) или значению целого числа, обозначаемого ĭ(<*целое число*>). Правило 2 утверждает, что значение нетерминального символа <*целое число*> такое же, как значение нетерминального символа <*знак*> в домене целых чисел, умноженного на значение нетерминального символа

<*целое число без знака*>, или значение нетерминального символа <*целое число без знака*>. Правило 3 утверждает, что значение нетерминального символа <*действительное число*> получается умножением значения нетерминального символа <*знак*> в домене действительных чисел и значения нетерминального символа <*действительное число без знака*> или значением нетреминального символа

<*действительное число без знака*>.

Области семантики: домен целых чисел $\mathbb{Z}_{10}$

домен действительных чисел $\mathbb{R}_{10}$

домен натуральных чисел: $\mathbb{N}_{10} = \mathbb{Z}_{10} \uplus \mathbb{R}_{10}$ % $\uplus$ обозначает несвязное объединение

Семантическая алгебра в $\mathbb{R}_{10}$: сложение действительных чисел ‘$+^{R}$’; умножение действительных чисел ‘$\times^{R}$’: $\mathbb{R}_{10} \times \mathbb{R}_{10} \rightarrow \mathbb{R}_{10}$

Семантическая алгебра в $\mathbb{Z}_{10}$: сложение целых чисел ‘$+^{I}$’;

умножение целых чисел ‘$\times^{I}$’: $\mathbb{Z}_{10} \times \mathbb{Z}_{10} \rightarrow \mathbb{Z}_{10}$

Семантическая алгебра в смешанном домене: возведение в степень^’: $(\mathbb{R}_{10} \times \mathbb{Z}_{10}) \rightarrow \mathbb{R}_{10}$

Семантические функции: ň; ř; ĭ;

Правило 1: ň(<*натуральное число*>) ::= ĭ(<*целое число*>) | ř(<*действительное число*>)

Правило 2: ĭ(<*целое число*>) ::= ĭ(<*знак*>) $\times^{I}$ ĭ (<*целое неотрицательное число*>) | ĭ(<*целое*

неотрицательное число>)

Правило 3: ř (<*действительное число*>) ::= ř(<*знак*>) $\times^R$ ř(<*действительное число без знака*>) | ř(<*действительное число без знака*>)

Правило 4: ř (<действительное число без знака>*) ::=* ř(<*целая неотрицательная часть*>) $+^R$ ř(<*дробная часть десятичной дроби*>) |ř(<*целая неотрицательная часть*>) $+^R$ ř(<*дробная часть десятичной дроби*>)) ×(10.0 ∧ ĭ(<*целое число*>))

Правило 5a: ř(<*целая неотрицательная часть*>1) ::= ř(<*целая неотрицательная часть*>2) $\times^I$ 10.0 $+^R$ ř(<*цифра*>) | ř(<*цифра*>)

Правило 5b: ĭ(<*целое неотрицательное число*>1) ::= ĭ (<*целое неотрицательное число*>2) $\times^I$ десять $+^I$ ĭ(<*цифра*>) | ĭ (<*цифра*>)

Правило 6: ř(<*дробная часть десятичной дроби*>1) :: ř(<*цифра*>) $+^R$ ř(<*дробная часть десятичной дроби*>2)) / 10.0 | ř(<*цифра*>) / 10.0

Правило 7a: ř(<*цифра*>) ::= **ноль с плавающей точкой** | **один с плавающей точкой** | … | **девять с плавающей точкой** % 1 с плавающей точкой равно 1,0

Правило 7b: ĭ(<*цифра*>) ::= **ноль** | **один**| **два** | … | **девять** % интерпретация цифр

Правило 8a: ř(<*знак*>) ::= **плюс число с плавающей точкой один** | **минус число с плавающей точкой один** % + 1.0 or −1.0

Правило 8b: ĭ(<*знак*>) ::= **плюс один** | **минус один**

РИСУНОК **3.20** Семантические правила, соответствующие синтаксическим правилам.

Правило 4 гласит, что значение нетерминального символа <*действительное число без знака*> - это сумма значений нетерминального символа <*целая неотрицательная часть*> и нетерминального символа <*дробная часть десятичной дроби*>. С другой стороны, правило 4 утверждает, что значение <*действительное число без знака*> - это сумма значений нетерминального символа <*целая неотрицательная часть*> и <*дробная часть десятичной дроби*>, умноженная на значение экспоненциальной части. Значение экспоненциальной части - это отображение 10 в домене действительных чисел, возведенное в степень значения целого числа в экспоненциальной части.

Правило 5 имеет разные интерпретации для домена действительных чисел и домена целых чисел. Правило 5 утверждает, что значение нетерминального символа <*целая неотрицательная часть*> левой части продукционного правила - это сумма (1) произведения числа с плавающей точкой 10, и значения нетерминального символа <*целая неотрицательная часть*> правой части продукционного правила, и (2) значения нетерминального символа <*цифра*> в домене действительных чисел, полученного согласно правилу 7a. Правило 5b утверждает, что значение левой части нетерминального символа <*целое неотрицательное число*> - это сумма (1) произведения целого числа 10 и значения правой

части вхождения нетерминального символа

<*целое неотрицательное число*> в домене целых чисел и (2) значения нетерминального символа <*цифра*>

в домене целых чисел, полученного согласно правилу 7b.

Правило 6 гласит, что значение нетерминального символа <*дробная часть десятичной дроби*> - это отображение в домене действительных чисел суммы значений нетерминального символа <*цифра*> в правой части и нетерминального символа <*число с плавающей точкой*> в правой части, полученное согласно числу с плавающей точкой 10. Правила 4 и 6 отличаются, несмотря на то, что они оба формируют последовательность цифр. Значения, связанные с обоими правилами, отличаются. В определении нетерминального символа <*целое неотрицательное число*> в правиле 5, перемещение <целое *неотрицательное число*> влево эквивалентно умножению значения нетерминального символа <*целое неотрицательное число*> на целое число 10, в то время как в определении <*число с плавающей точкой*> в правиле 6 перемещение <*число с плавающей точкой*> вправо эквивалентно делению значения <*число с плавающей точкой*> на число с плавающей точкой 10. Кроме того, два значения определяются в разных доменах: нетерминальный символ <*целое неотрицательное число*> определяется в

семантической области $\mathbb{Z}_{10}$, а нетерминальный символ <*число с плавающей точкой*> определяется в семантической области $\mathbb{R}_{10}$.

Правило 7 имеет две интерпретации: одну для домена действительных чисел и одну для домена целых чисел. Правило 7a утверждает, что значение нетерминального символа <*цифра*> есть ноль с плавающей точкой, один с плавающей точкой, два с плавающей точкой, три с плавающей точкой и т.д. Правило 7b утверждает, значение нетерминального символа <*цифра*> может быть целым числом один, два, три и т.д.

Правило 8 гласит, что значение знака ‘+’ имеет значение *плюс один*, а значение знака ‘−’ имеет значение *минус один*. Отображение *минус один* в области семантики $\mathbb{R}_{10}$ - *минус один с плавающей точкой*, а отображение *плюс один* в области семантики $\mathbb{R}_{10}$ - *плюс один с плавающей точкой*.

Для понимания денотационной семантики абстракций управления и данных области семантики должны быть четко определены. Кроме того, в функциональное определение денотационной семантики вводится понятие ошибки включением элемента, обозначаемого символом $\perp$, в область значений семантики. Включение данного нижнего символа преобразует область значений семантики в соответствующую *расширенную область семантики*. Если применение семантической

функции не находит отображения в *расширенной области семантики*, тогда функция преобразуется в нижний символ $\perp$. Употребление нижнего символа $\perp$ позволяет обрабатывать условия ошибок программы в значении на основе функции в денотационной семантике. Вот некоторые из областей семантики: (1) *набор базовых значений, обозначается символом* B, (2) набор идентификаторов, обозначается символом *Ide*, (3) набор l-значений, обозначается символом L, (4) набор r-значений, обозначается символом R, (5) набор накапливаемых значений, обозначается несвязным объединением $B \uplus L$, (6) блок памяти, обозначается символом S, который эквивалентен $L \mapsto R \uplus$ *неиспользуемые ячейки*, (7) набор процедур, обозначается символом P, (8) набор различимых значений (l-значений и накапливаемых значений), обозначается символом D, *эквивалентным* $L \uplus R \uplus$ *{неопределенное},*

(9) среда, обозначается преобразованием $Ide \mapsto D$, (10) набор результатов выражений, обозначается расширенной областью $R_{\perp}$, (11) набор результатов команд, обозначается расширенной областью $S_{\perp}$, и

(12) набор возвращаемых из программы значений, обозначается расширенной областью $B_{\perp}$.

Как правило, денотационная семантика страдает от ряда проблем: (1) изменяются области семантики при добавлении в языки программирования новых абстракций управления и данных; (2) семантика итерационного цикла объясняется относительно сложных математических понятий, таких как *семантика фиксированной точки*, несмотря на то что программисты хорошо интуитивно понимают абстракцию управления, и им не нужны для этого такие математические понятия; (3) добавление команд “goto” требует понятия *продолжения*, на самом деле последовательность команд выполняется сразу после выполнения текущей команды; и (4) сложные обозначения, которые непросто понять программистам и разработчикам языков.

Применение функций для определения значения в денотационной семантике делают их удобными для парадигмы функционального программирования, и не удобными для других парадигм программирования, таких как парадигма объектно-ориентированного программирования, парадигма императивного программирования и парадигма ситуационного программирования, которые имеют лучшее представление в семантике, основанной на преобразованиях вычислительного состояния.

3.5.4 Семантика действий

Модели семантики используются для объяснения значения абстракций разработчикам программ, разработчиками языков и разработчиками компиляторов. Три основных модели семантики (операционная, аксиоматическая и денотационная), взятые по отдельности, ограничены и делают описание абстракций языка программирования трудными для восприятия. Тем не менее, прагматичное сочетание трех видов семантики обеспечивает лучшее представление для разработчиков языков и разработчиков компиляторов.

Семантика действий соединяет в себе преимущества денотационной семантики, аксиоматической семантики и операционной семантики, что позволяет ей объяснять значения абстракций управления и данных, используя правила естественного языка. Эти правила называются *действиями*. Основным преимуществом семантики действий является то, что она определяет значения абстракций реалистичного языка программирования в доступной форме.

Семантика действий заимствует идею контекстно-свободной грамматики для получения абстрактных синтаксических деревьев, а также использует семантические уравнения из денотационной семантики. Существует три основных составляющих семантики действий: *действие*, *данные* и *выдача значений*. *Действия* представляют вычислительные объекты, отвечающие за поэтапную детализацию поведения работающей программы. Информация, обработанная действиями, называется *данными*. *Выдача значений* извлекает информацию, но не обрабатывает ее. Существует два вида действий: *простые* и *комбинированные*. *Простое действие* отвечает за единичный шаг обработки информации, а *комбинаторы* занимаются составными действиями. Действие может быть *завершенным*, *отклоняющимся* или *поврежденным*, как результат обработки исключений.

Спецификация семантики действий состоит из (1) спецификации узлов, которые будут созданы в абстрактном синтаксическом дереве с помощью обозначения [[**…**]], и (2) группировки компонентов в виде последовательности операторов. Для каждого семантического правила вместо сложных функциональных записей дается соответствующая семантика действий на простом естественном языке (английском). Например, семантическое правило, соответствующее синтаксическому правилу <*идентификатор*> ‘=’ <*выражение*>, может быть следующим:

выполнить[[<*идентификатор*> ‘=’ <*выражение*>]]
действие: оценить <*выражение*>, сначала придав значение, затем

сохраняет результирующее значение в l-значении (<*идентификатор*>)
Аналогично составляется семантика действий условного оператора

выплнить[[‘if’ <*выражение*> ‘then’ <*оператор*>1 ‘else’ <*оператор*>2]]

действие: оценить <*выражение*>, придается значение *B*, затем ((проверить истинное-значение *B* и выполнить <*оператор*>1) (проверить не (истинное-значение *B*) и выполнить <*оператор*>2))

Общий формат семантики действий содержит объявление модулей, объявление семантических переменных, представляющих области семантики, и правила действий для каждого правила абстрактного синтаксиса. *Выдача значений* используется, когда данные извлекаются после некоторого действия.

3.5.5 Другие модели семантики

Существуют другие поведенческие модели семантики для предоставления значения *предметно-ориентированным языкам* и *объектно-ориентированным языкам*. Поскольку разработка программного обеспечения становится более сложной, то для нее требуется больше чем один домен, и каждый домен должен обладать *предметно-ориентированным языком* (DSL). Семантика DSL может быть смоделирована как переход состояний группы активных объектов из одного состояния в другое за счет использования откликов объектов на обращения, как способов перехода из текущего состояния в следующее состояние. Диаграмма перехода состояний выражается в виде поведенческой семантики языка. Каждый объект моделируется кортежем формы (*набор признаков*, *методов*, *сообщений*, *которые* он *может получить*, *сообщения*, *которые он может отправить*, *переходы*, *которые происходят в ответ на сообщения*, и *триггеры*, к*оторые инициируют реакцию в ответ на сообщение*). *Состояние системы* моделируется как набор состояний объектов. Сообщение от одного объекта другому объекту может инициировать новый переход в полученном объекте, если выполнены условия инициирования, и изменить состояние полученного объекта, таким образом изменяя вычислительное состояние в целом. Измененный объект посылает новые сообщения, которые могут спровоцировать изменение в переходе состояний в других объектах, что в результате приведет к изменению состояния всей системы.

2.5 КРАТКИЕ ВЫВОДЫ

В этой главе описаны два основных элемента языков программирования: *синтаксис* и *семантика*. И синтаксис, и семантика необходимы для понимания конструкций, их значений и их переходов к промежуточному коду низкого уровня. *Синтаксис* отвечает за правильность структуры предложения, а *семантика* отвечает за получение значения операторов в языках программирования. Процесс анализа правильности структуры предложения состоит из двух стадий: *лексического анализа* и *синтаксического анализа*. Лексический анализ отвечает за интернализацию входного потока символов за счет создания токенов. Синтаксический анализ отвечает за проверку на правильность структуры токенизированных предложений.

Грамматика состоит из 4-кортежной формы (*начальный символ*, *набор продукционных правил*, *набор нетерминальных символов и набор терминальных символов*). *Набор терминальных символов* формируется из *алфавита* языка программирования. Набор *нетерминальных символов* используется в грамматике для создания промежуточных форм предложений во время синтаксического анализа. Автоматизированные синтаксические анализаторы могут быть двух типов: *сверху-вниз* или *снизу-вверх*. Автоматизированные синтаксические анализаторы используют таблицы перехода и предварительный просмотр символа для разрешения неоднозначности применения подходящих продукционных правил. Автоматизированный синтаксический анализатор замещает часть переходной формы предложения на соответствующий нетерминальный символ левой части продукционного правила в грамматике с сопоставлением правой части. Процесс синтаксического анализа проводится до тех пор, пока промежуточная форма не будет упрощена до начального символа. Продукционные правила грамматики представлены в форме текста как БНФ или РБНФ или визуально в форме *синтаксических диаграмм*. БНФ использует правила хвостовой рекурсии для моделирования повторяющегося вхождения группы символов. РБНФ убирает сложность определений хвостовой рекурсии с помощью фигурных скобок и специальных символов, таких как ‘*’ или ‘+’. РБНФ также уменьшает избыточность повторного определения с помощью группировки повторных определений в круглые скобки, и использует квадратные скобки для изображения необязательного определения.

Синтаксические диаграммы являются графическим изображением и лучше воспринимаются человеком. Синтаксические диаграммы могут быть преобразованы в синтаксические правила (1) простым

соотношением определений хвостовой рекурсии с циклом с обратной связью, (2) переходом параллельных дуг в повторные определения, (3) преобразованием нескольких смежных узлов на одной дуге во вхождение нескольких объединенных символов.

Есть два типа правил: *конкретные синтаксические правила* и *абстрактные синтаксические правила. Конкретные синтаксические правила* содержат полный набор нетерминальных символов. *Абстрактные синтаксические правила* используют в качестве нетерминальных символов только абстракции управления и данных и избегают нетерминальных символов низкого уровня. Дерево разбора, созданное согласно абстрактным синтаксическим правилам, называется *абстрактным синтаксическим деревом*, и может быть получено из *конкретного дерева разбора* при удалении избыточных нетерминальных символов и сохранении терминальных символов и символов абстракций управления и данных. Продукционные правила также связываются с различными атрибутами для оказания помощи перевода проанализированных предложений в коды промежуточного уровня. Атрибуты в атрибутной грамматике устанавливают ограничения, наложенные разработчиками архитектуры и языка, для контроля эффективности и прагматичных проблем. Атрибутная грамматика также содержит семантические правила для создания кода низкого уровня из дерева разбора.

Семантика может быть разделена на пять основных типов: *операционная семантика*, *аксиоматическая семантика*, *денотационная семантика, семантика действий* и *поведенческая семантика. Операционная семантика* создает абстракции управления и данных как переход из одного вычислительного состояния в другое вычислительное состояние в абстрактной вычислительной машине. *Операционная семантика с малым шагом* описывает, как состояние базовой абстрактной машины изменяется при выполнении абстрактной команды низкого уровня. *Операционная семантика с большим шагом* описывает, как состояние базовой абстрактной машины изменяется при воздействии абстракций данных и управления высокого уровня. Аксиоматическая семантика использует логические выражения в качестве предусловий и постусловий для предложения и дает значение предложению в виде различия между постусловиями и предусловиями. Предусловия и постусловия представляют собой логические выражения. Денотационная семантика придает значение правилам абстрактного синтаксиса с помощью математических функций в различных областях

семантики. Как аксиоматическая, так и денотационная семантика не зависят от базовой абстрактной вычислительной машины. Тем не менее, и аксиоматическая, и денотационная семантика являются достаточно сложными для полномасштабных языков программирования по различным причинам. В аксиоматической семантике сложно смоделировать инвариантные условия в циклах и сопоставить логические выражения. Области семантики в денотационной семантике изменяются, если появляются новые конструкции высокого уровня; использование операторов “goto” требует применения *семантики* на *основе понятия продолжения*, и использование сложной семантики с фиксированной точкой для объяснения значения цикла WHILE, которое легко воспринимается программистами на понятийном уровне. Ни денотационная, ни аксиоматическая семантика не могут объяснить конструкции событийно-ориентированного программирования и объектно-ориентированного программирования.

Операционная семантика наряду с диаграммами потоков управления может использоваться для перевода абстракций управления высокого уровня в эквивалентные абстрактные команды низкого уровня, таким образом, что итоговое воздействие на вычислительное состояние будет одинаковым. Аксиоматическая семантика в ограниченной степени применяется для проверки правильности программ и научной разработки программ, для этого используются преобразования логических условий и обратный логический вывод. Денотационная семантика используется для синтаксически управляемых преобразований и выполнения программ.

Семантика языковых конструкций необходима для программистов, разработчиков языков и разработчиков компиляторов для генерации кода некоторых компиляторов. Для облегчения понимания семантики была разработана *семантика действий*. *Семантика действий* состоит из трех элементов: *действий*, *данных* и *выдачи значений*. Действия работают над данными, преобразовывая текущее состояние, а выдача значений извлекает данные из текущего состояния. Действия описываются на естественном языке (английском) для улучшения восприятия.

Для управления предметно-ориентированными и объектно-ориентированными языками используется поведенческая семантика, основанная на модели перехода состояния. Вычислительное состояние - это совокупность состояний отдельных объектов и поведенческой семантики, смоделированной в виде перехода от одного состояния в другое за счет использования откликов объектов на входящие обращения.

Модель не зависит от исполнения, лежащего в основе.

3.7 ОЦЕНКА

3.7.1 Концепции и Определения

Абстрактная машина; абстрактный синтаксис; абстрактное синтаксическое дерево; семантика действий; алфавит; неоднозначная грамматика; атрибутная грамматика; автоматизированный синтаксический анализ; аксиоматическая семантика; поведенческая семантика; БНФ; нижний символ; синтаксический анализ снизу-вверх; контекстно-свободная грамматика; контекстно-зависимая грамматика; денотационная семантика; расширенная БНФ; грамматика; гипер-правила; лексический анализ; расширенная область; мета-определения; нетерминальные символы; операционная семантика; синтаксический анализ; дерево разбора; предусловие; постусловие; продукционные правила; корректность программы; регулярная грамматика; семантическая алгебра; область семантики; семантическое правило; предложение; начальный символ; синтаксис; синтаксическая диаграмма; синтаксическое правило; синтаксический анализ сверху-вниз.

3.7.2 Решение Задач

4. 1. Составьте конечный автомат, который вводит зарезервированные слова ‘int’, ‘float’, ‘char’, ‘Bool’, and ‘;’, числа и идентификаторы, допуская, что зарезервированные слова, числа и идентификаторы могут быть разделены единичным пробелом, а все другие пробелы должны быть удалены.

5. 2. Напишите простое представление БНФ грамматики, которая принимает восьмеричное число. Преобразуйте грамматику в расширенную БНФ.

3. Постройте синтаксическую диаграмму для следующих синтаксических правил:

<блок> **::=** ‘**{**‘ *<объявление>***;** *<операторы>* ‘**}**’

<операторы> **::=** *<оператор>* **‘;’***<операторы>* |*<оператор>* ‘**;** ‘|
∈

<оператор> **::=***<операторусловия>*|*<цикл WHILE>*|*<присваивание>*
|*<цикл FOR>* | *<вызов процедура>* | **return** (*<выражение>*)

<цикл WHILE> **::= while** *<логическое выражение>* **do** *операторы>*

<условный оператор> **::= if** *<логическое выражение>* **then**

<оператор> **else** *<оператор>*
<присваивание> **::=** *<идентификатор>* = *<выражение>*
<логическое выражение> **::=** *<логическое выражение>*
<l-операция> *<логическое выражение>* | *<предикат>* |**true** | **false**
<l-операция> **::= '&&'** | **'||'** | **'not'**
<предикат> **::=** *<идентификатор> с-операция><идентификатор>*
<с-операция> **::='=='**| **'>'**| **'>='**| **'=<'**
<выражение> **::=** *<выражение>* *<а-операция>* *<выражение>*
<а-операция> **::= ' + '** | **'–'** | **'*'** | **'/'**
<объявление> **::=** *<тип>* *<последовательность идентификаторов>*; *<объявление>* | ∈
<тип> **::= int** | **float** | **Bool** | **string**
<последовательность идентификаторов> **::=** *<идентификатор>* **','***<последовательность идентификаторов>* |*<идентификатор>*

4. Напишите простую БНФ для цикла WHILE, включающую логическое выражение и оператор, и нарисуйте соответствующую синтаксическую диаграмму.

5. Изучите передачу параметров в C++ и напишите соответствующие синтаксические правила. Преобразуйте синтаксические правила в синтаксические диаграммы и оптимизируйте синтаксическую диаграмму.

6. Напишите простую однозначную грамматику в РБНФ для конструкции условного оператора if, включая в нее полное логическое выражение, на C++ или Java и нарисуйте соответствующую синтаксическую диаграмму.

7. Постройте дерево разбора для числа с плавающей точкой 23,416, используя синтаксическую грамматику (Рисунок 3.14) и семантические правила (Рисунок 3.15) для нахождения значения числа.

8. Напишите предусловия и постусловия для следующего фрагмента кода, начиная с состояния "не определено."

```
x = 4; y = 6; z = 7;
if (x > y) then max = x else max = y;
```

9. При условии, что среда $\sigma^E = [x \mapsto 1, y \mapsto 2, z \mapsto 3]$, а блок памяти $\sigma^S = [1 \mapsto 43, 2 \mapsto 5, 3 \mapsto 10]$ и дамп $\sigma^D = []$, и предполагая, что вычислительное состояние представлено как тройка (σ^E, σ^S, σ^D), напишите вычислительное состояние после объявления "int w = 5;", после которого выполняется команда "x = x + 4; y = z;".

10. Напишите грамматику, которая принимает шестнадцатеричное число, и запишите соответствующую денотационную семантику в области целых чисел. Определите область, семантическую алгебру и оценочную функцию.

3.7.3 Развернутый ответ

11. Сравните три типа грамматики: регулярную грамматику, контекстно-свободную грамматику и контекстно-зависимую грамматику.

12. Почему регулярная грамматика предпочтительнее по сравнению с другими типами грамматики для лексического анализа? Объясните.

13. Какова роль таблицы символов в лексическом анализе? Объясните на простом примере.

14. В чем заключается различие между абстрактным синтаксисом и конкретным синтаксисом? Объясните на простом, но наглядном примере.

15. Рассмотрите возможный источник неоднозначности в условном операторе и способы устранения неоднозначности.

16. Почему логическое ИЛИ входит в аксиоматическую семантику? Объясните на простом примере.

17. Сравните семантику действий с другими видами семантики и укажите преимущества семантики действий.

19. Зачем нужна поведенческая семантика в языках программирования? Объясните.

Глава 4

Абстракции в программах и обмен информацией

ВТОРОСТЕПЕННЫЕ ПОНЯТИЯ

Абстрактный синтаксис (Раздел 3.2.6); Абстрактные понятия в вычислениях (Раздел 2.4); Грамматика (Раздел 3.1); Управляющая логика (Раздел 1.4.2); Концепции структуры данных (Раздел 2.3); Дискретные структуры (Раздел 2.2); Графики (Раздел 2.3.6); Опыт программирования; Деревья (Раздел 2.3.5); Синтаксические диаграммы (Раздел 3.3).

Программа заключается в управлении структурированными данными для достижения предполагаемого вычислительного состояния, которое удовлетворяет требованиям имеющегося конечного условия. Программирование может быть выполнено на нескольких уровнях: на уровне машинного программирования, программирование на уровне сборки, процедурное программирование низкого уровня, процедурное программирование высокого уровня, декларативное программирование и т.д. Количество абстракций и уровень явного управления разделяют языки программирования высокого уровня и языки программирования низкого уровня. Одна из целей абстракций заключается в том, чтобы обеспечить повторное использование программного обеспечения с минимальными изменениями, то есть, чтобы расходы на развитие программного обеспечения при увеличении требований или технологических изменениях были минимальными.

Существует два типа абстракций, которые были описаны ранее: *абстракции данных* и *абстракции управления. Абстракции данных* используются для создания реальных объектов с помощью требуемой подгруппы их атрибутов, а *абстракции управления* используются для создания последовательности команд, что улучшает их восприятие и

использование. Элемент данных может быть охарактеризован набором атрибутов и может быть абстрагирован с помощью подходящей группы тех атрибутов, которые являются необходимыми для решения имеющейся проблемы.

Элементы данных обычно описываются в мотивировочной части программы, а составляющая управления находится в теле программы. Как было описано ранее, мотивировочная часть программы изменяет среду, а раздел управления программы, который содержит операторы присваивания, изменяет блок памяти программы.

В зависимости от языка программирования, разграничение между данными и управлением может быть четким или расплывчатым:

1. Блок программы может быть рассмотрен как данные, а программа может быть создана в динамическом режиме во время выполнения в виде данных, а в дальнейшем преобразована в блок программы.

2. Программа может использовать другую программу как данные, как это происходит в мета-программах.

3. Класс может сформировать единый пакет абстракций данных и абстракций управления и регулировать воздействие других вычислительных объектов, объявляя элементы, находящиеся в этом пакете, либо открытыми, либо защищенными, либо закрытыми.

Как было сказано ранее, программа состоит из нескольких подпрограмм или модулей, которые обмениваются друг с другом информацией. Информация, регулируемая разработчиком и спрятанная с помощью инкапсуляции, также является частью абстракции. Инкапсуляция является естественной границей для скрытия данных. Инкапсуляция может быть осуществлена подпрограммами, объектами и классами, модулями или их сочетанием. Классы обеспечивают скрытие информации посредством инкапсуляции как абстракций данных, так и абстракций управления. Видимость таких абстракций данных и управления может регулироваться в явном виде с помощью *механизма импорта-экспорта* или *использования наследования. Механизм экспорта-импорта* состоит из двух элементов: выборочно делает видимыми для других модулей и программ инкапсулированные данные или программные модули с помощью *механизма экспорта* и извлекает данные или программные модули из модуля или программы с помощью *механизма импорта*. Механизм экспорта делает закрытые данные

открытыми, а механизм импорта позволяет видеть экспортированный элемент в локальной среде. Импорт и экспорт напрямую объявляются программистом.

В императивном программировании основная единица, содержащая абстракции данных и абстракции управления, - это подпрограмма или модуль, в парадигме объектно-ориентированного программирования основная единица - это подкласс или объект - действующий представитель класса или подкласса. В языках на основе модулей подпрограмма или элемент данных может быть инкапсулирован внутри модуля, и применение элемента данных или подпрограммы может регулироваться за счет *применения механизма экспорта-импорта.*

В этой главе мы изучим абстракции данных и управления, обмен информацией между частями программы и различные абстрактные программные конструкции. Тем не менее, конкретные примеры, связанные с парадигмами программирования, отличающимися от парадигмы императивного программирования, будут описаны в соответствующих главах, после того, как парадигмы будут рассмотрены в подробностях.

4.1 АБСТРАКЦИИ ДАННЫХ

Одним из важных понятий в абстракции данных является создание элемента абстрактных данных, который может быть использован для хранения информации, которую необходимо обработать. При этом информация может несколько раз использоваться в различных местах программного модуля, и ее можно легко изменить. Определение бывает *конкретным* или *общим*, которое можно создать во время выполнения программы в указанной абстракции данных через механизм передачи параметров или переобозначение, используя механизм замещения. Элемент данных может быть *единым элементом*, таким как “литерал” или “нечисловая константа”, который в дальнейшем нельзя разделить, или элемент может представлять собой *объединение* элементов, обладающих рядом общих свойств, над которыми выполняются общие, точно определенные операции. Набор общих свойств не следует путать с обладанием объектами общим типом данных. Единственным ограничением является некоторое общее свойство, которое делает возможной повторяющуюся общую операцию над различными элементами объединенного элемента. Если они обладают одинаковым типом данных, например, таким как целые числа, тогда общая операция относится к операции над целыми числами. Однако, во многих случаях

операции *связаны со структурой*, и при этом не возникает необходимости вовлекать особый тип данных. Это различие станет более понятным, после того как мы изучим теорию типов в Главе 7.

У элемента данных есть область, в которой он видим, и вычисления производятся в этой области. Для того, чтобы убедиться в видимости элемента, язык программирования должен определить части программы с четкими границами, в которых элемент данных является видимым. Существует необходимость установить границы видимости элемента данных, в случае, если мы хотим позволить находиться нескольким элементам данных с одинаковым именем в программе. Иначе, возникнет ошибка из-за совпадения имен, так как одинаковое имя относится к разным элементам данных. Существует два подхода для получения доступа к элементу данных в части программы: (1) присвоение имени элементу данных и (2) определение пределов местонахождения, где он может быть видимым из объявленной позиции. Использование имени облегчает доступ к элементу данных в нескольких местах программного модуля, и пределы местонахождения определяют границу видимости элемента данных.

К элементу данных можно обращаться, используя несколько имен для разных частей программы. Несколько имен, относящиеся к одному элементу данных, называются *псевдонимами*, и в дальнейшем они будут подробно описаны. Когда мы объявляем элемент данных, существует необходимость в шаблоне общей структуры, так как несколько элементов данных могут быть связаны с подобным шаблоном данных. Такие структуры называют *абстракциями данных.*

Для работы подпрограммы над неопределенным количеством элементов данных, обладающих некоторыми хорошо определенными свойствами, нам необходимо понятие *постоянных* и *переменных*. Постоянная - это хранитель значения, которое может быть передано элементу данных во время компиляции и которое, в дальнейшем, не будет изменяться. Переменная отображается в ячейке памяти, которая может быть перезаписана несколько раз в процессе выполнения программы. Переменная может быть анонимно создана компилятором, или во время выполнения программы системой исполнения языка, или указана программистом. Отображение переменной в элементе данных может несколько раз изменяться во время выполнения программы, как, например, при деструктивном присваивании в парадигме императивного программирования, или оно может быть зафиксировано после того как один раз было присвоено значение, как, например, в парадигме

декларативного программирования. Парадигма императивного программирования позволяет программисту изменять по желанию отображение с помощью введения понятия ячейки памяти или блока памяти. Благодаря появлению ячеек памяти, в которых идентификатор↦ ячейка памяти, а ячейка памяти↦ элемент данных, переменные в парадигме императивного программирования становятся *изменяемыми*, и называются *изменяемыми объектами*. Напротив, декларативные языки позволяют программисту преобразовать идентификатор в элемент данных только один раз: идентификатор элемент данных. После того как значение было присвоено, программисту необходимо создать новую переменную, чтобы присвоить новое значение. Объекты, которым можно присвоить значение только один раз, называются *неизменяемыми объектами*.

В современных языках программирования такие классы абстракций данных широко распространены. Все современные языки программирования поддерживают *единичные элементы* и *объединения*. Объединения включает *составные типы*, *наборы* и *расширяемые структуры данных*. Многопарадигматичные языки, которые поддерживают декларативное и импер ативное программирование, различают и поддерживают изменяемые и неизменяемые объекты.

4.1.1 Единичные элементы данных

Абстракция для определения элемента данных связана с необходимостью моделирования реальных объектов и их взаимосвязи с другими объектами. Реальный объект имеет множество атрибутов, а эти атрибуты в свою очередь могут подразделяться. Такие атрибуты могут быть представлены с помощью базовых математических областей, например, целых чисел, чисел с плавающей точкой, символов и строк, или областей вычисления, например, биты, байты и семафоры, или перечисляемых областей, определенных пользователем. Разные предметные области имеют разные типы элементов. Большинство языков обрабатывают строки как индексируемую последовательность символов, а в некоторых языках, например, в Java или C#, “строка” рассматривается в рамках базового объявления типа данных, которая также может обрабатываться как последовательность символов, если это потребуется.

4.1.2 Составные элементы данных

У составного объекта есть больше одного атрибута. В составном

элементе каждое поле может само быть составным элементом данных. Для модели составного объекта используют кортежи. Кортежи могут быть именованными или неименованными. Большинство языков программирования использует именованные кортежи, такие как запись или структ, для моделирования составного элемента данных. Единственное различие между кортежами и записями или структами заключается в том, что записи - это шаблоны последовательностей, обладающие названиями. Преимущество именованных шаблонов кортежей в объявлении типа данных состоит в том, что они могут использоваться для создания нескольких кортежей, связанных с различными переменными для удобства составления программы. Введение наименования заключается в том, что шаблон может повторно использоваться для определения нескольких сложных пользовательских типов. Независимо от синтаксического приема, используемого в разных языках программирования, все определения составных объектов представляются в виде кортежа. Рекурсивное определение кортежа употребляется в модели расширяемого элемента данных. Например, связанный список представляют в виде пары (информационное поле, связанный список); а дерево можно представить как тройку (информационное поле, левое поддерево, правое поддерево).

Пример 4.1

Примером составного элемента является определение “класса” (Рисунок 4.1). Класс - это 6-элементный кортеж из формы (*номер курса, название курса, преподаватель, студенты, место*, *время*). Атрибуты *номер курса*, *название курса* и *преподаватель* - это единичные элементы; *место* и *время* - пары; а атрибут *студенты* - это набор элементов данных.

Как показано в определении “класс*”,* составной элемент данных состоит из наименования элемента данных, количества атрибутов и описания различных атрибутов. Каждый атрибут может быть кортежем, состоящим из абстракций данных, или набором абстракций данных, или единичным элементом. Например, *место* - это абстракция данных, моделируемая в виде пары. Аналогично, *студенты* - это набор *студентов*.

Студент - это 4-элементный (четверной) кортеж формы (*id_студента, имя_студента, название-кафедры, лет в ВУЗе*). Каждое поле - это единичный элемент. Место может быть абстрагировано как пара вида

(*корпус*, *кабинет*). Время может быть абстрагировано как пара вида (*время начала, продолжительность*).

```
абстракция данных: класс
тип-акции: ортеж
размер бута: 6
начать описание-атрибутов
атрибут1: единичный-элемент номер-курса
атрибут2: единичный-элемент название-курса
атрибут3: единичный-элемент преподаватель
атрибут4: набор студентов
атрибут5: кортеж место
атрибут6: кортеж время
завершить описание-атрибутов завершить абстракцию данных
абстракция-данных: студент
тип- стракции: кортеж
размер атрибута: 4
начать описание-атрибутов
атрибут1: единичный-элемент id-студента
атрибут2: единичный-элемент имя-студента
атрибут3: единичный-элемент название-кафедры
атрибут4: единичный-элемент лет-в-ВУЗе
завершить описание-атрибутов завершить абстракцию данных
абстация-данных: место
тип-бстракции: кортеж
начать описание-атрибутов
атрибут1: единичный-элемент корпус
атрибут2: единичный-элемент кабинет
завершить описание-атрибутов завершить абстракцию данных
абстация-данных: время
тип-бстракции: кортеж
начать описание-атрибутов
атрибут1: единичный-элемент время начала
атрибут2: единичный-элемент продолжительность
завершить описание-атрибутов завершить абстракцию данных
```

РИСУНОК 4.1 Абстрагирование элемента данных «класс».

Пары – это специальные типы кортежей, состоящие из двух атрибутов, которые используются в абстракциях математических объектов, таких как комплексное число, рациональное число и временной интервал. Например, комплексное число представляет собой пару вида (действительная часть, мнимая часть), а рациональное число представляет собой пару вида (числитель, знаменатель). Абстрактные понятия, такие как события, могут быть смоделированы как пара (название события, наступление события), где наступление события - это пара вида (время, место). Многие языки программирования, например, ADA, позволяют комплексные и рациональные типы как встроенные

типы. Полиномы представляют собой набор троек, где каждая тройка имеет вид (коэффициент, имя переменной, степень).

4.1.3 Набор элементов данных

Набор элементов данных - это множество элементов и совершенно отличается от составного объекта. В составном объекте каждый атрибут - это часть абстракции одного объекта, и он не индексируется. В наборе или множестве каждый объект - это отдельный элемент, который индексируется. Существует несколько способов моделирования набора элементов данных: (1) последовательность— упорядоченное множество или (2) множество пар вида (ключ, элемент), где ключ - это, индивидуальный идентификатор, относящийся к элементу данных.

В упорядоченном множестве каждый элемент данных соотносится с определенным положением, и может быть представлен в виде индексируемой последовательности, например, в виде массива или в виде вектора.

Последовательности реализуются за счет использования массивов, связанных списков и векторов. Массивы и векторы индексируются. Связные списки не индексируются; чтобы достигнуть i-ого элемента, необходимо сначала пройти $(i - 1)$ элементы. Массивы могут быть *статическими*, *полудинамическими* и *динамическими*. Размер *полудинамического массива* задается в виде параметров из вызывающей подпрограммы и может изменяться в промежутках между вызовами. Динамический массив размещается в памяти, реализуется как рекурсивная структура данных, например, как дерево, и является расширяемой. Например, одним из способов выражения динамического массива является квадродерево (или дерево квадрантов). У квадрадерева есть четыре ветви. Если размер множества находится в интервале от 1 до 4, то требуется дерево одного уровня. Если размер множества находится в интервале $4 + 1$ и 4^2, то требуется двухуровневое дерево, а если размер находится в интервале $4^2 + 1$ и 4^3, то необходимо трехуровневое дерево. Все элементы данных сохраняются в листовом узле. Преимущество такого представления заключается в том, что для расширения множества точное по ложение элемента данных в дереве можно вычислить в любой момент, посмотрев на значение индекса, и это значение может перемещаться в логарифмическом времени, так как дерево растет сбалансированным образом.

Набор пар (ключ, элемент данных) называют *ассоциативными списками* в декларативных языках, и отображается в Scala - современном

мультипрограммном языке, объединяющем функциональное программирование и объектно-ориентированное программирование. Ключ делает элемент данных независимым от частного неявного упорядочивания за счет дополнительной памяти, затрачиваемой на хранение ключа; элементы данных могут размещаться в произвольном порядке в любой ячейке множества. Для получения элемента данных требуется механизм поиска, такой как хэширование или бинарный поиск, который подбирает необходимый ключ и затем извлекает соответствующий элемент данных.

Любой динамический набор элементов данных должен обладать двумя основными свойствами: (1) эффективный поиск элементов данных по значению индекса или ключа и (2) дополнительное расширение размера набора абстракций данных во время выполнения программы для размещения новых элементов данных, которые поступают как через механизм ввода-вывода, так и через программную генерацию. Существует две основных схемы для эффективного поиска элементов данных: (1) хэш-функция размещает ключ в почти постоянном времени для того, чтобы найти значение индекса, а затем индекс будет использоваться для получения данных из соответствующего массива и (2) использование дерева поиска в логарифмическом времени. В зависимости вида иерархического представления, а именно двоичного или третичного дерева, ключ будет находиться с помощью соответствующего алгоритма поиска.

4.1.4 Расширяемые элементы данных

Для моделирования расширяемости используются рекурсивные определения и кортежи. Определенная совокупность данных абстрагируется рекурсивно за счет понятия кортежа, который содержит два поля: элемент данных и связанное поле для неограниченного Пары – это специальные типы кортежей, состоящие из двух атрибутов, которые используются в абстракциях математических объектов, таких как комплексное число, рациональное число и временной интервал. Например, комплексное число представляет собой пару вида (действительная часть, мнимая часть), а рациональное число представляет собой пару вида (числитель, знаменатель). Абстрактные понятия, такие как события, могут быть смоделированы как пара (название события, наступление события), где наступление события - это пара вида (время, место). Многие языки программирования, например, ADA, позволяют комплексные и рациональные типы как встроенные типы.

Полиномы представляют собой набор троек, где каждая тройка имеет вид (коэффициент, имя переменной, степень).расширения совокупности данных во время выполнения программы.

Набор можно моделировать с помощью хэш-таблицы или дерева. Несмотря на отсутствие необходимости упорядочивания ключа или значений в хэш-таблице, иерархическая реализация совокупности должна быть упорядоченной для эффективного логарифмического времени поиска.

Последовательность рекурсивно определяется как *<последовательность>*::= *(<элемент-данных> <последовательность>)* | *ноль*. В таком представлении элементы данных расширяются линейно, и после любого количества расширений накопление может быть прекращено основным сценарием *ноль*. Размер совокупности данных не определен и может расширяться во время выполнения программы. Связные списки, массивы и векторы используются для создания последовательности.

Другой пример рекурсивной абстракции, использующей кортежи, - бинарное дерево, которое может быть смоделировано рекурсивно как *<бинарное-дерево>* ::= *(<элемент-данных>, <бинарное-дерево>, <бинарное дерево>)* | *void*. Левое поддерево и правое поддерево снова представляют собой кортежи, а уровень вложенности кортежа задает уровень дерева. Базовый случай бинарного дерева - «нулевое» дерево. Дополнительный узел может быть добавлен к концевому узлу дерева для расширения первоначального дерева. Например, один узел бинарного дерева имеет вид *(инфо, ноль, ноль)*, где инфо обозначает элемент данных, а *ноль* обозначает «нулевое» дерево; двухэлементная совокупность имеет вид *(инфо, (инфо, ноль, ноль), ноль)*; а трехэлементная совокупность имеет вид *(инфо, (инфо, ноль, ноль), (инфо, ноль, ноль))*.

Рекурсивные структуры данных и расширяемые динамические структуры данных, например, векторы, реализуются с помощью *указателей. Указатели* - это адреса, соединяющие физически разделенные фрагменты памяти. Указатель может направлять на следующий элемент данных или может быть нулевым: не направлять ни на какой объект. Это свойство указателей используется для осуществления расширения в реализациях связанных списков, векторов и деревьев. Дерево расширяется за счет присоединения указателя к концевому узлу для добавления нового элемента данных. Вектор расширяется, сохраняя заголовок, в котором хранится вся информация о последней ячейке. Новые элементы присоединяются к концу вектора. Указатели могут быть

напрямую связаны с рекурсивными структурами данных, как в Pascal. Во многих языках, таких как ADA, ALGOL и C, указатели поддерживаются как независимый элемент, который можно связать с любой структурой данных.

4.1.5 Порядок элементов данных и доступ к элементам данных

В составном объекте значения атрибутов представлены в виде имени поля, например, *<элемент-данных>.<имя-атрибута>*. Если поле составное, то его атрибуты извлекаются как *<элемент-данных>.<имя-атрибута>.<имя-вложенного атрибута>*. Во время компиляции, различные атрибуты отдельного элемента размещаются в смежных ячейках памяти, и генератор команд высчитывает смещение каждого названия атрибута относительно базового адреса элемента данных. Вычисленное смещение добавляется к базовому адресу для осуществления доступа к соответствующему атрибуту.

В совокупности элементов данных значение индекса или ключа играет важную роль для получения доступа к элементу данных. Если совокупность представлена массивом или вектором с фиксированным размером каждого элемента данных, то смещение заданного элемента данных рассчитывается как *размер(отдельный элемент-данных) * значение-индекса*, при условии, что начальное значение индекса ноль. Если совокупность представлена в виде дерева, тогда ключ находится с помощью алгоритма, зависимого от типа представления дерева: двоичного или третичного. Если совокупность изображена в виде хэш-таблицы, тогда доступ осуществляется в почти постоянном времени. Для ключа применятся хэш-функция, в результате чего создается подходящий индекс, и объекты данных, связанные с этим индексом просматриваются последовательно для согласования ключа с идентификацией соответствующего элемента данных.

4.1.6 Взаимосвязанные элементы данных

Внутри одного объединенного элемента данных могут находиться несколько элементов данных. Объединенный элемент данных можно представить в виде кортежа (составного элемента данных), совокупности элементов данных или расширяемого элемента данных. Одно из интересных объединений - используемая в зыке Lisp *семантическая сеть*, представляющая собой граф элементов, соединенных связями. Каждый элемент изображается в виде *фрейма* - пары (имя объекта, список свойств), где список свойств - это совокупность пар (название атрибута,

значение). Значение может быть выражено отдельным элементом, объединением, вызовом функции вычисления значения или ссылкой на другой элемент данных. Семантическая сеть представляет собой интересную абстракцию, так как ее можно использовать для создания модели отношений между несколькими реально существующими объектами. Пересекая граф по соединительным реляционным дугам между объектами, находят атрибуты других взаимосвязанных объектов.

4.1.7 Персистентность

Персистентность связана с двумя важными свойствами: видимость во всех программных блоках и время жизни, превышающее время жизни программы. Преимущество персистентности заключается в том, что разные блоки программы способны обмениваться информацией и результатами вычислений, которые возможно были сделаны другими блоками программы в другое время. Персистентность может относиться к двум типам элементов данных: (1) глобальные переходные элементы данных, время жизни которых ограничено временем жизни программы, и (2) постоянные элементы данных, время жизни которых превышает время жизни программы, к ним относятся базы данных, файлы и заархивированные объекты.

Доска объявлений или пространство кортежей - это глобальный переходный элемент данных, состоящий из совокупности пар (ключ, значение), где ключ - уникальный идентификатор, заданный пользователем. Операции с доской объявлений: извлечение значения с помощью ключа; удаление пары (ключ, значение); обновление значения, привязанного к ключу; введение новой пары (ключ, значение).

Файлы и базы данных - это постоянные элементы данных, в которых хранится информация, используемая даже после завершения программы, или информация, которую можно передать в другие программы. Файлы - это последовательность символов, сформированная линеаризацией элементов данных. Все свойства последовательностей, которые мы изучали ранее, применяются и к файлам. Кроме того, файл - это логический элемент, который может быть открыт для чтения или записи. Тем не менее, файл - это огромная последовательность, и обработка данных в оперативной памяти после прочтения всех элементов файла вызывает колоссальные расходы памяти и задержку в загрузке данных.

В большинстве случаев программному модулю требуется часть данных из входного файла, которые он обрабатывает, а потом повторяет процесс с оставшимися данными следующим образом

```
my-sequence = file;
while (not is _ empty (my
    _ sequence)) { data-
    entity = first(my-
    sequence); process
    data-entity;
    my _ sequence = rest(my-sequence);}
```

Так как оставшиеся данные из файла не обрабатываются одновременно с текущим элементом данных, то нет необходимости держать все элементы данных в рабочей области в тот момент, когда обрабатывается только первый элемент данных; второй элемент данных будет извлечен из файла, когда будет обработан первый элемент, а третий - когда второй будет обработан. Такие ресурсы как память и процессор, используемые предыдущим элементом данных, будут использоваться и для последующего элемента данных.

Поток данных - это абстракция данных, позволяющая повторно применять имеющиеся ресурсы без потери памяти. Поток - это последовательность, частично доступная по времени. Если поток имеет вид $<d_0, d_1, d_2, \ldots d_N>$, то в определенный промежуток времени в памяти будет доступна только подпоследовательность $<d_I, \ldots d_J>$. Если K элементов данных обрабатываются одновременно, тогда в следующий промежуток времени будет доступна подпоследовательность $<d_{I+K}, \ldots d_{J+K}>$.

Абстракция потока важна при обработке большого количества данных, особенно в сетевом программировании, мультимедийном программировании и мобильных приложениях, где видео и аудио файлы имеют большой объем и не могут обрабатываться на месте из-за задержек в каналах связи и большого количества вычислений, связанных с воспроизведением видео или аудио файлов. В таких случаях поток данных поступает непрерывно. Подпоследовательность данных обрабатывается в тот момент, когда следующая подпоследовательность данных передается по каналам связи.

4.1.8 Объявления и изменение среды

В объявлениях возможно наличие следующих пяти компонентов: привязка идентификатора к постоянной; макроопределение — привязка идентификатора к блоку текста; объявление пользовательского типа - привязка идентификатора к абстракции данных, содержащей значения

типа данных; объявление переменной - привязка идентификатора к памяти; объявление секвенсора - метки, используемой для перехода. Объявление также используется для связывания идентификатора с процедурой или функцией. В императивных языках объявление меток применяется для перехода, а объявление переменных не оказывает влияния на последующие объявления. Тем не менее, привязка идентификатора к константе, объявление определяемых пользователем типов и объявление макроопределений могут быть использованы в последующих определениях для создания более сложных объявлений.

Объявление переменной изменяет среду за счет связывания идентификатора с соответствующим l-значением. Вычисление можно представить в виде тройки (σ^E, σ^S, σ^D), где σ^E - среда, σ^S - блок памяти, σ^D - дамп — набор вызывающих процедур. При такой модели вычислительное состояние постоянно меняется, если (1) объявление изменяет среду, (2) выполнение оператора присваивания изменяет память, (3) дамп изменяется при вызове процедуры. При передаче параметров и среда, и память могут изменяться, так как формальные параметры отражаются в ячейках памяти, а ячейки памяти инициализируются в значение. Команда выполняется в среде, созданной объявлениями.

У последовательности объявления вида D_1 <*разделитель*> D_2 возможно наличие двух разных значений: последовательного и параллельного.

1. В *последовательных* объявлениях D_1 обновляет среду σ^E до σ'^E, а D_2 использует в своем определении уже новую среду σ'^E, чтобы перехватить совокупность переменных, связанных в объявлении D_1. Например, в языке функционального программирования *Lisp*, объявление (*let* ((X 4) (Y (+ X 4))))* соответствует связыванию переменной *X* со значением 4 и связыванию переменной *Y* со значением 8, потому что первое объявление становится частью среды, а когда объявляется *Y*, то оно перехватывает из среды связывание *X* и добавляет 4 к значению *X*, в результате чего получается значение 8. После того как *X* было привязано к 4 среда $\sigma'^E = \sigma^E \uplus \{X \mapsto 4\}$, а после второго объявления среда становится $\sigma^E \uplus \{X \mapsto 4\} \uplus \{Y \mapsto 8\}$.

1. В *параллельных* объявлениях и D_1, и D_2 перехватывают привязки из предыдущей среды σ^E, а D_2 не изменяется под воздействием D_1, как это показано в Примере 4.2.

Пример 4.2

Давайте снова разберем пример из языка программирования Lisp.

(**defun** my_square(b) % определить функцию
(**let** ((b (+ b 5)) (c (+ b 6))) % параллельное объявление (+ (* b b) (* c c)) % вычислить $b^2 + c^2$
% закрыть область let объявлений
) % закрыть область объявления функции

Конструкция let создает локальную среду для локальных переменных *b* и *c*. Вызов функции my_square*(4)* вычисляет значение *b* как значение(*b*) + 5 = 9, а значение c как *b* + 6 = 4 + 6 = 10. Хотя значение *b* в локальной среде изменяется на 9, во второй инициализации берется предыдущее значение b = 4, что вызвано *let конструкцией* семантики параллельного присваивания. Окончательный ответ: $9^2 + 10^2 = 181$.

При вызове процедуры или функции происходит изменения среды, так как большинство связываний, локальных для вызывающей процедуры, скрыты, то есть становятся невидимыми в пределах области видимости вызываемых подпрограмм. И в новой среде вызываемой подпрограммы сохраняются только связывания глобальных переменных, нелокальных переменных, импортированных переменных и связывания формальных параметров.

4.2 Абстракции управления

Команды, управляющие данными, абстрагируются в зависимости от парадигмы программирования. Вне зависимости от парадигмы программирования команды можно подразделить на: (1) *конструкторы* — создают новый элемент данных; (2) *мутаторы* — команды, изменяющие значения, к которым привязан идентификатор; (3) *селекторы* — команды, производящие выборку элементов данных из составного объекта данных или совокупности объектов данных; (4) *команды условий* — команды, которые выбирают один из вариантов в зависимости от условия; (5) *итераторы* — команды, циклично выполняющие одну и ту же операцию над совокупностью объектов данных в организованном порядке; (6) *вычислители* — команды, вычисляющие выражение; (7) *секвенсоры* — команды, осуществляющие управление через переход к той или иной метке; (8) *команды вызова* — команды, которые активируют функцию или процедуру для вычисления какого-нибудь значения.

Мутатор, как и оператор присваивания, изменяет блок памяти за счет преобразования значений, сохраненных в ячейке памяти. В некоторых языках блок памяти также изменяется при инициализации со значением при создании или объявлении нового элемента данных. В случае последовательности мутаторов, каждый мутатор постепенно преобразует имеющийся блок памяти в новый. Блок памяти изменяется, когда значение текущего параметра связано с ячейкой памяти, относящейся к формальному параметру, или, когда переменная при объявлении инициализируется в конкретное значение.

Последовательность команд может изменяться командами условий, операторами перехода или итераторами. Последовательность команд, следующая за выполняемой командой во время выполнения программы, называется *продолжение. Продолжение* необходимо для объяснения причины поведения программы и является неотъемлемой частью определения денотационной семантики в реалистичных языках программирования. У продолжения функции выбора, например, условного оператора, есть несколько вариантов, один из которых будет выбран во время выполнения программы после оценки условия. Аналогично, продолжение итерационного оператора включает в себя многократное разворачивание цикла при условии определения истинности условия. Продолжение итеративного цикла или условных операторов сложно предсказать, так как оно зависит от результата оценки условия.

Пример 4.3

Давайте рассмотрим продолжение предложенной программы.

```
x = 4; z = 6;
goto L;
z = 8;
L: y = 5;
while (z > 4)
    {x = y + 5; z = z - 1;}
```

Продолжение оператора $x = 4$ is {$z = 6$; $y = 5$; **if** $z > 4$ **then exit**; $x = y + 5$; $z = z - 1$; **if** $z > 4$ **then exit;** $x = y + 5$; $z = z - 1$; **if** $z > 4$ **then exit**}. Обратите внимание, что секвенсор “**goto** L” изменяет порядок

выполнения программы на отличный от текстуального порядка. Аналогично, операторы в цикле WHILE выполняются два раза, пока *Z* не станет равным 4, и затем управление выходит из цикла WHILE. Таким образом, продолжение зависит от вычисления, и его трудно предсказать, просто посмотрев на программу.

4.2.1 Присваивание и последовательность команд

Оператор присваивания связывает ячейку памяти с идентификатором для определения выражения, записанного в правой части оператора присваивания. Связывание осуществляется после определения. В общем случае присваивание состоит из одного связывания, что означает, что оператор имеет вид <*идентификатор*> = <*выражение*>. Тем не менее, в некоторых языках, например, таких как ALGOL-68, C++, Python и Ruby, поддерживается *цепное присваивание* (или *множественное присваивание*), которое имеет несколько переменных, связанных с определением одного и того же выражения. В синтаксисе современного языка Ruby, поддерживающего функциональное и объектно-ориентированное программирование, множественное присваивание записывается как:

x = y = 4 + 5 + 6 (поддерживается в Python и Ruby) x, y = 4 + 5 + 6 (поддерживается в Scala)

Оператор выполняет вычисление выражения "4 + 5 + 6" и связывает значение 15 с ячейками памяти переменных *x* и *y*.

В некоторых языках, например, Ruby, C++, Python, Perl и Lua, поддерживается *одновременное присваивание* или *параллельное присваивание.* При одновременном присваивании значения соответствующих выражений одновременно присваиваются нескольким отдельным несвязанным переменным, а вычисление выражения принимает значения исходной ячейки. Основное требование к одновременному присваиванию заключается в том, что переменные должны быть записаны в отдельные ячейки памяти. Одновременное присваивание имеет следующий вид:

$var_1, var_2, var_3, \ldots, var_N = exp_1, exp_2, exp_3, \ldots, exp_N$

Семантика записанного выше оператора состоит в том, что каждое выражение exp_i ($1 \le i \le n$) вычисляется относительно исходного бока памяти σ^S, а вычисленные значения связываются с ячейками памяти,

соответствующими переменным var_i ($1 \leq i \leq n$). Например, в синтаксисе языка Interactive Ruby следующее множественное присваивание будет связывать x1 с 4, x2 с 5 и x3 с 6. Тем не менее, если две переменные являются псевдонимами, тогда блок памяти содержит последнее связывание.

```
x1, x2, x3 = 4, 5, 6 % x1 ↦  4; x2 ↦  5; x3 ↦  6
y, y = 8, 9 % y ↦  9
```

Последовательность операторов присваивания вида C_1; C_2 в зависимости от парадигмы языка программирования способна последовательно или параллельно изменить блок памяти. При *последовательном изменении* блока памяти предыдущий оператор присваивания первым вносит изменения в блок памяти, а последующие операторы присваивания продолжают работу уже с измененным блоком памяти. В парадигме параллельного программирования при *параллельном изменении* блока памяти два или более оператора присваивания параллельно работают с одним и тем же исходным блоком памяти. Должно производиться изменение разных ячеек памяти; чтение или запись одной и той же совокупности ячеек памяти вызовет *состояние гонки* - состояние, при котором блок памяти несогласован и может выдавать различные значения, если один и тот же набор операторов производит вычисления несколько раз. Состояние гонки подробно описано в Главе 7.

Пример 4.4

Например, последовательное присваивание вида: "$x = 4$; $y = x$;" сначала разрушит блок памяти σ^S, чтобы создать новый блок памяти $\sigma'^S = \sigma^S \oplus (x \mapsto 4)$. Обратите внимание, что здесь символ "↦" обозначает деструктивное обновление в парадигме императивного программирования. Теперь оператор присваивания "$y = x$" прочитает значение переменной x из блока памяти σ'^S и создаст новый измененный блок памяти $\sigma''^S = \sigma'^S \oplus (y \mapsto 4) = \sigma^S \oplus (x \mapsto 4) \oplus (y \mapsto 4)$.

В языках программирования, поддерживающих парадигму императивного программирования, оператором присваивания является *мутатор*. Это означает, что идентификатор связан с ячейкой памяти, которая может быть деструктивно обновлена. Как говорилось ранее, преимущество разрушительного обновления заключается в повторном

использовании памяти за счет потери результатов предыдущих вычислений и побочных эффектов - неблагоприятных режимов программы, которые могут нарушить основные математические принципы, важные для программирования.

Парадигма декларативного программирования использует свойство однократного назначения и присваивает переменное значение выражения только один раз, и привязку невозможно разрушить явно. Тем не менее, некоторые языки программирования, например, Prolog, поддерживают перебор с возвратами - возвращение к предыдущему вычислению в поисках варианта решения. Во время выполнения перебора с возвратами машина базовой реализации в Prolog может отменить привязку переменной от предыдущего значения.

Вариантом оператора присваивания в логическом программировании служит понятие *унификации*. В унификации два логических терма приравниваются. Логический терм - это составная структура, содержащая переменные, постоянные, литералы и другие внедренные логические термы. При приравнивании не происходит оценка, так как термы не являются выражениями. Вместо этого, логические термы в левой части и логические термы в правой части сопоставляются позиция за позицией. Если один из термов - переменная, тогда каждое появление такой же переменной в обоих логических термах привязывается к соответствующему логическому терму за счет свойства однократного назначения в парадигме декларативного программирования. Если оба соответствующих терма - литералы, тогда они сопоставляются буквально.

Например, при унификации терма $X + 4 + 3$ с термом $5 + Y + 3$ получаем $5 + 4 + 3$, переменная X связывается с литералом 5, переменная Y связывается с литералом 4, а литералы 3 и 3 успешно сопоставляются. Напротив, унификацию $X + 4 + 3$ и $5 + Y + 2$ осуществить невозможно, так как соответствующие литералы 3 и 2 не сопоставимы.

Унификация будет подробно изучена в логическом программировании в Главе 10. Тем не менее, здесь она была упомянута для более полного представления о понятии условного оператора.

4.2.2 Условные операторы

Условные конструкции имеют практически одинаковое строение в языках программирования. Базовая конструкция условного оператора представлена следующим правилом абстрактного синтаксиса:

```
<условный оператор> ::= if '('<условие>')' then <оператор>
                                                [else
                                                <оператор>];
<условие> ::= <условие> && <условие> | <условие> ||
                    <условие> |not <условие> |
                    <арифметическое-выражение> <с-операция>
                    <арифметическое-выражение> |
                    true | false
<с-операция> ::= '>' | '<' | '>=' | '=<' | '=='
```

В разных языках применяются разные зарезервированные слова. Например, вместо '.**and**' может стоять '**&&**' или вместо '**.or**' - '||'. В условном операторе (if-then-else) оператор then и оператор else равновероятны, и изменения порядка выполнения после проверки отрицания условия будет функционально эквивалентным первичной форме условного оператора (if-then-else); это значит, что if (*<условие>*) then *<оператор-then>* else *<оператор-else>*" функционально равнозначен if (not *<условие>*) then *<оператор-else>* else *<оператор-then>*, поскольку *<условие>* и not *<условие>* требуют взаимного исключения.

Как оператор then, так и оператор else могут быть любым оператором, включающим другой условный оператор (if- then-else). В Главе 3 было указано, что в случае вложенного условного оператора часть else всегда относится к ближайшему оператору then.

Другой конструкцией отбора является *оператор выбора*. Структура оператора выбора имеет вид

```
case (<выражение>) of :
    <набор-значений_1>: <последовательность-команд_1>;
    ...
    <набор-значений_N>: <последовательность-команд_N>;
    otherwise: <последовательность-команд_N+1>
end case
```

В операторе выбора вычисляется выражение. Возможен многозначный результат, и также возможно взять различную последовательность

действий для каждого выходного значения или области выходных значений. Выполняются все варианты, обладающие различной последовательностью действий, а в конце находится оператор catch-all, который применяется, если все другие варианты не выполнимы. Обратите внимание, что оператор выбора является детерминированным, как только сделан один выбор в основе оценки выражения.

Lisp использует набор условных операторов следующего вида:

```
(cond   ((<предикат_1> <выражение_1>)
          …
         (<предикат_N>
         <выражение_N>)
         (t <catch-all-
         выражение>))
)
```

Такая функция проверяется сверху вниз; если любой из *<предикат$_i$>* является истиной, тогда выполняется соответствующая функция; иначе выполняется *<catch-all-выражение>*.

Другая условная конструкция, применяемая в некоторых функциональных языках, например, в Lisp, - *конструкция WHEN*, записывается как *(***when** *<предикат> <выражение>)*. Конструкция определяет выражение *<выражение>*, если предикат *<предикат>* является истиной. Другая условная конструкция - конструкция *unless* в языке программирования Ruby. У нее точно такая же семантика, как и у “if not.” Конструкция выглядит следующим образом:

unless ‘(‘*<условие>*’)’ *<последовательность-команд$_1$>* **else** *<последовательность-команд$_2$>* эквивалентна

if not ‘(‘*<условие>*’)’ *<последовательность-команд$_1$>* **else** *<последовательность-команд$_2$>*

4.2.3 Итерационные конструкции и итераторы

Итерационные конструкции подразделяются на три основные категории: (1) определенная итерация, (2) неопределенная итерация и (3) итераторы. И определенная, и неопределенная итерации предполагают, что совокупность элементов данных составлена однозначно с использованием индексируемых структур данных, таких как массивы

и векторы. Напротив, итераторы обрабатывает элемент данных как список, таким образом, что отдельные элементы и механизмы доступа к отдельным элементам могут быть недоступны программисту. Итератор извлекает следующий элемент из списка и выполняет операции над каждым элементом списка по порядку. Все три класса итеративных конструкций имеют точки однократного входа и однократного выхода, за исключением случаев, когда выполнение программы прерывается из-за ошибки или исключения. Определенные итерации не позволяют программисту вносить изменения в параметры цикла, такие как начальное связывание, конечное связывание, значение индекса переменной и величина шага, во время выполнения блока операторов. Все они наперед зафиксированы. Тем не менее, машина реализации автоматически меняет значение индекса переменной на величину шага при переходе на следующий итерационный цикл. Абстрактный синтаксис для определенной итерации имеет вид:

```
for (<индексируемая переменная> = <начальное выражение>';'
<конечное-выражение>';' <шаг выражения>)
        <блок>
```

В конструкции, расположенной выше, начальное выражение <*начальное выражение*> задает начальное значение, конечное выражение <*конечное выражение*> задает конечное значение, а шаг выражения<*шаг выражения*> задает шаг итерации. Выражения оцениваются до входа в цикл, в циклах содержится только один вход и обычно один выход, за исключением ситуаций, вызывающих прерывание цикла или выход из него. Определенная итерация может брать индексируемое значение из списка выражений. Конструкция неопределенной итерации принимает следующий вид:

```
for '('<индексируемая-переменная> in <список-выражений>')'
                        <блок>
```

Определенная итерация может также подобрать показатель из неупорядоченного множества. Преимущество использования множества заключается в том, что индекс может появиться в любом порядке, и у него не обязательно должна быть структура. Конструкция определенной итерации имеет следующий вид:

for ‘(‘<*индексируемая-переменная*> **in** <*множество*>’)’
<*блок*>

В неопределенной итерации встроенный блок операторов будет продолжать вычисление, пока не будут достигнуты заданные условия. В отличие от определенной итерации, где значение индекса постоянно изменяется в заранее определенном порядке, в неопределенной итерации условием может быть любой предикат. Более того, значения переменных, связанных с предикатом, могут без ограничений изменяться во встроенном блоке операторов. Благодаря этому свойству неопределенные итерации могут образовать бесконечный цикл.

Как было установлено в Главе 1, существует два типа неопределенных итераций, функционально эквивалентных модели любой задачи, которая требует повторяющейся обработки абстракций данных. Неопределенными итерациями являются цикл WHILE и цикл DO-WHILE, также называемый циклом REPEAT-UNTIL. Абстрактный синтаксис цикла WHILE:

<*цикл while*>**::= while '('**<*условие*>**')' '{'**<*блок-операторов*>**'}'**

Абстрактный синтаксис цикла REPEAT-UNTIL:

<*цикл repeat-until*> ::= **repeat** <*блок-операторов*> **until** <*условие*> |
do <*блок-операторов*> **while** <*условие*>

Как было описано в Главе 1, цикл WHILE выполняет встроенный блок <*блок-операторов*> ноль или более раз, а цикл REPEAT-UNTIL выполняет встроенный блок <*блок-операторов*> как минимум один раз. Конструкция REPEAT-UNTIL и конструкция DO-WHILE равнозначны и достигают одного и того же результата с одним лишь различием: конструкция REPEAT-UNTIL циклически выполняет блок операторов до тех пор, пока условие не станет истиной, в то время как синтаксис DO-WHILE циклически выполняет блок операторов, пока условие является истиной. Обобщенный итерационный цикл с несколькими выходами используется в языке ADA. Абстрактный синтаксис итерационного цикла с несколькими циклами:

< *итерационный цикл с несколькими выходами*> **::= loop** {<*условный-выход*>’*;*
<*последовательность-команд*>}*

end-loop
<*условный-выход*> **::= if** <*условие*> **then exit**;

Такой абстрактный синтаксис означает, что существует несколько условий выхода, находящихся внутри блока операторов цикла, и если условие не выполняется, то управление покидает цикл. Одновременно с тем, что цикл удобен некоторым программистам, в 1970х годах было установлено, что функциональная способность вложенного цикла WHILE и цикла DO-WHILE в сочетании с логическими переменными равнозначна циклам с несколькими выходами. Следовательно, они не добавляют никаких дополнительных программных способностей. Для того чтобы избежать явного использования индекса, который используется только в массивах и реализациях на основе вектора, *итераторы* поддерживают *абстракции на основе списка* и *набора*. Декларативные языки, поддерживающие программирование на основе списка, или языки, использующие расширяемые абстракции данных, применяют итераторы. Итератор совершает шаги в расширяемом наборе элементов данных, собирает следующий элемент данных и выполняет некоторые операции над элементом данных. Процесс повторяется для каждого элемента данных из набора. Основное преимуществом использования итератора заключается в том, что не требуется наличие явного индекса, а детали уровня реализации абстракций данных скрыты от программиста; единственное ограничение - это извлечение следующего элемента в следующем цикле.

Абстрактный синтаксис для итератора имеет вид:

<*итератор*> **::= foreach** <*переменная*> **in** <*упорядоченный-набор*> <*блок*>

В функциональных языках <*блок*> замещается на определение функции, применяемой к следующему элементу данных. Недостатками базовой схемы итераторов является то, что итератор пошагово проходит каждый элемент абстракции данных, что не похоже на итеративные конструкции, где индекс используется для выборочного доступа к элементам. Языки C++ и Java используют итератор как изменяемую структуру данных, которая может поочередно выдавать значения данных и имеет возможность проверить существование следующего элемента. Конструкция итератора в Java будет выглядеть так:

```
for (iterator i = data-object.iterator(); i.hasNext();)
                    … visit
                  i.next() …
```

4.2.4 Блочная структура

Блоки встроены в программный модуль. В отличие от программных модулей, блоки не привязываются к идентификатору и не могут быть вызваны простым вызовом. Блочные структуры в общем случае имеют *один вход* и *один выход*, а также обладают специальными функциями. Произвольное группирование операторов не рекомендуется во избежание ухудшения структуры и, как следствие, восприятия. Блочное структурирование позволяет только ограничивать количество операторов "go to" для выхода из вложенных блоков или для управления исключениями при обработке ошибок.

Блоки включают набор объявлений (возможно, нулевой), за которым следует набор команд (возможно, нулевой). Область видимости объявлений находится в пределах блока, это означает, что все переменные и абстракции данных видимы только внутри границ блока, отмеченных зарезервированными словами, например, **begin** … **end** или фигурными скобками. Эти зарезервированные слова зависят от применяемого языка программирования. В блок может быть вложено несколько блоков. Вложенные блоки попадают под действие внешнего блока, это означает, что объявления во внешнем блоке, не имеющие противоречий с объявлениями внутреннего блока, видимы во внутреннем блоке. В случае конфликта наименований, объявления внутреннего блока имеют преимущество, а соответствующие объявления из внешнего блока скрываются во внутреннем блоке. Вложенные одноуровневые блоки (и их потомки) внутри внешнего блока не обмениваются средой. Уровни вложения блоков можно представить в виде дерева, корневой узел - программный модуль, а вложенные блоки - узлы-потомки. Свойство разделения сред одноуровневых блоков, используемое в памяти, многократно применяется во время выполнения. Блоки (и их потомки), определяемые после одноуровневых блоков (и их потомков) могут относиться к той же области памяти, если область видимости одноуровневого блока, определенного ранее, будет закрыта до выполнения следующего одноуровневого блока.

Пример 4.5

Рассмотрим следующую структуру вложенного блока:

```
{объявление A
        …
        {объявления B
                …
                {объявления C
                        …}
                        {объявления D
                        …}
                …
}
        {объявления E
… }
}
```

Соответствующее графическое представление области объявлений и соответствующая структура дерева представлены на Рисунке 4.2.

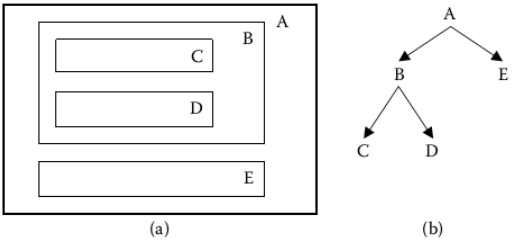


РИСУНОК 4.2 Вложение (a) блочной структуры и (b) ее представление в виде дерева.

Блоки B и E одного уровня, они вложены в блок A, а блоки C и D вложены в блок B. Так как B и E - блоки одного уровня, а блоки C и D вложены в блок B, то блоки B, C и D не делятся своими локальными средами с блоком E. Аналогично, блоки C и D не делятся своими локальными средами. Вследствие иерархичности среда блока B включает

среду, объявленную в блоке A. Аналогично, блоки C и D наследуют среду, наследованную блоком B, и локальную среду блока B. Ячейки памяти, выделенные для блока B, и вложенных в него блоков находятся в совместном пользовании с ячейками памяти блока E и вложенных в него блоков.

Детали будут разобраны в Главе 5, посвященной моделям реализации языков программирования.

4.2.5 Программные модули и вызовы

Программные модули - это группа объявлений и команд, которые привязаны к идентификатору и могут быть вызваны во время работы другой программой с помощью идентификатора в программе, к которой привязывается модуль. Это два свойства, которые отличают программный модуль от блоков.

Большинство языков программирования, например, семейство языков Modula, предусматривают вложенные объявления программных модулей. Свойство обмена среды при вложении аналогично блокам: вложенные программные модули наследуют локальную и наследуемые среды своих предков, а одноуровневые блоки не делятся своими локальными средами.

Абстрактный синтаксис (в РБНФ) программного модуля в блочно-структурированном языке представлен в Главе 3. Ограниченный вариант правил абстрактного синтаксиса представлен ниже

```
<программный-модуль> ::= (program | function | procedure)
<идентификатор>
   <блок>
<блок> ::= '{' [{<объявление>}*] [{<команда>}*] '}'
<объявление> ::= <секвенсор-объявлений>| <объявление-типов> |
<объявление-переменных>
<команда> ::= null | <присваивание>| <условный> |
                <определенная-итерация> | <неопределенная
                итерация> |
                  <итератор> |
                <вызов-подпрограммы>|
                '{'<блок>'}' |
                <секвеносоры>
```

Записанный выше абстрактный синтаксис указывает на то, что программный модуль может быть и главной программой, и функцией, и процедурой, связанной с идентификатором. Тело программы представляет собой структурированный блок. В разных языках используются различные зарезервированные слова. Например, в Lisp используется зарезервированное слово “defun” для определения функции, а в Scala и Ruby для определения функции используется зарезервированное слово “def”.

Для решения сложной задачи один программный модуль может вызывать другой программный модуль. С помощью ссылки на программный модуль внутри итерационной конструкции или рекурсивного вызова можно несколько раз вызвать программный модуль, ссылка программы на саму себя - это абстракция команды и часть последовательности команд внутри программного модуля. Компиляция программного модуля не зависит от количества ссылок; каждый программный модуль осуществляется только один раз за компиляцию или JIT компиляцию. В JIT компиляции программный модуль после первого выполнения кэшируется для дальнейшего извлечения.

Обращения к программным модулям можно изобразить в виде ориентированных графов с направлениями, показывающих структуру обращения к программным модулям. Выполнение программы выглядит как граф, если этот программный модуль будет вызван из нескольких программных модулей. Граф бывает ацикличным и цикличным. Ацикличный граф показывает, что программные модули вызываются нерекурсивно, а циклический граф показывает, что программные модули вызываются рекурсивно.

Каждый вызов, сделанный программой, обладает отдельной собственной средой, отличной от предыдущих вызовов того же программного модуля, и новый вызов работает в своей собственной среде. Как было описано в Главе 2, среда - это совокупность объявлений глобальных переменных, нелокальных переменных - переменных, объявленных в родительских программных модулях, если текущий программный модуль вложен, локальных объявлений и ссылочных параметров.

4.2.5.1 Саморекурсивные процедуры

Рекурсивная программа является *саморекурсивной*, если она напрямую вызывает саму себя. Например, определение функции *factorial* саморекурсивное. Предположим, что внутренний вызов P_0;

тогда следующий вызов будет P_1, P_2, P_3 и т.д. Вызов прекратится, когда базовый случай будет определен, и результат будет передан назад в порядке, обратном порядку поступления. В графовой модели вызова процедуры длина вызова саморекурсии равна нулю, так как процедура вызывает саму себя. Обратите внимание, что среды различных вызовов саморекурсивных процедур отделены друг от друга.

Пример 4.6

Например, обращение к factorial(4) начнет выполнение цепочки вызовов: factorial(4) → factorial(3) → factorial(2) → factorial(1) → factorial(0). Каждый вызов обладает своей средой, своим блоком памяти и своим дампом, которые включают среду и блок памяти предыдущего вызова, добавленного к дампу предыдущего вызова. Дамп представляет собой стек, а предыдущий вызов добавляется к стеку с помощью абстрактной отправки данных среды и блока памяти в стек. Обратите внимание, на употребление выражение «абстрактная отправка данных». На самом деле, когда создается дамп, часть предыдущей среды может быть извлечена с помощью адресной ссылки, называемой *динамической ссылкой*. Тем не менее, обсуждение соответствующего применения будет описано в Главе 5.

После завершения вызова *factorial(0)*, полученный результат возвращается назад в обратном порядке, обновляя блок памяти предыдущих вызовов. После получения результата соответствующая среда и стек вызова удаляются из дампа, а управление передается обратно следующей инструкции предыдущего вызова.

4.2.5.2 Взаимно рекурсивные процедуры

Программный модуль может поддерживать *взаимную рекурсию*. Это означает, что программный модуль начинает цепочку вызовов одного или нескольких программных модулей, кроме себя, и в конце последний вызов в последовательности снова вызывает первый программный модуль. Математически цепочку вызовов можно представить как P_0, P_1, … P_N ($N > 0$); следовательно программный модуль $P_i (0 < i < n)$ не равен программному модулю P_0 или программному модулю P_N, а программный модуль P_0 равен программному модулю P_N. Есть четыре типа структур: (1) иерархическая структура – наподобие дерева; (2) структура как направленный ациклический граф; (3) цикл нулевой длины, обозначающий саморекурсию, и (4) цикл ненулевой длины,

обозначающий взаимную рекурсию.

Пример 4.7

Есть три типа структур: (1) иерархическая структура – наподобие дерева, (2) цикл нулевой длины, обозначающий саморекурсию, и (3) цикл ненулевой длины, обозначающий взаимную рекурсию. Пример графа вызовов процедур показан на Рисунке 4.3. Подпрограмма верхнего уровня вызывает две подпрограммы B и C в указанном порядке. Подпрограмма B вызывает две подпрограммы D и E. Обе подпрограммы D и E формируют терминальные узлы, что означает, что они не вызывают никакие другие подпрограммы. Подпрограмма C вызывает подпрограммы F и G. Подпрограмма F - саморекурсивная, что отмечено кругом; подпрограмма G - взаимно рекурсивная, так как G вызывает H; а подпрограмма H вызывает обратно подпрограмму G. Длина цикла во взаимно рекурсивной подпрограмме равна 1. Обратите внимание, что вызывающий шаблон программных модулей сильно отличается от вложенной структуры программных модулей. Вызывающие шаблоны программных модулей являются свойствами режима выполнения, в то время как вызывающие шаблоны вложенной структуры являются свойствами статической программы.

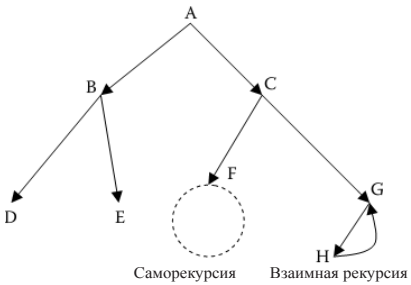


РИСУНОК 4.3 Вызывающий шаблон программных модулей.

4.2.6 Модули

Модуль - это абстракция, обеспечивающая логическую инкапсуляцию вокруг объявления, моделирующую одну или несколько абстракций данных и соответствующих подпрограмм (или методов в терминологии объектно-ориентированного программирования), выполняющий какие-

то важные операции над абстракциями данных. Объявления могут быть (1) *импортирующими* информацию или функцию из другого модуля; (2) объявления констант, или пользовательских типов, или переменных; или (3) определения процедур, функций, объектов и интерфейса для оставшейся системы или других встроенных модулей. Определение модуля обеспечивает явную логическую границу, которая не пропускает внешние объявления в локальную область. У объявления внутри модуля есть естественная область видимости, и в случае *экспортирования* оно не видимо вне модуля. Модуль можно заменить на другой равнозначный модуль с таким же интерфейсом без изменения семантики системы.

Явные модули обычно включают в себя все общие операции для управления абстрактным типом данных, которые можно вычислить или сохранить для дальнейшего использования в виде библиотеки. Когда модуль загружается или импортируется в другой модуль или программу, тогда экспортируемые абстрактные типы данных и все их библиотеки также становятся доступными. Основная цель абстракции модуля заключается в обеспечении (1) систем с повторными наименованиями; (2) независимости от остальной системы; (3) регулируемых взаимодействий с другими составляющими системы через интерфейс модуля с помощью механизма импорта-экспорта описываемых элементов, таких как переменные и подпрограммы.

В отличие от подпрограмм в императивном языке или механизма наследования во многих объектно-ориентированных языках интерфейс строго регулируется механизмом *экспорта-импорта*. Элемент, объявленный внутри модуля, не доступен вне модуля, не считая явного *экспорта*, и другой модуль не может использовать объявленный элемент и явно экспортируемый элемент из других модулей, если только он не *импортирует* его напрямую. Существует два подхода импорта информации из модулей. Первый метод предполагает импорт каждого элемента, экспортируемого из обозначенного модуля. Второй метод заключается в экспорте определенного списка элементов из обозначенного модуля. Механизм экспорта-импорта модулей обеспечивает лучшее регулирование, чем нелокальные переменные и механизм наследования, описанный в подразделе 4.3.5. Внутри модуля информация между подпрограммами и объектами передается посредством локальных переменных, глобальных переменных и параметров передачи.

Модуль можно разработать, компилировать и архивировать независимо от других частей системы. Абстрактный синтаксис для модуля представлен ниже

```
<модуль> ::= module <идентификатор>
             [export {<программный-модуль>}*]
             [{import {<программный-модуль>}* from
             <идентификатор>}*]
             {<объявление>}*
             {<программный-модуль>}*
             {<модуль>}* % определение вложенных модулей
             end <идентификатор>
```

Модули становятся активными, если они загружены в память другими программами. Модуль может временно передавать управление другому модулю для осуществления вычисления с некоторым запросом или предполагаемым ожиданием результата. Программа в модуле выполняется на основе доступной в среде этого модуля информации. Если комплексная функция слишком большая, ее можно распределить между несколькими модулями, а высокоуровневая функция может вызвать встроенные в другие модули функции.

Одним из ограничений модулей является недостаток информации, передаваемой между двумя модулями с помощью механизмов передачи. Список экспорта также становится довольно большим. Несмотря на то, что модули вычисляются отдельно, программисты должны соблюдать соответствие типов между переменными и операторами. Типы переменных хранятся в *файле символов* - метафайле, используемом для обеспечения согласованной работы между модулями. Каждый раз, когда вычисляется пользовательский модуль - модуль, использующий включенный модуль, в файле символов проверяется совместимость типов.

Большинство языков, например, ADA, C++, CLU, Euclid, Fortran 90 onward, Java, семейство языков Modula и Ruby поддерживают понятие явного модуля. В языках ADA и Java модули называются *программными пакетами*. В то время как вышеупомянутые языки поддерживают вложенную структуру модулей, другие языки, например, Mesa, поддерживают плоскую структуру модулей. В языке Mesa модули - это простые функциональные единицы, которые разбивают большую программу на несколько упрощенных единиц по принципу функциональности.

4.2.7 Объекты и классы

Другой важной категорией инкапсуляции является понятие об объектах. Как уже было указано, основной целью инкапсуляции является регулирование видимости объявленных элементов в границах модуля, то есть один модуль становится независимым от другого с точки зрения объявления, вычисления и низкоуровневой информации, содержащейся в нем. Объекты делают именно это. Тем не менее, в отличие от явного определения модулей, являющихся пассивными единицами, объект - является активной единицей, которую можно создать во время выполнения программы, вызвать и которая имеет вычислительное состояние во время выполнения. Объекты создаются с помощью конструктора, который локализует область памяти в динамической области памяти.

Каждый объект инкапсулирует элементы данных и методы - подпрограммы и функции, которые применяются к этим элементам данных. Инкапсуляция обеспечивает сокрытие данных. В большинстве языков программирования поддерживается понятие об объектах, например, в таких как ADA, Modula-2 и других объектно-ориентированных языках. Доступ к элементам данных внутри объекта может быть получен с помощью абстрактного синтаксиса вида *<имя-объекта>.<имя-элемента-данных>*, а доступ к методам может быть получен с помощью синтаксического правила вида *<имя-объекта>.<название-метода>*.

Одна из интересных категорий языков программирования - объектно-ориентированное программирование. Оно основано на *классах объектов.* Как правило, за *определением класса* следует иерархическая структура, это означает, что корневой узел дерева находится в определении класса верхнего уровня, на уровне 1 располагаются определения класса, являющегося подклассами корневого класса, это свойство определения подкласса относится и к уровню 2, уровню 3 и т.д. Объект является примером класса или подкласса, который может быть создан или вызван во время выполнения. Кроме того, иерархия классов предусматривает наследование методов - подпрограмм и функций, объявленных в родительском классе - и нет необходимости переопределять или напрямую импортировать их. Тем не менее, подкласс может добавить свое собственное определение новых методов или *переопределить* определение методов из родительского класса или класса-предка. Определения класса отличаются от явных определений модуля, так как у классов есть объекты как экземпляры классов, и они поддерживают наследование; явное определение модулей не поддерживает ни

экземпляров, ни наследования. Тем не менее, у модулей могут быть методы, как например в мультипарадигматическом языке Ruby. Абстрактный синтаксис класса представлен ниже

```
<class> ::= class
          <идентификатор>
          [subclass-of]
          <идентификатор>
          [private {<объявление-
          данных>}*]
          [protected {<объявление-
          данных>}*] [public
          {<объявление-данных>}*]
          [private {<объявление-
          метода>}*] [protected
          {<объявление-метода>}*]
          public {<объявление-
          метода>}+
```

Общедоступный метод (public method) видим другим объектам, частный метод (private method) невидим для объектов из других классов или подклассов; защищенные методы (protected methods) видимы только для подкласса, не считая явно запечатанных внутри класса. В зависимости от языка, как общедоступное объявление, так и частное объявление могут быть явными.

Объект подкласса создается во время выполнения с помощью *конструктора*, локализующего область памяти для объекта, связывает идентификатор и область памяти и инициализирует элементы данных в объекте к исходным значениям. У каждого объекта есть свое собственное состояние, и объекты взаимодействуют друг с другом с помощью отправки сообщений вызываемым общедоступным методам.

Объектно-ориентированные языки поддерживают абстракции данных в виде классов. Некоторыми из поддерживаемых классов являются хэш-таблицы - пары (ключ, значение), в которых ключ находится с помощью хэш-функций; массивы - индексируемый набор элементов данных, матрицы - объединение наборов элементов данных, структуры - именованные кортежи, векторы - расширяемые наборы элементов данных и таблицы - набор пар (ключ, значение).

Языки программирования, такие как C++, Eiffel, CLOS, Modula-3, Ruby, Scala и Java поддерживают классы объектов. Объектно-ориентированное

программирование основано на понятии об иерархических классах объектов и подробно рассматривается в Главе 11.

4.3 ОБМЕН ИНФОРМАЦИЕЙ

Каждый программный блок выполняет программы, включая вызов другой программы. Программный блок вызывает другой программный блок с помощью обращения или сам может быть вызван другим программным блоком. Каждый программный блок может быть программой, функцией или методом внутри объекта. Многие языки поддерживают вложенность программных блоков, программный блок может быть составлен в иерархическом порядке, это означает, что программный блок может быть встроен в другой программный блок. Программный блок также может передаваться как аргумент другому программному блоку. Всегда, когда программный блок вызывает другой программный блок, часть среды сохраняется, по большей части информация передается вызываемому программному блоку вызывающим программным блоком, а результат возвращается обратно вызывающему программному блоку от вызываемого программного блока. Обмен информацией между программными блоками необходим для успешного решения задач.

4.3.1 Механизмы обмена информацией

Существует несколько способов обмена информацией. Для обмена информацией подгруппа среды и блока памяти, доступная вызывающему программному блоку, должна стать видимой вызываемому программному блоку. Видимость можно достигнуть, если (1) сделать подгруппу видимой всем программным блокам; (2) сделать подгруппу видимой для подгруппы программных блоков, исходя из уровня вложенности программных блоков; и (3) сделать подгруппу видимой с помощью видимости точка-точка между программными блоками путем обмена одним из трех основных атрибутов переменной: *имени*, *ячейки памяти* или *значения*. Видимость может быть достигнута несколькими способами: (1) специальным созданием копии значения из вызывающего программного блока; (2) передачей ссылки ячейкам памяти; (3) копированием ссылки для первой ячейки памяти элемента данных; или (4) передачей имени переменных.

FORTRAN использует объявление “COMMON” между подпрограммами для общего пользования областью памяти. Преимуществом общего пользования областью памяти является то,

что создается только одна копия области памяти, и одновременно и вызывающая подпрограмма, и вызываемая подпрограмма могут быть записаны в совместную область памяти и извлечены из нее. Это уменьшает излишние траты на копирование и распределение памяти. Общим блокам присваивается имя, и элементы данных в двух подпрограммах, обладая одинаковыми блоками, сопоставляются для соответствия ячейка за ячейкой. С точки зрения программирования у объявления общего блока есть два основных недостатка:

1. 1. Общий блок должен быть объявлен в каждой подпрограмме, в которой он используется.

2. Расположение нескольких переменных, объявленных в общем блоке, является ненадежным.

Блочно-структурированные языки используют глобальные переменные, чтобы сделать информацию видимой каждому программному блоку, и нелокальные переменные для вложенных программных блоков. Нелокальные переменные доступны только программным блокам, встроенным внутрь. Информация внешним программным блокам может быть передана с помощью переменной, локальной для внешнего блока, которая является нелокальной переменной для программного блока, вложенного во внешний программный блок. Кроме того, блочно-структурированные языки используют имя переменной, или адрес ячейки памяти или значение переменных для передачи информация между вызывающим программным блоком и вызываемым программным блоком. При передаче параметров аргументы в вызывающей программе называются *фактическими параметрами*, а соответствующие аргументы в вызываемом программном блоке называются *формальными* параметрами. При передаче параметров соответствие параметров может быть достигнуто четырьмя способами:

1. Фактические и формальные параметры совмещаются слева направо, это соответствие - позиция за позицией. Это общий режим соответствия параметров в большинстве языков программирования.

2. Имена фактического и формального параметра сопоставляются с помощью ассоциаций имен, например, в языке программирования ADA. Если используются ассоциации имен, тогда нет необходимости в сопоставлении по позициям. Соответствие между фактическим и формальным параметром в вызове подпрограммы. В случае совмещения

имен - один и тот же фактический параметр связывается с двумя другими формальными параметрами - для связывания атрибутов фактического параметра с формальными параметрами используется упорядочивание позиций.

3. Если количество аргументов в вызове процедуры и количество формальных параметров не сопоставляются, тогда формальные параметры, оставшиеся после сопоставления формальных и фактических параметров, инициализируются по умолчанию.

4. Формальные параметры могут быть расширяемого типа, например, список, и неопределенное количество аргументов может передаваться вызываемой программе, как например в C# с помощью объявления “param”.

Объектно-ориентированные языки программирования могут обладать вложенными структурами классов. Таким образом, переменная может быть описана в классе и быть видимой во вложенных подклассах. Переменная, объявленная в классе, называется *переменной класса*, и она видима среди всех методов и элементов данных в этом классе. Переменная может быть *статической глобальной переменной*, что означает, что она является доступной во всех классах и, следовательно, для всех объектов. Переменная может быть характерной для метода. Информация может передаваться между объектами с помощью глобальных переменных, переменных класса, передачи параметров между методами и передачи сообщений между объектами. Если язык поддерживает модули, а также классы и объекты, как, например, Modula-3, тогда методы и переменные могут быть использованы между модулями посредством механизма импорта-экспорта.

У функциональных языков программирования есть мощный механизм обмена информацией. Они могут предавать целую функцию как параметр, к которой могут обратиться в вызываемой функции. Двойственная природа функции как данных и как функции - это особенность функциональных языков. Функциональное программирование также использует текстуальную замену формального параметра на имя фактического параметра перед вычислением выражения. Если вычисление выражения откладывается до востребования, то оно не может быть вычислено до передачи информации вызываемой функции. Скорее всего, оно должно полагаться на текстуальную замену. Это свойство передачи информации за счет использования текстуальной замены формального параметра

на имя фактического параметра называется *вызов по имени* и будет рассмотрено в следующем разделе.

4.4 ПЕРЕДАЧА ПАРАМЕТРОВ

Переменная в императивном языке определяется как *имя↦ ячейка памяти ↦ r-значение*. К переменной относятся три атрибута: *имя, ячейка памяти* и *значение*. Только при задании всех трех можно получить окончательное r-значение. Это делает передача параметров. Если имя передается как аргумент в фактическом параметре, тогда механизм передачи параметров называется *передача параметров по имени*. Если ячейка памяти передается как аргумент, тогда механизм передачи параметров называется *передача параметров по ссылке*, а если передается r-значение, тогда механизм передачи параметров называется *передача параметров по значению* или *передача параметров по копии.*

В большинстве случаев вызываемая подпрограмма передает результаты, полученные из локального вычисления, обратно вызывающей программе. Результаты могут быть переданы с помощью глобальных переменных,

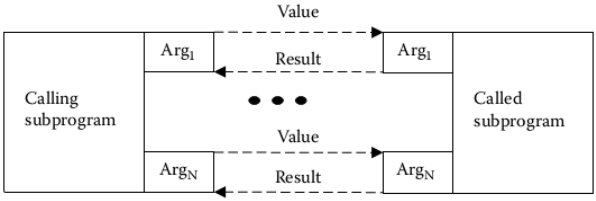


РИСУНОК 4.4 Передача параметров с использованием копирования r-значения аргументов.

нелокальных переменных, разделения общей области памяти и использования передачи параметров. Передача параметров по имени и передача параметров по ссылке неявно обрабатывают ячейки памяти, связанные с переменными в вызывающей подпрограмме. Есть три варианта передачи параметров по значению, что показано на Рисунке 4.4.

Вызывающая программа может предавать информацию от фактического параметра к формальному параметру. Тем не менее, результат не передается обратно от вызываемой подпрограммы к вызывающей подпрограмме. Этот тип передачи параметров называется *передача параметров по значению.*

Вызывающая программа может предавать информацию от фактического параметра к формальному параметру. После того как выполнена вызываемая подпрограмма, результат возвращается в вызывающую программу. Этот механизм называется *передача параметров по значению результата.* В *передаче параметров по значению результата* передача информации осуществляется двумя вариантами. Тем не менее, результат передается обратно только после того, как завершится вызываемая программа. Обмен информацией происходит в определенном порядке. Как правило, слева направо, исходя из соответствия позиций.

Последний механизм передачи параметров, в котором r-значение передается явно, называется *передача параметров по результату*. При передаче параметров по результату информация не передается от фактического параметра к формальному, когда происходит вызов подпрограммы; формальные параметры инициализируются по умолчанию. Тем не менее, после завершения вызываемой подпрограммы результат возвращается в вызывающую программу посредством соответствия формального параметра и фактического параметра.

Как правило, количество аргументов, передаваемых как параметры четко определено и зафиксировано. Однако, некоторые языки допускают переменное количество фактических параметров, передаваемых вызывающей подпрограмме. Переменное количество фактических параметров представляет собой расширяемую абстракцию данных в формальном параметре, которая может быть обработана итерационным циклом, получающим размер расширяемой абстракции данных и обрабатывающим каждый элемент расширяемой абстракции данных. Такую особенность поддерживают языки C# и Java. Например, C# отображает три аргумента, такие как “Mike,” “Karen,” и “Ambika”, в вызывающей программе с помощью объявления формального параметра “param string[] names.” Зарезервированное слово “param” дает вызывающей подпрограмме команду обращаться с остальными аргументами взывающей подпрограммы как с расширяющейся абстракцией данных. Здесь “names” - вектор, состоящий из трех элементов данных: “Mike,” “Karen,” и “Ambika.” Преимущество передачи расширяемых параметров как вектора заключается в том, что его можно представить и обработать как вектор.

Механизмом передачи параметров в парадигме логического программирования является *унификация.* Различие между унификацией и другими механизмами передачи параметров состоит в том, что

унификация допускает двухстороннюю передачу информации и присваиваний, но не вычисляет выражения. Унификация подробно изучается в Главе 10.

4.4.1 Передача параметров по значению и ее разновидности

При *передаче параметров по значению* (*передаче параметров по копии*) или *передаче параметров по режиму* в выражении сначала вычисляется фактический параметр, а затем результирующее значение передается вызываемой программе и связывается с соответствующим формальным параметром. Формальный параметр имеет такой же статус, как и локальная переменная, а связывание формальных параметров со значениями выражений в фактическом параметре подобно оператору присваивания. При обращении к вызываемой подпрограмме, в среде вызываемой подпрограммы создаются новые ячейки памяти, и ячейкам памяти присваиваются значения вычисленных выражений из соответствующих фактических параметров. Однако, копирование происходит строго в одном направлении: из ячейки памяти, соответствующей фактическому параметру, в ячейку памяти формального параметра. После копирования значений между формальным и фактическим параметрами не происходит никаких взаимодействий. Вызываемая подпрограмма проводит вычисление и завершается без передачи какой-либо информации.

Передача параметров по значению используется для вызывающих функций, которые не возвращают результат с помощью соответствия параметров. Однако, функции явно возвращают значение с помощью “return (выражение)”, как показано в Примере 4.8.

Пример 4.8

Давайте рассмотрим следующий код. Программа считывает значения переменных *x* и *y*, передает значения переменных *x* и *y* функции “square_ sum” с помощью передачи параметров по значению, а функция возвращает вычисленное значение главной вызывающей программе. Общий синтаксис для объяснения понятия.

```
program main
{ integer x, y, z;
  read(x, y);
  z = square_sum(x, y)
  print("square sum of the numbers: ~d and ~d is
```

```
  ~d", x, y, z);
}

unction integer square_sum(a, b)
{ return(a*a + b*b);}
```

Формальный параметр “a” соответствует фактическому параметру “x”, а формальный параметр “b” соответствует фактическому параметру “y”. Обратите внимание, что формальные параметры имеют такой же тип, как и фактические параметры. Как “a”, так и “b” - это копии переменных *“x”* и *“y”*, вызывающие распределение дополнительной памяти в среде функции *square_sum*. Основное преимущество передачи параметров по значению заключается в том, что поскольку результат не передается обратно соответствующему фактическому параметру, вызываемая подпрограмма не обновляет деструктивно соответствующие ячейки памяти, что поможет избежать побочных эффектов - нежелательного поведения программы из-за неожиданных деструктивных обновлений в среде вызываемой процедуры. Передача параметров по значению также используется в случаях, когда вызываемой подпрограмме для вычисления необходимы только исходные значения параметров, как показано в Примере 4.9.

Пример 4.9

Следующая программа состоит из двух программных блоков: *main* и *my_print*. Программа *main* считывает два массива “a” и “b”, размер каждого 100, и вызывает подпрограмму *my_ print* , чтобы сложить их поэлементно и напечатать результат.

```
program main
{     integer x[100]; y[100];
      for (i = 0, i = < 99; i++) read(x[i], y[i]);
      call my_print(x, y);
}

subprogram my_print(integer a[100], b[100])
{       nteger c[100];
  for (i = 0; i = < 99; i++) {
```

```
        c[i] = a[i] + b[i];
        for (i = 0; i = < 99, i++)
        print("c[~d]= " ~d~n", i, c[i]);
        }
}
```

У передачи параметров по значению есть два недостатка: (1) копирование требует дополнительное количество ячеек памяти, такое же, которое требуется для хранения фактических параметров. Если фактический параметр большой, например, матрица размером 10000 × 10000, как часто бывает в крупномасштабных научных вычислениях, тогда потребуется дополнительные 100 миллионов × размер (отдельного элемента данных) ячеек памяти для выполнения вызываемой подпрограммы, и (2) затраты на копирование крупных структур данных из памяти вызывающей подпрограммы в память вызываемой подпрограммы будут достаточно большими.

4.4.1.1 *Вызов по значению для совместного использования сложных объектов*

Сложные объекты расширяемых структур данных и динамические объекты в объектных языках программирования хранятся в динамической области - общем глобальном пространстве. Эти объекты извлекаются из среды программного блока с помощью ссылки. Ссылка указывает на основной адрес объекта данных, сохраненного в динамической области. Когда вызывающей подпрограмме необходимо поделится объектом с вызываемой подпрограммой, то ссылка на объект, которым необходимо поделится, копируется в среду вызываемой программы с помощью передачи параметров по значению, как показано на Рисунке 4.5. Левая часть показывает последовательность действий до обращения к вызываемой подпрограмме. Так как скопированная информация представляет собой адрес, то формальный параметр переносит адрес объекта. Этот механизм используется в каждом языке, который поддерживает динамические объекты, хранящиеся в динамической области.

4.4.2 Передача параметров по ссылке и ее разновидности

Передача параметров по ссылке, также называется *передача параметров по доступу,* передает l-значения или ячейку памяти фактического параметра формальному параметру. Формальный параметр

- это ссылка на фактический параметр. В вызываемой программе ячейка памяти фактического параметра доступна расшифровке формального параметра, а ячейка памяти, соответствующая

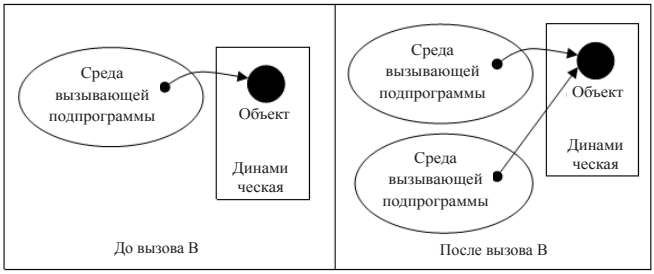


РИСУНОК 4.5 Использование передачи параметров по значению для копирования ссылок на объекты.

фактическому параметру прочитывается или обновляется. В случае набора элементов данных, например, массива или вектора, ячейка памяти первого элемента данных передается, а другие элементы данных становятся доступными благодаря методу смещения, который рассчитывает адрес *i-ого* элемента данных, прибавляя размер смещения *i* * (одного элемента данных) и добавляя его в ячейку памяти, хранящуюся в формальном параметре.

Как видно из объяснения, что изменение потерявшего связь значения формального параметра совпадает с доступом к ячейке памяти фактического параметра и ее обновлением. Нет необходимости передавать результаты явно вызывающей подпрограмме из вызываемой подпрограммы.

У *передачи параметров по значению* есть много преимуществ:

1. 1. Формальный параметр - это только указатель первой ячейки памяти сложной структуры данных, и он не зависит от размера структуры данных.

2. 2. Нет необходимости явно копировать значения из фактического параметра, излишне тратя ресурсы ячеек памяти и времени выполнения.

3. 3. Нет необходимости явно передавать результаты вычисления вызывающей программе, так как ячейки памяти фактических

параметров постоянно обновляются, когда изменяются соответствующие формальные параметры.

Пример 4.10

Следующая программа состоит из двух программных блоков: *main* и *count.* Программный блок *main* использует генератор случайных чисел в библиотеке программ для создания целых чисел от 1 до 200, а вызываемая подпрограмма *count* считает количество элементов части массива *d*, в котором значения чисел больше 100. Предположим, что целое число занимает 4 байта и указатель занимает 4 байта.

В программе *main* содержится массив целых чисел *d* и три дополнительных целочисленных переменных: *i, j,* и *final_count*. Программа *main* вызывает подпрограмму *count*, используя три фактических параметра: массив *d* посредством передачи данных по ссылке; переменную *j* посредством передачи параметров по значению; и переменную *final_count* посредством передачи данных по ссылке. Переменный аргумент *j* хранит значение последнего индекса массива *d*. Символ "&" обозначает, что адрес был передан

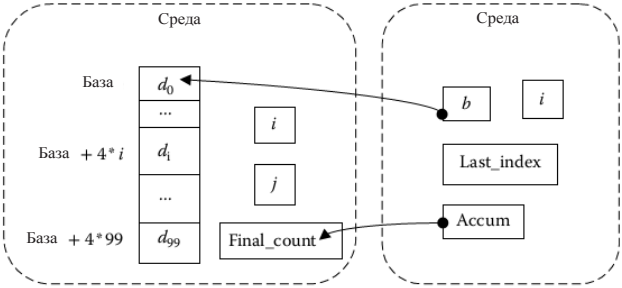


РИСУНОК 4.6 Принципиальная схема передачи параметров по ссылке.

вызываемой подпрограмме, а символ "*" означает, что следующий символ является ссылкой на фактический параметр, и нет необходимости разыменовывать доступ к r-значению.

В программном блоке *count* есть три формальных параметра: переменная *b* - ссылка на элемент данных d[0], переменная *last_index* - копия фактического параметра *j* и ссылочная переменная *accum* - ссылка на фактический параметр *final_count*. Ссылочные переменные *b* и *accum*

занимают 4 байта (32 бита), независимо от размера элементов данных, на которые они указывают. Адрес элемента *b[i]* в программе рассчитывается как *address(d[0])* + 4 * *i*. Обратите внимание, что фактический параметр *j* передается по механизму передачи по значению, а формальный параметр *last_index* получает копию значения *j*. На Рисунке 4.6 изображены ссылки в передаче параметров по ссылке.

```
program main
{     integer d[100], i, j, final_count;
      for (i = 0; i = < 99; i++) d[i] = random_
      number(1, 200); j = 50;
      call count (&d,j, &final_count);
}
subprogram count (integer *b, last_index, *accum)
{     integer index;
      *accum = 0;
      while (index =< last_index)
           {      if (*b[index] > 100) *accum =
                    *accum + 1; % end_if
            index = index + 1;
               }% end_while
}
```

В передаче параметров по ссылке есть несколько недостатков:

1. Вызываемая подпрограмма обновляет блок памяти вызывающей программы. После завершения вызываемой программы, блок памяти вызывающей программы изменяется относительно своего состояния до выполнения вызываемой программы. Если обновление не требовалось вызывающей программе, то она будет выполняться ошибочно, так как она будет считывать значения для вычисления из искаженного блока памяти.

2. Каждый раз, когда используется фактический параметр, имеют место два обращения к памяти: одно обращение к ячейке памяти фактического параметра, и дальнейшее использование ячейки памяти фактического параметра для доступа к r-значению.

3. В распределенном вычислении, если вызывающая и вызываемая подпрограммы находятся на разных процессорах (или компьютерах), то у них будут различные адресные пространства. Доступ к разным адресным

пространствам требует использования протоколов передачи данных, а также упаковку и распаковку данных, что значительно повышает затраты, и, кроме того, каналы передачи данных характеризуются низкой надежностью.

Чтобы устранить проблему, связанную с самопроизвольным обновлением блока памяти вызывающей программы вследствие неправильного использования ссылочных параметров, некоторые языки, например, C++ и Modula-3, поддерживают опцию *неизменяемой* (доступной только для чтения) *передачи параметров по ссылке.* В *неизменяемой* п*ередаче параметров по ссылке* значение фактического параметра может считываться только при помощи указателя, хранящегося в ячейки памяти формального параметра; ячейку памяти, соответствующую фактическому параметру, нельзя обновить деструктивно.

В другом варианте сочетается передача первого параметра по механизму передачи по ссылке с последующей передачей параметров по значению. В результате все последовательности вызываемых подпрограмм будут иметь доступ к первоначальному фактическому параметру: первая ссылка, копирующаяся последовательно с помощью передачи параметров по значению, создает ссылку в каждой последующей вызываемой программе. Например, когда программный блок “A” использует передачу параметров по ссылке для передачи фактического параметра “x” вызываемой подпрограмме “B”. Формальный параметр “y” в программном блоке “B” будет ссылкой на “x” в вызываемой подпрограмме «B». Если программный блок “B” вызывает другой программный блок “C” и передает “y” посредством передачи параметров по значению формальному параметру “z” в программном блоке “C”, тогда формальный параметр “z” также будет ссылкой на “x.” Программные блоки “B” и “C” имеют доступ к фактическому параметру “x” через свои ссылки. Такая схема применяется во многих языках, особенно объектных языках, таких как C#, Java, и C++, и называется *вызов по обмену*.

4.4.3 Вызов по результату

Вызов по результату, также называемый передачей параметров с помощью *копирования режима*, является противоположностью вызова по значению. В вызове по результату фактический параметр не копируется в ячейку памяти формального параметра. Вместо этого, формальный параметр инициализируется в значение по умолчанию согласно типу

объекта. В конце вызываемой процедуры значение формального параметра копируется обратно в фактический параметр. Как и вызов по значению, вызов по результату воспринимает формальные параметры как локальные переменные. Во время выполнения вызываемой программы между формальным параметром и фактическим параметром нет никаких взаимодействий.

4.4.4 Вызов по значению результата

Вызов по значению результата, также называемый *передачей параметров по режиму вход-выход,* передает r-значение двумя способами: после определения выражения в фактическом параметре в формальный параметр, он передает обратно результат после завершения вызываемой подпрограммы. Обратная передача результата осуществляется в те же ячейки памяти в установленном порядке. Во время выполнения вызываемой подпрограммы, между вызывающей и вызываемой подпрограммами не происходит никаких взаимодействий.

Как и вызов по значению, вызов по значению результата воспринимает формальные параметры как локальные переменные и создает ячейки памяти в локальной среде вызываемой подпрограммы. Затраты на распределение памяти в вызове по значению результата такие же, как и в вызове по значению. Однако, затраты на копирование в вызове по значению результата в два раза больше, чем в вызове по значению, так как вызов по значению результата также передает обратно результат фактических параметров.

Пример 4.11

Следующая программа создает локальные переменные x[100], y[100] и z[100, 100] и копирует значения a[100] в соответствующие ячейки памяти x[100], b[100] в y[100], a c[100, 100] в z[100, 100]. Окончательный результат передается обратно после завершения подпрограммы “multiply” («умножить»), так, что из x[100] результат копируется в a[100]; y[100] копируется в b[100]; a z[100, 100] копируется в c[100, 100].

```
Program main
{   integer a[100], b[100], c[100, 100], i;
     for (i = 0; i =< 99; i++) read(a[i]);
     for (i = 0; i =< 99; i++) read(b[i]);
     call multiply (value-result a[100], b[100],
```

```
    c[100, 100]);

}
subprogram multiply(integer x[100], y[100], z[100,
100])
{  integer i, j;
    for (i = 0; j = < 99; i++)
   for (j = 0; j = < 99; j++)
            z[i, j] = x[i] * y[j];
}
```

Несмотря на то, что вызов по значению результата передает обратно в вызывающую программу результат осуществления вычисления, между вызовом по ссылке и вызовом по значению результата существует значительное различие:

1. 1. Вызов по ссылке сохраняет изменения фактических параметров, так как вычисление происходит в вызываемой подпрограмме и сохраняет последнее значение в связи с продолжительным деструктивным обновлением. Хотя окончательное значение в вызове по значению результата зависит от порядка аргументов в вызове процедуры, фактические параметры обновляются только после успешного завершения вызываемой подпрограммы. Окончательное значение фактического параметра в вызове по значению результата не обязательно должно быть таким же, как и в вызове по ссылке, что показано в Примере 4.12.

`main ( );`
`integer i;`
`{i = 1; sub(& i, & i);}`
 	
`void sub(integer *j, *k);`
`{*k = 4; *j = 2}`
	`main ( );`
	`integer i;`
	`{i = 1; sub(value-result i, value-result i);}`
	
	`void sub(integer j, k);`
	`{k = 4; j = 2}`

РИСУНОК 4.7 Сравнение *вызова по ссылке* и *вызова по значению результата*.

2. Для вызова по ссылке необходима только одна ячейка памяти, чтобы хранить ссылку доступа к большой структуре данных, а вызов по значению результата создает копию фактического параметра. Таким

образом, затраты на выделение памяти для больших структур данных пренебрежимо малы в вызове по ссылке по сравнению с вызовом по результату значения.

3. В вызове по ссылке есть дополнительные затраты на доступ к фактическим значениям, что связано с получением доступа к дополнительной памяти. Если вызывающая программа выполняет операции, связанные с большим объемом вычислений, над большими массивами, векторами или большим набором элементов данных, то тогда затраты на доступ к памяти могут быть намного больше чем затраты на копирование в вызове по значению результата.

4. В распределенном вычислении, где вызывающая и вызываемая программы принадлежат двум разным процессорам или компьютерам, затраты на доступ к памяти в вызове по ссылке будут значительными, и работать с локальной копией фактического параметра более предпочтительно .

Пример 4.12

Давайте установим различие между вызовом по значению результата и значением по ссылке с помощью программы, представленной на Рисунке 4.7. В левой колонке представлен вариант программы с вызовом по ссылке, а в правой колонке вариант той же самой программы с ссылкой по значению результата. Знак “&” обозначает адрес переменной, а знак “*” используется для удаления ссылки указателя доступа к значению фактического параметра. В варианте программы с вызовом по ссылке ячейка памяти фактического параметра обновляется в порядке выполнения операторов. Следовательно, окончательное значение переменной *i* будет равно 2. В варианте программы с вызовом по значению результата результат возвращается обратно в конце вызываемой процедуры, в порядке слева направо. Следовательно, окончательное значение переменной *i* будет окончательным значением формального параметра “*k*”, а значит равно 4.

4.4.5 Вызов по имени

Вызов по имени - это третья основная категория передачи параметров, в которой формальный параметр замещается дословно на целое выражение фактического параметра без каких-либо вычислений до замещения, и это замещенное тело вызываемой подпрограммы выполняется в среде вызывающей процедуры по запросу с помощью метода, называемого

переключение. Переключение - это безпараметровая процедура с невычисленным выражением, которое вычисляется каждый раз, когда среда вызывающей процедуры имеет доступ к фактическому параметру. Переключение возвращает адрес фактического параметра каждый раз, когда выражение вычисляется в среде вызывающей процедуры. Если тело вызываемой подпрограммы после замещения содержит локальную переменную, с которой у объявленной в вызывающей программе переменной возникает конфликт наименований, тогда имя локальной переменной в теле вызываемой подпрограммы изменяется во избежание конфликта наименований. Различие между вызовом по имени и вызовом по значению заключается в том, что выражение в фактическом параметре при вызове по имени не вычисляется непосредственно перед замещением. Наоборот, оно откладывается до тех пор, пока не будет выполнено замещение, и вычисляется каждый раз заново, когда доступен фактический параметр.

Пример 4.13

На Рисунке 4.8 представлена программа с передачей параметров с *вызовом по имени.* Слева представлена фактическая программа, а справа показана программа во время выполнения, после обращения к вызываемой подпрограмме “sub”. Формальные параметры *a, b*, и *w* замещаются на выражение $x + y$, $x + z$ и w без каких-либо вычислений.

Переменная *z* в правой части выражения оператора присваивания соответствует переменной *z*, объявленной в среде программы *main*, и вступает в конфликт наименований с локальной переменной *z* в подпрограмме *sub*. Таким образом, переменная *z* переименовывается в *z1*, и замещенное тело в вызываемой подпрограмме логически подставляется на место вызова. Результат представлен в правой колонке на Рисунке 4.8. В общем случае, после подстановки выражения тело подпрограммы *sub* выполняет функцию блока вызывающей программы. При выполнении программы *z1* связывается с вычислением (3 + 4) * (3 + 4) + (3 + 5) *(3 + 5), а окончательное значение получает переменная *w*. Вызов по имени - это мощный механизм. Тем не менее, у него есть два недостатка:

1. Задерживание вычисления из-за использования переключения ведет к дополнительным затратам на выполнение программы и вычисление.

2. Разрешение конфликта имен также требует дополнительных затрат.

Задерживание вычисления выражения в переключении означает, что отображение идентификатора в ячейке памяти может измениться во время выполнения, а это может вызвать серьезные проблемы

Программа	Поведение программы после вызова подпрограммы sub

program main
{ **integer** x, y, z;
real r;
x = 3; y = 4; z = 5;
call sub(x + y, x + z, w);
}
 	
subprogram sub (**name** a, b, w)
{ **integer** z;
z = a * a + b * b;
w = square_root(z);
}
	program main
	{ **integer** x, y, z;
	real w;
	x = 3; y = 4; z = 5;
	
	{ **integer** z1; % rename the variable
	z1 = (x + y) * (x + y) + (x + z)
	* (x + z);
	w = square_root(z1);
	}
	}

РИСУНОК 4.8 Пример механизма вызова по имени.

```
program main
{   integer  i, j, k; integer a[5];
    k = 3; j = 2;
    for (I = 0; I =< 4; i++) a[i] =0;
    swap(name j, k);
    swap(name k, a[k];
}

subprogram swap(name m, n);
{   integer  temp;
    temp = m; m = n; n = temp;
}
```

```
program main
{   integer  i, j, k; integer a[5];
    k = 3; j = 2;
    for (I = 0; I =< 4; i++) a[i] =0;
    {   integer temp;
        temp = j; j = k; k = temp;
    }
    {  integer temp;
       temp = k; k = a[k]; a[k] = temp;
    }
 }
```

РИСУНОК **4.9** Проблема путаницы в ячейках памяти в *вызове по имени.*

в поведении программы во время выполнения, которое может отличаться от понимания программы по структуре. Очень трудно делать выводы о программах с вызовом по имени, особенно тех, в которых содержатся индексируемые элементы a[i], так как значение “i” может быть вычислено как выражение, изменяя при этом ячейку памяти, связанную с идентификатором “a[i]”, в то время как программы могут быть написаны для a[i], связанной с фиксированной ячейкой памяти, как показано в Примере 4.14 и на Рисунке 4.9.

Пример 4.14

На Рисунке 4.9 показана программа, заменяющая значения переменных с помощью нескольких обращений к подпрограмме *swap*. В первую очередь она заменяет значения *j* и *k*; затем заменяются значения *k* и *a*[*k*]. Процедура замены является стандартной. В правой колонке на рисунке показано поведение программы при вызове по имени.

При первом обращении к *swap*(*j, k*) все работает корректно, присваивается значение *j* = 3 и *k* = 2. Однако, второй вызов присваивает значение *k* = *a*[*2*], равное 0. Как только *k* становится равным 0, то вместо обновления *a*[*2*], программа обновляет *a*[*0*] до старого значения *k*, равного 2.

Вызов по имени ограниченно применяется в императивных языках, из-за того что перепутываются ячейки памяти. *Вызов по имени* первоначально был представлен в языке ALGOL-60. Позже его перестали использовать в других императивных языках программирования. Тем не менее, вариант *вызова по имени*, который называется *вызов по необходимости,* используется в функциональных языках программирования, он задерживает вычисление выражения до востребования. Язык Haskell - это пример языка программирования, где используется вызов по необходимости.

4.4.6 Вызов по необходимости

Вызов по необходимости - это вариант *вызова по имени.* В отличие от вызова по имени, где адрес фактического параметра вычисляется при каждом вычислении выражения, вызов по необходимости кэширует значение, полученное во время первого вычисления, и извлекает кэшированное значение при каждом вычислении выражения. Первое вычисление замедляется. При этом последующие вычисления не задерживаются.

В большинстве случаев, когда индекс не пересчитывается, как в примере на Рисунке 4.9, адрес не изменятся, и вызов по необходимости в теории ведет себя как эффективный *вызов по имени*, в котором не надо каждый раз вычислять значение. Результат кэширования рассчитывается один раз, а при последующих случаях копируется. *Вызов по необходимости* может быть рассмотрен как *вызов по имени,* за которым следует повторяющийся *вызов по значению*, где значение копируется из кэша.

Пример 4.15

Давайте сравним эффективность использования вызова по имени и вызова по необходимости на простом примере. На Рисунке 4.8 представлены два выражения $x + y$ и $x + z$, вычисляемые два раза в правой части оператора присваивания "z1 = ($x + y$) * ($x + y$) + ($x + z$) * ($x + z$);". Вызов по имени будет вычислять выражения оба раза, а *вызов по необходимости* будет вычислять выражение $x + y$ один раз и сохранит результат вычисления в кэш. В следующий раз, когда он столкнется с выражением $x + y$, он извлечет значение из кэша. Аналогично, вызов по имени будет оба раза вычислять выражение $x + z$, а вызов по необходимости будет вычислять первое появление выражения $x + z$, сохранит результат вычисления в кэш и при последующем появлении выражения извлечет значение из кэша.

Вызов по необходимости используется в функциональных языках программирования, например, в Haskell, где вычисление выражения откладывается до тех пор, пока оно не понадобится. Вызов по необходимости увеличивает скорость вычисления за счет сохранения значения подвыражения и использования результирующего значения во избежание вычисления такого же подвыражения, возникающего в другом месте выражения.

4.4.7 Передача подпрограмм в виде параметров

Языки, поддерживающие парадигму функционального программирования, такие как Lisp, Scheme, Haskell, Ruby и Scala, и императивные языки программирования, такие как ALGOL и Pascal, поддерживают передачу функций/подпрограмм в виде параметров. Передача функций в виде параметров включает в себя передачу ссылки на первую инструкцию функции и ссылку на соответствующую среду для проверки количества и типов аргументов во время выполнения. Это представляется достаточно сложным для языков со строгим статическим контролем типов данных - языков, где тип элементов данных объявляется во время компиляции - потому что разные подпрограммы могут быть вызваны разными обращениями во время выполнения, а анализ кода во время компиляции должен проследить за тем, чтобы среда, включающая типы аргументов, полученных вызываемой функцией, совпадала со средой и типами аргументов функции, переданной в виде параметра. В функциональных языках программирования с динамическим контролем типов данных такой проблемы не возникает, так как проверка типов

данных осуществляется во время выполнения.

4.4.8 Передача параметров для распределенных вычислений

Распределенное вычисление требует вызова удаленных процедур с разных процессоров. Это означает, что разные подпрограммы выполняются в разных адресных пространствах. Распределенное вычисление использует три типа передачи параметров: (1) *вызов по перемещению*, также называемый *вызовом по копии*; (2) *вызов по ссылке*; и (3) *вызов по посещению*, также называемый *вызов по копированию/восстановлению.* Эти механизмы передачи параметров распределяются адресно-пространственными двойниками вызова по значению, вызова по ссылке и вызова по результату в однопроцессорной реализации языков программирования. Вызов удаленной процедуры включает в себя вычисление выражения, перенесение копии объекта в удаленный узел и его связывание с формальным параметром, вычисление удаленной процедуры и возвращение результата или объекта обратно вызывающей процедуре.

Вызов по посещению временно создает копию объекта в удаленном процессоре и копирует объект обратно после успешного завершения вызываемой процедуры. *Вызов по перемещению* создает копию объекта для вычисления вызываемой процедуры на удаленном процессоре. Однако, объект не копируется обратно. Вызов по посещению аналогичен вызову по значению результата, а вызов по перемещению аналогичен вызову по значению. Вызов по ссылке просто копирует ссылку на сложный объект.

Передача адреса объекта из процессора вызывающей процедуры на удаленный процессор вызываемой процедуры связана со значительными затратами в распределенном вычислении, заключающихся в затратах на пропускную способность и прохождение через системные подпрограммы к рассредоточенным адресным пространствам. Несмотря на эти издержки, у вызова по ссылке есть преимущество, заключающееся в том, что ссылку на объект можно легко передавать между распределенными процессорами без дополнительных затрат на передачу данных.

Язык распределенного программирования Emerald использует три схемы передачи параметров, а именно, вызов по ссылке, вызов по посещению и вызов по перемещению. Передача параметров для распределенного вычисления повторно рассматривается в Главе 8, посвященной параллельному программированию.

4.5 ПОБОЧНЫЕ ЭФФЕКТЫ

У процедуры есть доступ к блокам памяти, созданным не локальными переменными, из-за наличия глобальных переменных, нелокальных переменных, ссылок на среду вызывающих процедур и объектов долговременных данных. *Побочный эффект* определяется как эффект, который существует дольше вызываемой подпрограммы. Побочный эффект может быть вызван (1) обновлениями, происходящими в блоке памяти, не относящемуся к среде, созданной локальными переменными вычисляющейся в данный момент процедуры; (2) наблюдаемым взаимодействием с внешним миром, например, запись в файл или поток; или (3) возникновением исключения. Как только изменения, вызванные в соответствующем локальном переменном блоке, прекращаются после выполнения процедуры, изменения в других блоках памяти, постоянных объектах, динамических объектах и рекурсивных структурах данных, существующих дольше чем подпрограмма, в которой они были созданы, сохраняются, даже если вычисляемая в данный момент процедура завершилась.

Программирование, основанное на побочном эффекте, используется для передачи результата обратно вызывающей подпрограмме в императивных языках. Однако, если блок памяти бессистемно изменяется некоторым *вычислением в сверхоперативной памяти* в вызываемой подпрограмме, тогда другие программные блоки, не зная о том, что ячейки памяти повреждены, могут непреднамеренно использовать поврежденные значения. Это вызывает непредсказуемое поведение, приводящее к некорректному результату вычисления.

Одной из серьезных проблем побочного эффекта является *потеря коммутативности.* Коммутативность - это основное свойство многих операторов при вычислении выражения, таких как сложение и умножение. Дано выражение $e_1 + e_2$, оба подвыражения e_1 и e_2 определяются считыванием значения из одного и того же первоначального блока памяти, и, в идеальном случае, они не должны изменять блок памяти. Тем не менее, если при вычислении выражения e_1 или e_2 создается побочный эффект, тогда первоначальный блок памяти изменится после вычисления подвыражения e_1 или e_2, а следующее вычисляемое выражение будет считывать значения из нового блока памяти, выдавая отличающееся значение выражения, как показано в Примере 4.16.

Пример 4.16

Рассмотрим следующую программу, использующую вызов по ссылке для передачи параметров вызываемой функции *square_sum*. Функция *square_sum* возводит в квадрат две переменные *x* и *y*, складывает поучившиеся значения и передает полученный результат обратно в главную программу. Однако, она деструктивно обновляет фактические параметры *x* и *y* до значений 9 и 16. После возвращения, когда значение *x* прибавляется к результату выполнения функции *square_sum*, значение *B* будет равно 34, что отличается от того значения 28, которое будет получено при замене подвыражений на *x* + *square_sum*(*x*, *y*).

```
program main
{    integer
     A, B,
     x, y; x
     = 3; y
     = 4;
     A = square_sum(&x, &y) + x; % A становится
     равным 34 вместо 28
     B = x + y;% B становится равным 25 вместо 7
     print(A, B, x, y);

}

function integer square_sum(integer *x, *y);
{         *x = *x * *x; *y = *y * *y; %
               Присваивание изменяет переменные x и y
          return (*x + *y);
}
```

4.5.1 Совмещение имен и побочные эффекты

Совмещение имен определяется как наличие двух идентификаторов, отображающихся в одной и той же ячейке памяти или два указателя, направляющих в одну и ту же ячейку памяти. Если одна из переменных было присвоено значение, другая переменная автоматически обновляется. Это может оказать сильное воздействие, если программист не знает о псевдонимах, или вызываемая подпрограмма собрана отдельно от вызывающей подпрограммы. Давайте разберем это явление в Примере

4.17.

Пример 4.17

В следующей программе описывается сочетание совмещения имен и вызова по ссылке, для того чтобы проиллюстрировать непредсказуемое поведение программы, которое может быть вызвано совмещением имен. Независимо друг от друга оба программных блока - *main* и *swap* - выполняются корректно. Эти блоки могут быть вычислены раздельно и соединены позже, или swap (замена) может быть загружен из другого модуля без знания исходного кода.

Главная программа дважды вызывает подпрограмму *swap*: (1) при первом вызове она заменяет значение, хранящееся в двух разных ячейках памяти и (2) второй раз она заменяет значение одной и той же ячейки памяти. Подпрограмма замены написана нестандартно, для того чтобы избежать использования любой локальной переменной, значение которой необходимо заменить. Вместо этого она использует арифметические выражения. Новое значение x равно $x_{стар} + y_{стар}$. Новое значение y равно $x_{нов} - y_{стар} \rightarrow x_{стар} + y_{стар} - y_{стар} \rightarrow x_{стар}$, а новое значение x равно $x_{стар} + y_{стар} - y_{нов} \rightarrow x_{стар} + y_{стар} - x_{стар} \rightarrow y_{стар}$.

Очевидно, что подпрограмма заменяет значения y и z без ошибок. Однако, когда вызывается *swap(&x, &x)*, оба формальных параметра относятся к одной и той же ячейке памяти, и первый оператор присваивания $*x = *x + *y$ удваивает значение фактического параметра x. Второй оператор присваивания сам из себя вычитает значение, сохраняя 0 в ячейке памяти фактического параметра x.

```
program main
{       integer x, y, z;
        x = 3; y = 4; z = 5;
        swap(&y, &z); % замена значений y и z
        swap(&x, &x); % сохранение 0 в ячейке
        памяти
          x - непредсказуемое поведение
        print(x, y, z) % x становится равен 0; Y = 5 и z
        = 4
}
subprogram swap(integer *x, *y);
```

```
{       *x = *x + *y;
        *y = *x - *y;
        *x = *x - *y;
}
```

Когда несколько указателей направляют на одну и ту же структуру в динамической области памяти, тогда снятие одного указателя отмечает ячейку памяти для повторного использования или очистки памяти, как показано на Рисунке 4.10.

На Рисунке 4.10 два указателя P и Q направляют на две разные части списка указателей: P указывает на начало, а Q на несколько ячеек данных. После того как снимают указатель P, целый список указателей становится пригодным для повторного использования, программа чистки памяти будет повторно его использовать, а ячейки памяти будут присвоены другим динамическим структурам данных. Таким образом, ячейка памяти будет повреждена, а программа будет выполняться с ошибками.

Другая проблема, которая вызывает нежелательные изменения в блоке памяти - нарушение границ абстракции данных во время выполнения, что может быть вызвано как адресной арифметикой, так и значением

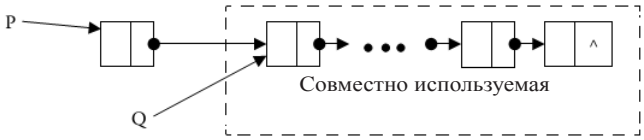


РИСУНОК 4.10 Указатели, совместно использующие структуру данных.

индекса, получающего доступ к массиву или вектору, которое выходит за верхнюю границу структуры данных. Например, если мы объявляем массив *a*[*100*] и во время выполнения значение индексной переменной *i* равно 130, тогда *a*[*i*] пересечет границу выделенной памяти для структуры данных *a*[*100*] и обратно вернет неверное значение, взятое из другой ячейки памяти.

4.5.2 Регулирование побочных эффектов

Существуют различные причины, взывающие появление побочных эффектов, которые приводят к некорректному выполнению программы.

Этими причинами могут быть: (1) сверхоперативное вычисление в переменных, область видимости которых не является локальной, (2) адресная арифметика, (3) независимое размещение указателей и их возможность независимо указывать на структуру данных, и (4) деструктивное обновление переменных или объектов данных, чьи области видимости не локальны. Существует несколько подходов решения проблем, связанных с побочными эффектами. Подходы должны обеспечивать ограничение на уровне языка, а также упорядоченный подход к программированию. Существуют следующие подходы:

1. Дисциплина *программиста:* Использование локальных переменных для сверхоперативных вычислений. Переменные, чьи области видимости выходят за пределы локальной среды, должны изменяться только для передачи информации. На сегодня это самый популярный подход управления нежелательными побочными эффектами, вызванными изменениями переменных, области видимости которых выходили за пределы локальной среды.

2. Отказ от адресной арифметики: Указатели не подвергаются каким-либо арифметическим действиям. Это относится к случаю нарушения типа, вызванного выходом указателя за пределы элемента данных во время выполнения.

3. Отказ от независимых указателей: Указатели могут быть объявлены, только при описании рекурсивной структуры данных и внутри программы для указания на объект. Это решает проблему несовместимых действий над разными типами данных.

4. Отказ от деструктивного обновления: Переменные программно можно связать со значением только один раз. Поскольку деструктивное обновление - это главная причина изменения блока памяти в вызывающей подпрограмме, то источник этого побочного действия необходимо устранить. Решение данной проблемы было испытано в ранних версиях декларативных языков. Свойство присваивания значения один раз: (1) ограничивает повторное использование памяти; (2) является причиной чрезмерного создания переменных; и (3) принудительно использует рекурсивный стиль программирования из-за отсутствия поддержки итерации, которая требует деструктивного обновления индексных переменных в каждом цикле. Ограничение использования изменяемых объектов вылилось в несколько интересных стилей рекурсивного программирования и использование итераторов в языках декларативного программирования. Итераторы впоследствии были заимствованы

большинством современных языков, так как они обеспечивают степень сокрытия информации об уровне реализации деталей.

4.5.3 Наглядный пример

В этом разделе описывается гипотетический пример программирования, в котором используется вызов по значению, вызов по ссылке, вызов по значению результата и происходит использование ошибочной ячейки, вызванное совмещением имен. В вызове по ссылке перед фактическим параметром ставится знак “&”, указывающий, что передается адрес, и перед формальным параметром ставится знак “*”, показывающий удаление ссылки для обеспечения доступа значению в фактическом параметре. Наличие символа “#” перед фактическим параметром означает, что передача параметров использует вызов по значению результата. Размещение символа “$” обозначает, что параметр передается с помощью вызова по результату. На Рисунке 4.11 показано влияние различных механизмов передачи параметров на примере программы *main* и подпрограммы *messy*.

После вызова подпрограммы *messy*, переменная *a*[*1*] передается посредством вызова по результату, переменная *a*[*2*] передается посредством вызова по ссылке, переменная *j* в третьем и четвертом аргументе передается с помощью вызова по ссылке, пятый аргумент *a*[*3*] передается с помощью вызова по значению результата, а переменная *k* передается с помощью вызова по результату. Формальный параметр *A* получает заданное значение 10 - значение переменной *a*[*1*], формальный параметр *B* является указателем на фактический параметр *a*[*2*], а формальные параметры *C* и *D* указывают на ячейку памяти фактического параметра *j*. Поскольку формальные параметры *C* и *D* указывают на одну и ту же ячейку памяти, изменения значения ячейки памяти указателем, хранящемся в ячейке памяти формального параметра *C* изменят значение **D* и наоборот. Формальный параметр *E* получает копию значения *a*[*1*]= 10. Формальному параметру *F* присваивается значение 0, потому что значение не копируется в вызов по результату.

Первый оператор берет значение фактического параметра *a*[2] = 10 и значение фактического параметра *j* = 0 и складывает их, чтобы значение *A* стало равно 10. Второй оператор берет значение фактического параметра *j* = 0 и прибавляет значение *E* , которому было присвоено значение 10 (копия значения *a*[3]) и значение *F*, которому было присвоено значение 0, для получения значения 10, которое будет записано обратно в ячейку памяти фактического параметра *a*[2]. Третий оператор

считывает значения переменных *A* и *E*, складывает их для получения значения 20 и сохраняет в блоке памяти фактической переменной *j*. Четвертый оператор считывает значение фактического параметра *j* и вычитает значение фактического параметра *j* для получения значения 0, которое сохраняет в ячейке памяти фактического параметра *j*. Пятый оператор считывает значение фактического параметра *a*[2] и значение фактического параметра *j* и складывает их для получения значения 10, которое присваивается переменной *E*. Последний оператор считывает значение переменной *E*, которому присвоено значение 10,

```
program main( )
{           Integer i, j, k, [6];
i = 0; j = 0; k = 2;
for (i = 1; i =< 5; i++) a[i] = 10;
messy(a[1], &a[2], &j, &j, #a[3], $k);
}

subprogram messy(integer  A, *B, *C, *D, E, F)
{           A = *B + *C;  % A = значение(a[2]) +значение(j) = 10 + 0 =
10
*B = *D + E + F; % a[2] = значение(j) + значение(E) + значение(F) = 0
+ 10 + 0 = 10
*C = A + E; % j = значение(A) + значение(E) = 10 + 10 = 20
*D = *C – *D; % j = значение(j) – значение(j) =  20 – 20 = 0
E = *B + *C;  % E = значение(a[2]) + значение(j) = 10 + 0 = 10
F = E + A; % F = значение(E) + значение(A) = 10 + 10 = 20
}
```

РИСУНОК 4.11 Пример сочетания влияния параметров и совмещения имен.

прибавляет значение переменной *A*, равное 10, и присваивает полученное значение 20 переменной *F*.

После завершения процедуры *messy* обратно копируются только результирующие значения переменных *E* и *F*. Значение переменной *E* копируется обратно в ячейку памяти фактического параметра *a*[3], а значение переменной *F* копируется обратно в ячейку памяти фактического параметра *K*. Окончательными значениями являются $i = 0$, $j = 0$, $k = 20$, $a[0] = 10$, $a[1] = 10$, $a[2] = 10$, $a[3] = 10$, $a[4] = 10$, $a[5] = 10$.

4.6 ОБРАБОТКА ИСКЛЮЧЕНИЙ

Одной из главных задач при выполнении программы является успешное завершение и избежание внезапного прерывания вследствие неисправимых ошибок, таких как “file not found” (“файл не найден”), “invalid memory access” (“недопустимое обращение к памяти”), “divide by zero” (“деление на нуль”), “unable to open a file” (“невозможно открыть файл”) или “array out of bounds” (“выход массива за пределы”). В большинстве случаев ошибки зависят от данных, и не могут быть отнесены к логическим ошибкам. Иногда ошибки могут возникать из-за входных данных или состояния системы. В таких случаях у программы должна быть возможность завершить работу, освободив все выделенные пользователем ресурсы, такие как файлы, буферы и устройства ввода/вывода, а не прервать работу преждевременно, заблокировав ресурсы.

Ошибка может быть системной, например, “file not found” (“файл не найден”), или пользовательской. Ошибки могут быть исправлены операционной системой с помощью *ловушек* - прерываний программы, проверяемых в конце команды. Однако, ловушки находятся на уровне операционной системы, и не смогут определить ошибочное условие. Для того чтобы определить ошибочное условие, чтобы программа не прервала работу преждевременно, необходимо в языках программирования создать обработчика особых ситуаций, определяемых программным устройством.

Обработка особых ситуаций (ошибок) - это абстракция на уровне языка программирования, предпринимающая действие в ответ на ошибочное или нежелательное условие, которое может преждевременно прервать программу или спровоцировать патологическую ситуацию, из-за возникновения которой результат вычисления будет содержать ошибки. Особые ситуации могут быть системными или пользовательскими. Языки, в которых поддерживаются особые ситуации, также могут поддерживать особые ситуации, определенные пользователем. Встроенные особенные ситуации устанавливают заранее определенное логическое условие, которое проверяется в обработчике особых ситуаций. Обработчики особых ситуаций, определяемых пользователем, необходимо объявлять, как *тип особой ситуации*. *Тип особой ситуации* - это статическая логическая переменная, которая проверяется в обращении подпрограммой к цепочке вызываемых подпрограмм. Разные языки используют разные механизмы обработки области видимости особых ситуаций между процедурами, которые их вызывают, и обработчиками особых ситуаций. Одна схема предполагает использование вызова по значению для передачи особой ситуации в подпрограмму обработчика особых ситуаций.

Если обработчик особых ситуаций сможет подтвердить признак состояния особой ситуации, тогда будет выполнена соответствующая подпрограмма или последовательность установленных команд. При отсутствии подходящего обработчика особых ситуаций, управление передается следующей вызывающей подпрограмме более высокого уровня. Перед передачей управления на следующий уровень выполняется вспомогательный блок, за которым следует зарезервированное слово "*finally*". Обратите внимание, что объявление конечного блока не является обязательным.

Если управление находит соответствующего обработчика особых ситуаций на данном уровне, тогда выполняется обработчик особых ситуаций. Иначе, процесс повторяется, и управление переходит на следующий уровень вызывающих подпрограмм в цепочке вызывающих подпрограмм, пока управление не перейдет к главной программе. Если не найден соответствующий обработчик особых ситуаций, тогда программа завершается, после того как будет достигнуто главное тело программы.

Разные языки характеризуются разным синтаксисом для обработчиков особых ситуаций. Абстрактное представление обработчика особых ситуаций выглядит так:

```
<расширенный оператор> ::= try
        <оператор>
        if <выражение1> raise <особая ситуация1>;
        if <выражение2> raise<особая ситуация2>;
        …
        if <выражениеM> raise <особая ситуацияM>;
        обработчики-особых-ситуаций
        {       when <особая ситуация1>: <блок1>
                when <особая ситуация2>: <block2>
                …
                when <особая ситуацияN>: <блокN>;
   }
[finally <блокF>]
```

Обработчик особых ситуаций может выполнять несколько действий, а именно (1) передача управления другой подпрограмме; (2) освобождение ресурсов; (3) исправление источника ошибочного условия и передача управления для повторения действий; (4) возвращение управления

следующей после обработчика особых ситуаций команде, после сообщения программирующему устройству об ошибочном условии; (5) обращение к обработчику особых ситуаций и возвращение к вызывающей подпрограмме более высокого уровня; (6) пропуск данных, вызывающих ошибку; (7) возвращение в вызывающую подпрограмму.

У разных частей программы, выполняющих одинаковый оператор, могут быть разные обработчики особых ситуаций, которые связаны с оператором. У разных операторов может быть общий обработчик особых ситуаций.

Обработка особых ситуаций впервые была применена в PL I. В большинство современных языков, таких как ADA, Java, C++ и Ruby, входит сильная обработка особых ситуаций.

Пример 4.18

В программе на Рисунке 4.12 показана обработка особых ситуаций с использованием предполагаемого синтаксиса без ошибок. Программа открывает файл “myfile”, соответствующий поток данных “mystream” считывается для определения суммы долга для компании, работающей с кредитной картой. Программа выдает встроенную особую ситуацию “file-not-found”, если файл не найден. Существует пользовательская особая ситуация “incorrect_debt”, которая выдается, если остаток на счете больше нуля. Первый обработчик особых ситуаций выводит оператор “Account-file missing”, и возвращается в вызывающую подпрограмму. Второй обработчик особых ситуаций выводит оператор, закрывает поток и возвращается.

```
subprogram illustrate _ exceptions
integer i;
real deposit, account;
file myfile;
stream mystream;
exception incorrect _ debt; % особая ситуация, определенная
пользователем
open _ file(myfile, mystream, read);
exception-handler {
        when file-not-found: write('Account file missing'); return
        }
read(mystream, account);
if (account > 0) raise incorrect _ debt;   % выдать особую ситуацию,
определенную пользователем
exception-handler {  % обработать особую ситуацию, определенную
пользователем
```

```
when incorrect _ debt: write('Incorrect debt');
close(mystream);
return;
```

РИСУНОК 4.12 Пример обработки особых ситуаций.

4.7 НЕДЕТЕРМИНИРОВАННЫЕ ВЫЧИСЛЕНИЯ

Недетерминированные вычисления обеспечивают альтернативный поток управления программой во время выполнения программы. Единственное условие для недетерминированных вычислений заключается в том, что конечное состояние должно предоставлять правильное решение проблемы. Если представить выполнение программы в виде задачи обхода графа, где каждый узел графа - это вычислительное состояние, тогда программа будет недетерминированной, если мы сможем найти более одного маршрута из начального состояния в конечное. Часть вычисления является недетерминированной, если конечное состояние не зависит от порядка выполнения операторов. Выделяют несколько основных свойств программирования, поддерживающих недетерминированные вычисления:

1. Коммутативность операторов в арифметических выражениях и логических выражениях способствует недетерминированному вычислению выражения на более низком уровне. Согласно определению, “коммутативность” не зависит от порядка вычисления. Некоторые из операторов, поддерживающие “коммутативность” - это сложение, умножение, логическое И, логическое ИЛИ. Например, если мы вычисляемы выражение 4 + 5 + 3 + 9 с помощью набора команд промежуточного уровня, тогда его можно представить в виде нескольких вариантов одновременного сложения двух чисел. Конечный результат будет одинаков, потому что сложение по своему характеру ассоциативно и коммутативно. Аналогично, если мы вычисляем логическое выражение $(exp_1 \wedge exp_2 \wedge exp_3)$, включающее логическое И, то не имеет значения, в каком порядке мы вычисляем выражение, результат будет одинаковым, поскольку “логическое И” и “логическое ИЛИ” - это по ассоциативные и коммутативные операторы.

2. Если блок памяти σ^S можно разбить на несвязанное объединение наборов вида $\sigma_1^S \uplus (\sigma_1^S \uplus \sigma_2^S)$, и выполняется последовательность команд C_1 ; C_2 , такая что C_1 выполняется в σ_1^S , а C_2 выполняется в σ_2^S, а σ_1^S постоянна во время выполнения C_1 и C_2. Более того, σ_1^S не

зависит от $\sigma 2^S$; воздействия на них не будут складываться между собой. После команды C_1, $\sigma 1^S$ будет отражена в новом блоке памяти $\sigma 1^S$; а после команды C_2, $\sigma 2^S$ будет отражена в новом блоке памяти $\sigma'2^S$. Таким образом, общее воздействие последовательности команд C_1; C_2 создаст новый блок памяти $\sigma's = \sigma 1s \uplus (\sigma 1s \uplus \sigma 2s)$. Если мы изменим порядок команд, то они выдадут такой же конечный блок памяти $\sigma's = \sigma 1s \uplus (\sigma 1s \uplus \sigma 2s)$. Например, если мы возьмем две команды присваивания $x = 4$; $y = 5$, такие что x и y не являются псевдонимами, тогда они будут изменять две разные ячейки памяти, и могут быть выполнены в любом порядке.

1. 3. Если мы берем команду выбора, например, условный оператор (if-then-else), и записываем его в форме, в которой все условные операторы освобождаются от зависимости, создаваемой потоком команд, и рассматриваются как равновозможные для вычисления, тогда изменение порядка выполнения команд не влияет на конечный результат.

Давайте разберем пример из известной статьи Дейкстра о защищенных программах:

if (x >= y) smaller = y;
elseif (y >= x) smaller = x;

Верхний код находит меньшее из двух значений, и может быть преобразован в два семантически эквивалентных кода, как показано в Таблице 4.1.

Оператор является недетерминированным, потому что существует два способа вычисления оператора и получения одинакового конечного результата. Если $x > y$, тогда программа будет вычислена в следующем порядке: логическое условие $y >= x$ не будет выполнено, если будет проверяться первым, и будет вычислен первый оператор. В общем случае недетерминированную условную команду можно записать с помощью следующей формы абстрактного синтаксиса:

if { <*условное-выражение*$_1$> —→ <*команда*$_1$> |
<*условное-выражение*$_2$> —→ <*команда*$_2$> |
… |
<*условное-выражение*$_n$> —→ <*команда*$_n$>
}

В вышезаписанной абстрактной конструкции нет зависимости от

последовательности команд управления, это означает, что условные команды вида (<*условное-выражение*$_i$> → <*команда*$_i$>) ($1 \leq i \leq N$) равновозможны. Каждый условный оператор связан с другим при помощи логического ИЛИ,

обозначаемого символом '|'. Символ '→' означает, что часть команды следуют выполняется после успешного вычисления логического выражения. По крайней мере, для получения результата должно быть выполнено одно из логических выражений <*логическое-выражение*$_i$> ($1 \leq i \leq N$). Если все логические выражения не вычисляются, тогда выдается ошибка. После успешного выполнения первого логического выражения для получения возможного решения выполняется соответствующая команда, а другие условия не проверяются.

ТАБЛИЦА 4.1 Семантически эквивалентные программы

`if (X > = Y) then smaller = Y |`	`if (Y > = X) then smaller = X |`
`if (Y > = X) then smaller = X`	`if (X > = Y) then smaller = Y`

4.7.1 Защищенные команды

Дейкстра предложил модель недетерминированного программирования, в которой каждое логическое выражение было самым *слабым необходимым условием,* которое должно быть истиной для того, чтобы передать выполнение условному оператору, и, как только выполнение программы переходило к одному из условных операторов, другие условные операторы не проверялись, хотя другие условные выражения также могли быть успешно выполнены. Дейкстра назвал такие условные выражения *защитой*, а условные операторы - *защищенными командами*. Защищенная команда состоит из двух компонентов: *защиты* и *команд*. Защита - это логические выражения, которые не изменяют блок памяти. Команды содержат операторы присваивания и изменяют *предусловие* на *постусловие*.

Защищенные команды обладают следующими свойствами:

1. Защита - это слабейшие условия для перехода к защищенной команде.

2. Защита необходимое, но не достаточное условие. Защищенные команды могут быть проверены в любом порядке.

3. Если защита успешно проверена, тогда осуществляется соответствующая команда, а другие защиты прерываются.

4. Так как выполняется только одна из защищенных команд, и защита является необходимым, но недостаточным условием, то защита может успешно выполниться без получения результата в части выполнения команд. Решение может находиться в другой защищенной команде. Однако, другие защищенные команды не проверяются после первого успешного выполнения защиты. Таким образом, программа с защищенной командой является неполной: она может не найти решения, хотя решение будет существовать.

Пусть дано постусловие Q и оператор S, обозначение $wp(S, Q) = P$ говорит о том, что P - *самое слабое предусловие* для недетерминированного оператора, а Q - это *постусловие.* С другой стороны, дано предусловие P и оператор S, Q - *самое сильное постусловие.* Выделяют несколько качеств самых слабых предусловий: например, если $wp(S, Q) = P_1$ и $wp(S, R) = P_2$, тогда $wp(S, Q \wedge R) = P_1 \wedge P_2$. Аналогично, $wp(S, Q \vee R)$ равно $P_1 \vee P_2$; $wp(skip,Q) = Q$ для всех постусловий; а $wp(abort, Q) = fail$, это означает, что начальное условие не должно быть выполнено (*fail*), для того чтобы получить конечное условие, имеющее значение *abort* (прерываение).

Дана последовательность операторов $S_1; S_2$ и конечное условие Q, постусловие оператора S_1 - это самое слабое предусловие оператора S_2. Таким образом, можно записать уравнение $wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$.

Существует два вида конструкций в защищенных командах: выбор защищенной команды и итерация защищенной команды. Выбор защищенной команды имеет вид:

```
if
{     <защита1> —→ <команда1> |
      <защита2> —→ <команда2>|
      …
      <защитаN> —→ <командаN>
}
```

Символ ‘→’ отделяет защиту от соответсвующей команды в каждой защищенной команде, а символ ‘|’ разделяет защищенные команды. Структура, представленная выше, проверяет защиты в любом порядке.

Если защита проверена успешно, тогда выполняется соответствующая команда, а оставшиеся защищенные программы не проверяются. Если защита выполнена не успешно, тогда проверяется другая защита, до тех пор, пока все защиты не будут проверены или одна из защит будет проверена успешно.

Итеративная конструкция защищенной команды имеет вид:

```
цикл
{     <защита1> —→ <команда1> |
      <защита2> —→ <команда2> |
      …
      <защитаN> —→ <командаN>
}
```

Итеративная конструкция сохраняет зацикленность до тех пор, пока все защиты после проверки окажутся неверными. Если любая из защит успешно пройдет проверку, тогда будет выполнена соответствующая команда, и заново начнется следующий итерационный цикл. Если управление выходит из итерационной конструкции, тогда все защиты не прошли проверку, это означает, что постусловие Q равно $P \wedge \neg$ *защита*$_1$ $\wedge \neg$ *защита*$_2 \wedge$ ... $\wedge \neg$ *защита*$_N$. Используя теорему де Моргана, можно записать конечное условие как $P \wedge \neg$ *(защита*$_1 \vee$ *защита*$_2 \vee$ ... $\vee$ *защита*$_N$*)*. Итерационная конструкция прибавляет логическое условие $\neg$ *(защита*$_1 \vee$ *guard*$_2 \vee$ ... $\vee$ *защита*$_N$*)* к предусловию.

Пример 4.19

В следующем примере, взятом из фундаментальной работы Дейкстра, посвященной недетерминированным программам, мы увидим пример недетерминированной итерации. Предположим, что мы хотим расформировать последвательность из четырех чисел *a0, a1, a2*, and *a3*. Для решения этой задачи мы можем воспользоваться недетерминированной итерацией.

```
цикл
{       a0 > a1 → замена(a0, a1)|
        a1 > a2 → замена(a1, a2)|
        a2 > a3 → замена(a2, a3)|
}
```

4.7.2 Пошаговое создание программы

Предполагаемое конечное условие и аксиоматическая семантика операторов могут использоваться для нахождения самого слабого предусловия, которое необходимо проверить для предыдущего оператора, и процесс может повторяться в обратном порядке для поэтапного создания недетерминированной программы.

Дано конечное условие и начальное условие, определяется различие. Если различие можно представить как конъюнктивную форму $B_1 \wedge B_2 \wedge ... \wedge B_n$, тогда можно использовать итерационную конструкцию, как указано далее: защита определяется согласно закону де Моргана за счет взятия отрицания и использования правил о самых слабых предусловиях, описанных в предыдущем разделе. Из закона де Моргана следует, что *not* $(B_1 \wedge B_2 \wedge ... \wedge B_n)$ представляет защита$_1$ как *not* (B_1), защита$_2$ как *not* (B_2), и т.д. Аналогично, если есть различие в дизъюнктивной форме $B_1 \vee B_2 \vee ... \vee B_n$, тогда выбор защищенной команды можно использовать так, что защита$_1$ представляется как $wp(command_1, B_1)$, защита$_2$ как $wp(command_2, B_2)$, и т.д., таким образом, что каждое самое слабое предусловие $wp(command_i, B_i)$ включается в начальное условие P.

Пример 4.20

Давайте разберем случай из Примера 4.2, и будем размышлять в обратном направлении для того, чтобы создать недетерминированную программу. Конечное условие для рассортированной последовательности имеет вид $a0 \leq a1 \leq a2 \leq a3$, в конъюнктивной форме записывется как $(a0 \leq a1) \wedge (a1 \leq a2) \wedge (a2 \leq a3)$. Смотря на конъюнктивную форму, мы знаем, что необходимо использовать недетерминированную итерационную конструкцию. Защита$_1$ будет иметь вид *not* $(a0 \leq a1)$, это означает, что выражение защита$_1$ можно представить как $a0 > a1$. Аналогично, можно получить другие защиты в виде $a1 > a2$ and $a2 > a3$.

4.8 ПРОГРАММЫ КАК ДАННЫЕ

В языках программирования, в общем случае, данные отделены от команд, и данные - это нечто, что преобразовывают, в то время как команды не изменяются. Однако, во многих случаях, например, в случае искусственного интеллекта, появляется необходимость создания программы как объекта данных с последующим преобразованием в

программу, которая может выполнять действия над данными.

Если программа разрабатывается в виде данных, а затем преобразовывается в программу, тогда программа будет рассматриваться как *объект первого уровня. Объект первого уровня* можно создать во время обработки, связать с идентификатором, передать как параметр или вернуть как результат вычисления.

Точно так же, во многих случаях программе необходимо анализировать или обрабатывать другую программу как данные. Например, редактору необходимо обработать другую программу как данные. Язык можно записать так, что он будет делать выводы о своем собственном выполнении в абстрактной области. Все это примеры, когда программа рассматривается как данные. Процесс представления программы как данных с помощью некоторых структур данных называется материализацией.

4.8.1 Функции как объекты первого уровня

Декларативные языки, такие как ML, Haskell, Lisp, Scheme и Prolog обробатвают программы как объекты первого уровня. Например, в большинстве функциональных языков программирования есть режим формирования функции, который берет имя функции и соответствующие аргументы в виде данных и преобразовывает данные в активную функцию. Функция (apply ‘first ‘(Arvind Programs)) в языке программирования Lisp возьмет два аргумента данных - имя функции *first* и список *‘(Arvind Programs)* и преобразует данные в функцию *(first ‘(Arvind Programs))*, которая вернет элемент данных *‘Arvind*. Обратите внимание, что элементы данных в семействе языков Lisp имеют дополнительный знак кавычки для того, чтобы не спутать их с функцией. Используемый здесь синтаксис применяется в языках Lisp и Scheme, а отличается от синтаксиса других функциональных языков программирования.

4.8.2 Метапрограммирование и рефлексивность

Метапрограммы - это программы, которые управляют другими программами как своими данными или анализируют их. Если язык может написать метапрограммы для своих собственных программ, то тогда его можно назвать *рефлексивным языком*. Декларативные языки, в которых поддерживаются объекты первого уровня, также поддерживают метапрограммы. Если встречается объявление, тогда метапрограммы обновляют среду, а если встречается строка команд, тогда она

преобразуется в программу и выполняется. Примером метапрограммы является *абстрактный интерпретатор*, который делает выводы о других программах в *абстрактной области,* чтобы проанализировать поведение программы во время ее выполнения. Пример абстрактной области - область типов, в которой контретные значения преобразуются в соответствующие им типы. Еще одним примером метапрограммы является написание механизма, выполняющего программу языка. Метапрограммы также используются для создания других программ. Метапрограммы используются для создания шаблонов программ, которые легко заполняются программистом, что сокращает время и затраты на создание кода. Метапрограммирование поддерживается такими языками как Ruby, Scala, Lisp и Prolog.

4.9 ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Размер программного обеспечения увеличивается в связи с задачами комплексной автоматизации, оно становится императивным, это значит, что при разработке нового программного обеспечения повторно используется предыдущее программное обеспечение, для того чтобы избежать дублирования, сократить время разработки программного обеспечения и уменьшить количество ошибок в программном обеспечении, возникающих при его создании. С помощью различных парадигм можно эффективно закодировать различные частные задачи. Разработка сложного программного обеспечения требует интегрирования программ, разработанных с использованием нескольких парадигм программирования.

Разные языки используют разные парадигмы программирования (Приложение I). Разные парадигмы программирования отличаются по реализации абстракций поддерживаемых данных и управления. Например, императивные языки традиционно используют несколько индексированных массивов или векторов для реализации совокупности данных. Декларативные языки традиционно используют неизменяемые последовательности для реализации совокупности элементов данных. Парадигмы отличаются даже в абстракциях данных и управления, которые они поддерживают. Некоторые абстракции данных и абстракции управления больше подходят для различных классов областей задач, необходимых при разработке сложного программного обеспечения. Интеграция нескольких парадигм требует взаимозаменяемости - взаимодействия с другими языками и способности обрабатывать

библиотеки, созданные в других языках.

Современное программное обеспечение использует оба подхода: использование внешних готовых для использования библиотек и взаимодействие с программным обеспечением, разработанным на другом языке. Библиотека - это набор выполняемых функций, которые сделали общедоступными и которые можно импортировать в разрабатываемое программное обеспечение. Библиотеку можно импортировать с помощью команды “import<библиотека-название>.”

4.9.1 И вновь о способности к взаимодействию

Способность языков к взаимодействию позволяет вызывать программу, написанную на другом языке, программировать функции, имеющиеся в другом языке, а также повышать эффективность выполнения.

Основной проблемой взаимодействия языков является состыковка параметров вызывающей программы и вызываемой программы. Существует два способа решения этой проблемы: (1) разработка общего промежуточного языка для того, чтобы все языки переводились на этот общепринятый язык или (2) разработка интерфейса преобразования формата данных посредством двух функционально совместимых языков. Оба этих способа были использованы.

Общий язык промежуточного программного обеспечения обеспечивает взаимодействие путем указания *типовой системы* и *метаданных*. Типовая система каждого взаимодействующего языка переводится на этот *общепринятый тип* для обеспечения работы интерфейса. *Метаданные* обеспечивают единый механизм функционирования для перевода информации о типе с других языков на общепринятый тип в стандартном языке интерфейса, а также сохраняют и извлекают информацию о записях общего типа. *Метаданные* включают в себя таблицы и структуры данных, содержащие сведения о скомпилированной программе. Таблица содержит сведения о классах, полях и их типах, экспортируемых типах, а также ссылки на другие таблицы метаданных на нейтральном языке интерфейса, в результате чего пропадает необходимость в определяющем языке конкретного интерфейса и настраиваемом коде взаимодействия.

Несмотря на поддержку, предоставляемую системой общего типа и метаданные для перевода одного типа в другой, разные абстрактные данные поддерживаются разными языками, и при использовании полной функциональности трудно отличить процесс реализации одних абстрактных данных от другой реализации таких же абстрактных данных.

Общеязыковой подход был принят на платформе .NET Framework для многих языков программирования, поддерживаемых корпорацией Майкрософт, таких как, C#, C++ и Visual Basic. Платформа.NET Framework дает описание спецификации общеязыковой инфраструктуры (СОИ), который описывает особенности и правила общепринятого языка. XML стал стандартным общеязыковым интерфейсом для языков, основанных на интернет-технологиях и базах данных.

4.10 ТЕМАТИЧЕСКОЕ ИССЛЕДОВАНИЕ

В этом разделе мы сталкиваемся с несколькими современными языками программирования и обсуждаем поддерживаемые ими абстракции. Следует отметить, что не все языки поддерживают все абстракции данных, рассматриваемых в этой главе. В качестве простого примера возьмем языки ADA 2012, C++, C#, Fortran 2009, Java, Modula-3, Ruby, и Scala. ADA — богатый язык, который поддерживает императивный стиль программирования наряду с объектами и модулями. Fortran 2009 является императивным языком программирования со структурным научным блоком, который поддерживает понятия об объектах. C++ — это сочетание императивного и объектно-ориентированного программирования. Java также поддерживает парадигму императивного и объектно-ориентированного программирования. Lisp богат абстракциями управления и данных, поддерживая парадигму функционального программирования, наряду с ограниченным объемом императивного стиля программирования. Modula-3 поддерживает модули явного программирования, а также объектно-ориентированного программирования. Ruby объединяет понятия модулей и класса, поддерживая интегрированную функциональную и объектно-ориентированную парадигму программирования.

Scala — современный мультипарадигмальный язык, который интегрирует парадигмы функционального и объектно-ориентированного программирования. Язык Prolog, представляющий собой логический язык программирования, не включен в этот список, так как требует объяснения многих концепций, таких, как объединение и дерева типа И-ИЛИ, которые отличаются от абстракций в других языках программирования и рассматриваются подробно в Главе 10.

4.10.1 Абстракции данных в языках программирования

В современных языках программирования многие абстракции данных встречаются довольно часто. Почти все современные языки поддерживают составные типы, коллекции и расширяемые структуры данных. Мультипарадигмальные языки, поддерживающие как декларативное, так и императивное программирование, разграничивают изменяемые и неизменяемые объекты, поддерживая оба типа этих объектов. Изменяемые объекты могут быть обновлены с удалением, а неизменяемые объекты поддерживают назначенное один раз свойство.

4.10.1.1 Единичная абстракция

Такие единичные объекты, как целое число, булево выражение, реальное число и символьное, происходят из математического мира и являются вполне распространенными. Во многих языках, таких как ADA, Pascal, Modula, C и C++, пользователь может определить счетное множество. Тем не менее, счетные множества были удалены из Oberon — преемника Modula-2. Объявление поддиапазона и его использование в массивах являются стандартными в ADA, Fortran, ALGOL, Pascal, Modula-2. Однако тип диапазона был удален из C, C++ и Oberon из-за ограниченного использования этого объекта, а нижняя граница массива был установлена на «0.»

Тип набора поддерживается в Pascal, Modula-2 и в более позднем языке Scala. Однако, во время разработки коммерческого программного обеспечения, программирование на основе набора не было одобрено во многих языках из-за ограниченного использования программистами. Ниже приводится пример типа набора с помощью синтаксиса Modula-2

```
Type week = (Mon, Tue, Wed, Thu, Fri, Sat, Sun)
Var workdays: set of week
```

Тип неделя = (пн, вт, ср, чт, пт, сб, вс)
Переменная будни: **набор** недели;

Объявлен первый тип перечисления *неделя*. Переменная *будни* объявляется как тип *неделя*. Переменная рабочий день может принимать любое значение, которое является подходящим подмножеством недели.

Scala использует зарезервированное слово «Set (набор)» для создания

набора. Например, формулировка var major = Set(“CS,” “Math,” “Finance”) создаст основную переменную в качестве набора из трех элементов.

Тип перечисления может также быть поддиапазоном, как показано ниже в примере из ADA:

```
Type    week is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
Type    year is 1..12;
Subtype  weekend is week range Sat..Sun;
Subtype  workingdays is week range Mon..Fri;
Subtype  fall_semester is year 8..12
Subtype  summer is year 6..8
```

Типом неделя **является** (пн, вт, ср, чт, пт, сб, вс);
Тип год **является** 1**..** 12;
Подтип выходными днями **является диапазон** недели **сб**.. вс;
Подтип рабочими днями **является диапазон** неделя **пн**.. пт;
Подтип осенним семестром **является** год 8**..** 12
Подтипом лето **является** год 6**..**

ADA также поддерживает подтип, который наследует все свойства исходного типа, и рассматривается подробно в Главе 7. Oberon расширяет тип системы Modula-2 путем включения нескольких арифметических типов короткое целое ⊆ целое число ⊆ длинное целое число ⊆ реальное ⊆ реальное. Значение включенного типа можно присваивать переменным, включая тип. Например, если N - целое число, а M является длинным целым числом, то верным будет M = N. Это, по сути, случай приведения, как описано в Главе 7.

Scala или его расширенный вариант Escala – это современный мультипарадигмальный язык, который поддерживает интеграцию функционального программирования, объектно-ориентированного программирования и на основе событийно-ориентированного программирования (поддерживается в Escala), поддерживает такие базовые типы, как целые числа и булево выражение. Кроме того, он обрабатывает объявление строки.

В дополнение к регулярной единичной абстракции, многие языки поддерживают типы, необходимые для такого параллельного выполнения, как мониторы, сопрограммы и семафоры, которые рассматриваются в Главе 8.

4.10.1.2 Составная абстракция

Составными объектами являются кортежи. Именованные кортежи называются «структ» в языках ALGOL, C и C++, а в таких семействах языков, как ADA, Pascal и Modula – их называют «запись». «Структ» в языках C и C++ пишется под зарезервированным словом «struct», после чего следует название структуры, а затем различные поля идентификатора. Группа полей находится под левой и правой фигурными скобками. Возьмем, к примеру, запись *студент*, содержащую три поля: *имя, факультет, степень*. Если предположить, что все три поля являются строками, то составная абстракция на языке C++ с использованием «struct» будет выглядеть следующим образом:

```
struct student {
        char* name;
        char* department;
        char* degree;}
```

```
структ студент{
        символ* имя;
        символ* факультет;
        символ* степень; }
```

где символ * используется для обозначения массива символов. Вышеуказанное объявление говорит о том, что это имя является ссылкой, которая указывает на массив символов. Аналогичным образом, факультет является ссылкой, указывающей на массив символов. Ссылка указывает на первое расположение массива в памяти. Кортеж можно инициализировать путем связывания кортежа конкретных значений с помощью переменной типа студент следующим образом:

```
student plstudent = {“John”, “Computer Science”, “BS”};
```

```
студент множстудент = {«Джон», «Информатика», «бакалавр наук»};
```

В ADA представлен именованный кортеж с использованием «записи»:

```
Type PERSON is
record
```

```
        Name :  STRING(1..30);
        Department :      STRING(1..20);
        Degree : STRING (1..20)
end record;
```

Заданный пользователем тип можно использовать в определении другого типа. Например, следующее определение записи *студент* может быть расширено к новому определению записи *выпускник* путем добавления поля *год выпуска*. В Oberon это можно записать следующим образом:

```
student = RECORD
                name: ARRAY 30 of CHAR;
                department:  ARRAY 20 of CHAR;
                degree:      ARRAY 20 of CHAR;
        END
alumni = RECORD
                person: student;
                graduationYear: INTEGER;

        END
```

```
студент = ЗАПИСЬ
                имя: МАССИВ
                30 СИМВОЛ;
                факультет: МАССИВ
                20 СИМВОЛ; ученая
                степень:
                МАССИВ 20
                СИМВОЛ;
          КОНЕЦ
выпускник = ЗАПИСЬ
                субъект:
                студент; год
                выпуска:
                ЦЕЛОЕ
                ЧИСЛО;
          КОНЕЦ
```

Многие языки, такие как ADA, Pascal, и Modula-2, поддерживают *вариантную запись*. *Вариантная запись* является составной структурой,

которая состоит из двух частей: *фиксированной части* и *переменной части.* Переменной частью являются взаимоисключающие группы полей, которые могут быть размещены на те же самые области памяти. Вариантная запись интерпретирует совместно используемую область памяти в вариантной части записи, на основе значения пронумерованной переменной. В случае, если значение пронумерованной переменной изменяется во время выполнения, тогда та же самая область памяти может быть истолкована как различный тип данных. Неправильное толкование может вызвать ошибку нарушения типа, как описано в Главе 7.

Пример 4.21

Следующий пример, представляющий составной объект в синтаксисе ADA, демонстрирует объявление вариантной записи. Тип Date (дата) является записью, имеющей три поля типа целое число с различными диапазонами. Тип assignment (назначение) является вариантной записью, содержащей имя записи в качестве одного из полей. Поля problem, solved (проблема, решена) и date_assigned (дата назначена) являются фиксированными полями, а поле, начинающееся с пронумерованного типа submission (представление), является вариантным полем. Существует три возможности: submitted, to be, и missed (представлено, будет и пропущено). Если значение submission (представление) уже submitted (представлено), то вариантным полем становится тип целого числа с диапазоном 1.. 100. Если значение предствление – to be (будет), то поле обрабатывается как тип записи date (дата). Если значение представления - missed (пропущено), то типом поля является целое число с меткой 0. Обратите внимание, что у вариантной записи есть различные диапазоны и различные типы. Однако, интерпретация вариантного поля зависит от значения поля submission (представление).

```
type Date is
        record
                month: INTEGER range 1..12;
                day:  INTEGER range 1..31;
                year: INTEGER range 2012..2015
        end record

    type status is (submitted, tobe, missed);

    тип  Дата  является
```

```
            записью
                месяц: ДИАПАЗОН
                целых чисел 1.. 12;
                день:  ДИАПАЗОН
                целых чисел 1.. 31;
                год: ДИАПАЗОН
                целых чисел 2012..
                2015
            конец записи
  типом статуса является (запущено, будет, пропущено);
```

```
type assignment (submission : status) is
        record
                Problems: Integer range 1..10;
                solved: Integer range 1..10;
                date_assigned: DATE;
                case submission is
                        when submitted => score:
                                Integer range 1..100;
                        when tobe => expected: Date;
                        when missed => score: Integer := 0
                end case
        end record
```

```
  тип присваивание (запуск: статус)
    является записью
            Задачи: Диапазон
            целых чисел 1.. 10;
            решено: Диапазон
            целых чисел 1.. 10;
            присвоена дата: ДАТА;
            запуском кейса является
                    когда запущено => оценка:
                Диапазон целых чисел 1.. 100;
                    когда будет => ожидается: Дата;
                    когда пропущено => оценка: Целое
                    число: = 0
                    конец кейса конец записи
```

ADA, Pascal, и Modula-2 поддерживают вариантную запись. Однако, вариантная запись была исключена из Oberon и явно не присутствует во многих языках из-за ошибки нарушения типа в переменной части, как описано в Главе 7.

Prolog и Scala поддерживают динамические кортежи формы (1, 2, 3). Scala может создавать кортежи, используя команду new tuple<no-of-entries> '('<tuple-values>')'. Например, утверждение может быть val instructors = new Tuple2("Arvind," "Paul"), а к индивидуальным элементам можно обратиться через instructors._1 и instructors._2.

4.10.1.3 Совокупность элементов данных

Совокупности моделируются в виде последовательностей, которые реализуются с помощью индексируемых массивов, ассоциативных списков или векторов. Массивы могут быть объявлены с помощью (1) использования размера массива, (2) присваивания значений, хранящихся в разных измерениях, или (3) создания неровного массива, который может носить неопределенные расширяемые элементы данных в каждом измерении.

Пример 4.22

Например, возьмем различные схемы объявления массива в C#.

```
int [] x = новый int [5]; //единичный двумерный массив размером 5
int [] y = новый int [] {1, 30, 44, 33, 8}; //номером является размер
//из элементов
int[] y = {1, 30, 44, 33, 8}; //номером является размер
//из элементов
int [,] z = новый int [2, 3]; Двумерный массив
int [,] w = {{1, 2, 3}, {4, 5, 6}}; 2 × 3 матрица
int [] [] неровный1 = новый int [6] []; //неровный массив с 6 строками неровный1 [0] = новый int [3] {10, 20, 30};
неровный1 [1] = {1, 30, 44, 33, 8};
```

Аналогично, массивы Ruby могут быть объявлены для определенного размера или неопределенного размера, который может быть расширен динамически следующим образом:

x = Array.new # создать новый массив динамически неопределенного

размера
x = Array.new(3) # создать массив размером 3
x[0] = Array.new # первый элемент — снова массив
x[1] = [4, 5, 6, 7] # второй массив имеет размер 4

x = **Массив.новый** # создать новый массив динамически неопределенного размера x = **Массив.новый**(3) # создать массив размера 3
x [0] = **Массив.новый** # первый элемент снова является массивом x [1] = [4, 5, 6, 7] # второй массив имеет размер 4

Ruby также использует такие абстрактные операции, как pop (вытолкнуть), push (поместить), shift (сдвинуть), и unshift (отменить сдвиг) в качестве массива, и это является ничем иным, как индексируемой последовательностью. Операция pop (вытолкнуть)удаляет последний элемент из последовательности. Операция push (поместить) добавляет элемент в конце последовательности. *Операция* shift (сдвинуть) удаляет первый элемент из последовательности и перемещает все элементы массива на одну позицию к первому элементу, чтобы заполнить пустое пространство. Операция *unshift* (отменить сдвиг) перемещает все элементы к последней позиции и вставляет заданный элемент в начале массива.

Пример 4.23

Давайте введем следующие формулировки в интерактивном Ruby:

p = [10, 20, 30, 40] # Создать массив из четырех целых чисел
p.pop # Удалить последний элемент, а p будет иметь значение [10, 20, 30]
p.push(80) # Вставить 8 - в конце, а p будет иметь значение [10, 20, 30, 80]
p.shift # Удалить первый элемент, а p будет иметь значение [20, 30, 80]
p.unshift(100) # Вставить 100 как первый элемент, а p будет [100, 20, 30, 80]

p = [10, 20, 30, 40] # Создать массив из четырех целых чисел
p. **выталкивать**# Удалить последний элемент, а p будет иметь

значение [10, 20, 30]

p. **помещать**(80) # Вставка 8 - в конце, а p будет иметь значение [10, 20, 30, 80]

p. **сдвигать**# Удалить первый элемент, а p будет иметь значение [20, 30, 80]

p. **передвигать(**100) # Вставить 100 как первый элемент, а p будет [100, 20, 30, 80]

Языки, поддерживающие парадигмы императивного программирования, такие как ADA, Fortran, C, C++, C#, Java, Pascal, Modula-2, и Modula-3, поддерживают массивы. ADA также поддерживает полудинамические массивы. Объектно-ориентированные языки, такие как Cak, C#, Scala, и Ruby, поддерживают динамические массивы, так как массивы могут быть представлены как и любой другой объект, который может быть создан динамически. Динамические языки, такие как Lisp и множество реализаций Prolog, также поддерживают динамические массивы. Динамические массивы размещаются в неупорядоченном массиве программы. Некоторые примеры из синтаксиса объявления массивов на разных языках программирования приведены в Таблице 4.2.

Такие языки, как Modula-3 также поддерживают порядковый номер типа в объявлении массива. Например, массив в Modula-3, также может быть представлен *как* ARRAY *[«A».. «D»] ИЗ ЦЕЛОГО ЧИСЛА,* где *«A»* сопоставляется с индексом 1, а *«D»* с индексом 4. Этот массив эквивалентен ARRAY *[1.. 4] ЦЕЛОГО ЧИСЛА*.

Таблица 4.2 Объявление массива в отдельных языках

Язык	Описание массива
ADA	**array**(1..5) **of INTEGER**
C, C++:	**int** x[5]
C#	**int**[] x = **new int**[5];
Java	**int**[] x = **new int**[5];
Modula-3	**VAR** x : = **ARRAY** [1..5] **OF INTEGER**
Ruby	x = **Array.new**(5)
Scala	**var** x = **new Array [integer]**(30)

4.10.1.4 Расширяемые элементы данных

Почти все современные языки, которые поддерживают указатель, явно или неявно, на уровне осуществления, могут создать при помощи указателей любую рекурсивную структуру данных, такую как связные

списки, деревья и графики. Указатели могут быть сильно связаны явным образом с рекурсивными структурными данными, как в Pascal и Modula-3. Во многих языках, таких как ADA, C и C++, указатели поддерживаются в качестве независимого объекта, который может ассоциироваться с любой структурой данных. Доступ к элементу в структуре данных на основе указателя осуществляется с помощью dereferencing (удаление ссылок) *указателя* — механизма для чтения ячейки памяти, на которую показывает указатель. Такие декларативные языки, как Lisp и Lisp, вставляют указатели в реализацию языка и не позволяют программистам напрямую манипулировать указателями; структурные данные могут быть расширены с помощью операторов ядра более высокого уровня, таких как *cons, insert* и *append*. Эти операции рассматриваются в Главах 9 и 10.

Объявление связного списка в ADA*:*

```
type List_Pointer is access My_List;
        type My_List is
                record
                        Info : INTEGER;
                        Next : List_Pointer;
                end record;
        Start : List_Pointer; — Всегда указывает на начало списка
        Last : List_Pointer; — Указывает на конец списка
```

```
тип Указатель_Списка является доступом к Мой_Список;
тип Мой_Список является записью
Информация: ЦЕЛОЕ ЧИСЛО;
Следующий: Указатель_Списка;
конец записи;
Начало: Указатель_Списка; — Всегда указывает на начало
списка Последний: Указатель_Списка; — Указывает на
конец списка
```

Объявление связного списка в C и C++:

```
struct My_List
{       int info;
        My_List* next;
```

```
}
 структ Мой_Список
 {int информация; Мой_Список * далее;
 }
```

Объявление связного списка в Modula-*3:*

Modula-3 поддерживает тип ссылки, а расширяемые объекты данных строятся с использованием указателей.

```
TYPE
My_List = RECORD
info: INTEGER;
                next: Ref My_List
END
```

```
информация: ЦЕЛОЕ ЧИСЛО;
Следующий: Ссылка Мой_Список
КОНЕЦ
```

Объявление связного списка в Scala*:*

Scala - это мультипарадигмальный язык высокого уровня, который скрывает использование указателей на уровне программиста. Его список больше похож на Lisp и Prolog. Список в Scala может быть представлен разными способами, как показано ниже. Можно объявить подпись для списка, или можно автоматически сделать вывод на заданное значение.

```
val num : List[Int] = List(1, 2, 3) // num - это список целых чисел
val num = List(1, 2, 3) // Выводимый тип списка
val num = List()//Num — пустой список, который может быть
расширен позже
val nested = List (List (1, 2, 3), List(4, 5, 6)) // Вложенный список
```

```
 val num: Список [Int] = список (1, 2, 3) / / num является
 списком целых чисел
 val num = Список (1, 2, 3) / / Тип списка выведен
 val num = Список() / / Num является пустым списком, который
 может быть расширен позже
 val гнездовой= Список (Список (1, 2, 3), Список (4, 5, 6))
```

Соединение информационных объектов со списком задается оператором '*::*', таким образом, Список (1, 2, 3) также может быть представлен как *Val num = 1:: Список (2, 3)*. Многие языки, такие как Lisp, Ruby, и Scala, поддерживают пары ключевого значения. Эти пары ключ-значение, называются ассоциативными картами на Lisp, картами в Scala, и хэшем в Ruby.

4.10.2 Управление абстракциями в языках программирования

4.10.2.1 Мутации

Языки, поддерживающие парадигмы императивного программирования, поддерживают изменяемые объекты и деструктивные обновления. Например, такие языки, как ADA, Java, C++, C#, Fortran, и Pascal, поддерживают деструктивное обновление. Функциональные языки программирования поддерживают неизменяемые объекты. Кроме того, многие функциональные языки программирования, такие как семейство Lisp, Scala и Ruby, также поддерживают изменяемые объекты и деструктивные обновления в дополнение к неизменяемым объектам. Lisp поддерживает деструктивное обновление глобальных переменных, а также позволяет переменным индекса внутри оператора цикла быть привязанными к различным значениям внутри итерации. Такие мультипарадигмальные языки, как Ruby и Scala , поддерживают деструктивные обновления, а также неизменяемые и изменяемые объекты. Как правило, связные списки и строки рассматриваются как неизменяемые объекты, а такие индексируемые последовательности, как массивы, рассматриваются как изменяемые объекты.

4.10.2.2 Условные операторы

Условные операторы в языке программирования обычно являются стандартными. Почти все языки поддерживают вложенного условного оператора. Большинство языков также поддерживают оператора варианта. Некоторые языки, такие как Haskell и некоторые языки программирования параллельной логики типа Parlog и GHC поддерживают средства защиты. У языков функционального программирования есть функциональная версия условного оператора. Lisp также имеет общее назначение условного оператора, который группирует несколько условий в том же условной конструкции.

4.10.2.3 Итерации и Итераторы

Почти все языки программирования, которые поддерживают парадигму императивного программирования, явно поддерживают переменную индекса на основе цикла «ДЛЯ» и бессрочных итераций, таких как цикл с условием продолжения и цикл «ПОКА». Некоторые языки, такие как C, имитируют цикл «ДЛЯ» с помощью цикла «ПОКА». Многие функциональные языки программирования, такие как Lisp, Ruby, и Scala, поддерживают неопределенные итерации, такие как цикл «ПОКА» и цикл «ДЛЯ» для неоднократного вызова функций. Функциональные языки, поддерживающие деструктивное обновление переменных, также поддерживают итерационные циклы. Ruby и Scala являются примерами таких мультипарадигмальных языков. Scala поддерживает цикл с условием продолжения и цикл «ПОКА», цикл «ДЛЯ» и итераторов; а Ruby поддерживает цикл с условием продолжения, цикл «ДЛЯ», неопределенный цикл и итераторов. Неопределенный цикл сохраняет итерации до момента выхода с помощью условия выхода. Поскольку каждый элемент данных в Ruby является объектом, итераторы и циклы рассматриваются как методы. Ниже приводится пример итерации конструкций в Ruby .

```
[4, 5, 6].foreach {|i| puts i} will write 4, 5, and 6 in that other.
10.times {|i| puts i} will write 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 in
 separate lines
```

```
[4, 5, 6].длякаждого {| i | выводит i} напишет 4, 5 и 6 в этом.
10. times {| i | выводит i} напишет 0, 1, 2, 3, 4, 5,
   6, 7, 8, 9 в отдельных строках
```

4.10.3 Обмен информацией в языках программирования

Различные языки поддерживают различные механизмы передачи параметров. Например, использование вызова по значению и вызова по ссылке в языке «C», имитируется путем копирования адреса объектов данных. Язык «C++» использует вызов по значению, вызов по ссылке и «постоянный вызов по ссылке». Механизм передачи параметра «постоянный вызов по ссылке», разрешает доступ только для чтения и не позволяет писать в фактическом параметре. Доступ к динамическому объекту, хранящемуся в множестве, всегда передается с помощью вызова по значению. Это позволяет совместно использовать объекты. Java

использует вызов по значению. Поскольку доступ к объектам множества осуществляется с помощью указателей, вызов по значению позволяет совместно использовать объекты путем копирования указателей.

ADA использует вызов по значению (режим ввода), вызов по результату (режим вывода), вызов по значению результата (режим ввода-вывода) и вызов по ссылке (доступ). Объекты сопоставляются в зависимости от положения, и доступ к фактическим параметрам в вызывающей программе может быть защищен от модификации с помощью зарезервированного слова “protected” (защищено) перед фактическим параметром. ADA также позволяет использовать поименованные параметры формального параметра формы → фактический параметр для обеспечивания гибкой привязки фактических и формальных параметров; поименованные параметры не должны иметь соответствующие позиции.

Язык С использует вызов по значению. Однако язык С может передать адрес структуры данных, что соответствует вызову по ссылке. Языки C++, C#, F#, Modula, Modula-2, Modula-3, Pascal, Fortran 90 более поздней версии, PHP и Python используют два механизма передачи параметра: вызов по значению и вызов по ссылке; причем вызов по ссылке помечен зарезервированным словом, чтобы отделить его от вызова по значению. Все ссылки объектов множества передаются с помощью вызова по значению, а любые ссылки на объекты, хранящиеся в стеке управления, передаются с помощью вызова по ссылке. Кроме того, более мелкие структуры данных и единичные информационные объекты, хранящихся в стеке элемента управления, копируются с помощью вызова по значению. Ruby передает параметры путем вызова по значению. Каждый элемент данных в Ruby представляет собой объект, хранящийся в множестве; а каждая переменная является ссылкой на соответствующий объект. Присвоение значения переменной является эквивалентом создания нового объекта и указывает на этот новый объект.

Присвоение значения одной переменной другой переменной эквивалентно копированию адреса объекта. Вызов по значению в Ruby копирует адрес объекта, так что формальный параметр стартует с доступа к объекту. Следующая программа в интерактивной Ruby разъясняет этот принцип:

```
def entityId(Object);
        puts(“passed parameter: #{Object.object_id}\n”);
end
        y = 4.5;
```

```
puts("actual address: #{y.object_id}\n");
 entityId(y)

 def идентификаторобъекта(Объект);
      выводит («переданный параметр: #{Объект.объект_
      идентификатор}\n»);
 конец
          y = 4,5;
 выводит («фактический
 адрес: #{y.объект_
 идентификатор}\n»);
 идентификаторобъекта(y)
```

Вышеуказанный код даст одинаковый адрес в обоих случаях: идентификатор y.object_id дает адрес *y* перед передачей параметра, а идентификатор Object.object_id дает адрес объекта, на который указывает формальный параметр *Объект*.

Язык программирования Emerald (изумруд) поддерживает параметр передачи вызова по объекту ссылки, так же, как и вызов по значению для объекта обмена, поскольку переменные Emerald хранят указатели на объекты множества. Emerald поддерживает дистрибутивное вычисление, а передача ссылки объекта на удаленный процессор замедляет доступ к памяти. Таким образом, Emerald также использует вызов по визиту и вызов путем перемещения, чтобы создать локальные копии объектов, которые могут быть обработаны вызываемыми процедурами на удаленных процессорах.

Scala поддерживает как вызов по имени для передачи сложного выражения для функционального программирования, так и вызов по значению для передачи ссылки объектов. Функции являются объектами первого класса и могут быть переданы в качестве параметров. Программы можно разработать как с использованием функционального стиля, так и объектно-ориентированного стиля.

Haskell использует вызов по необходимости, вместо вызова по имени для обработки сложных выражений во время передачи параметров. Преимуществом вызова по необходимости является то, что сложные выражения вычисляются один раз и помещаются в кэш, и при следующем появлении используется кэшированное значение.

В языках ADA, C++, Clojure, Common Lisp, Fortran 90 и выше,

Python, Ruby и F# допускаются в параметре значения по умолчанию. Это означает, что если соответствующий фактический параметр отсутствует в конкретном вызове подпрограммы, то формальный параметр примет значение по умолчанию в теле подпрограммы.

4.11 Заключение

В этой главе мы обсуждали различные абстракции программы. Абстракции программы включают абстракции данных, управление абстракциями, инкапсуляцию данных и управления абстракцией, программы в качестве данных, мета программы, механизмы обмена информацией и взаимодействия между языками.

Абстракциями данных являются отдельные элементы, составные объекты с несколькими атрибутами, совокупность информационных объектов, расширяемый во время выполнения набор элементов данных, сеть объектов данных, а также переходные объекты с глобальной областью действия и постоянные объекты. Единичные элементы данных моделируются с помощью объявления такого базового типа, как целое число, плавающее (или реальный), символ, строка или элемент пронумерованного набора. Составной атрибут содержит несколько различных полей, которые могут быть разными типами и абстрагируется как *именованный кортеж*. Языки программирования используют *именованные кортежи* с *именованным полями* для лучшего доступа к кортежу и полю. Совокупность информационных объектов можно абстрагировать как набор, упорядоченный набор, мультимножество, упорядоченное мультимножество или мультимножество (ключ, значение) пар. Ключ является уникальным для каждого информационного объекта в совокупности (ключ, значение) пар. Все эти коллекции совокупности ассоциируются с именами для удобства доступа и обновления во время выполнения программы. Различные структуры данных могут использоваться для реализации совокупности информационных объектов. Его можно реализовать с помощью связных списков, индексируемых массивов, векторов и хэш-таблиц. Расширяемые совокупности могут быть реализованы с помощью связных списков, векторов — индексируемых, а также расширяемых — деревья и хэш-таблицы.

Абстракции управления являются операторами присваивания, включая множественное присваивание, последовательность команд, блок команд, условные операторы, итерационные конструкции, включая индексируемые переменные, итераторы, рекурсии и подпрограммы в виде функций и процедур. Блоки являются менее идентифицируемыми

структурами объявления и последовательности команд, таким образом, чтобы область объявления ограничивалась блоком. Блоки обеспечивают естественные границы для регулирования видимости данных и кода. Регулирование видимости важно для обеспечения гибкости такого же идентификатора имени в пределах блоков и модулей, и для избежания конфликтов при именовании.

Модули представляют собой еще один уровень абстракций, которые обеспечивают естественную границу для встроенных подпрограмм и абстракции данных. Разница между блоком и модулем заключается в том, что блоки содержатся в одной подпрограмме, а модули могут содержать несколько подпрограмм. Модули могут быть отдельно скомпилированы и помещены в архив для использования в будущем. Подпрограммы и абстракции данных из других модулей можно использовать с помощью механизма экспорта-импорта. Объект в модуле становится видимым для других модулей только после того, как он экспортируется из исходного модуля, и может использоваться в модуле только после импорта объекта с помощью объявления явного импорта. Класс представляет собой пассивный шаблон, содержащий абстракции данных и подпрограмм, которые работают на абстракциях данных; объекты — экземпляры классов, являющихся активными компонентами и имеющими состояние во время выполнения.

Обмен информацией между подпрограммами может происходить с использованием (1) глобальных переменных, (2) нелокальных переменных, (3) переменных класса в объектно-ориентированных языках, (4) общей области памяти в низкоуровневой реализации и ранних реализациях FORTRAN, и (5) передачи параметра в языках со структурным блоком. Поскольку переменная является носителем значения, то все атрибуты, которые могут извлечь r-значение переменных, можно использовать в передаче параметров. В целом, существует три атрибута: имя, ячейка памяти и само по себе r-значение. Исходя из этого, мы обсудили пять основных типов механизмов передачи параметров: вызов по имени, вызов по ссылке, вызов по значению, вызов по результату значения и вызов по результату.

Вызов по значению оценивает выражение в фактическом параметре и копирует значение в формальный параметр и обрабатывает формальный параметр как локальную переменную. Связь происходит в одностороннем порядке, и вычисляемые результаты в вызываемых подпрограммах не передаются обратно. Вызов по ссылке передает адрес ячейки памяти фактического параметра формальному параметру. В случае с

составными объектами данных или совокупностями объектов данных, базовый адрес структуры данных передается формальному параметру. Структура данных остается в среде вызывающей процедуры и доступ к ней осуществляется с помощью базовый-адрес + смещение отдельного элемента (или подполе в случае составных объектов). Объектно-ориентированные языки используют множество для хранения объектов. Ссылка на сложные объекты и динамические объекты, хранящиеся в множестве, дается с помощью связующей ссылки. Используя *вызов по значению*, можно скопировать связующие ссылки объектов. Мы также кратко обсудили два варианта вызова по ссылке: вызов по ссылке только для чтения и вызов по обмену. В вызове по ссылке только для чтения фактические параметры можно прочитать с помощью ссылки, хранящейся в формальном параметре, но не обновленной. Важно избежать любого случайного деструктивного обновления фактических параметров, что может привести к некорректному поведению программы. Вызов по обмену представляет собой комбинацию из первого вызова по ссылке, а затем последующего вызова процедуры в цепочке вызовов, используя вызов по значению для обмена ссылкой с другими вызываемыми подпрограммами.

Мы обсуждали непредсказуемое поведение во время выполнения при вызове по имени, из-за вычисления в ходе рабочего цикла индексированной переменной, которая может отобразить идентификатор другой ячейке памяти. Вызов по необходимости является вариантом вызова по имени, когда подвыражение вычисляется один раз, а значение кэшируется для последующего вхождения того же выражения. Вместо того, чтобы оценивать последующие подвыражения, кэшированное значение используется для повышения эффективности выполнения.

Мы обсуждали побочный эффект, изменяющий хранилище, доступное и изменяемое вызываемой программой, но с более продолжительным сроком службы по сравнению с вызываемыми программами. Результатом побочного эффекта является то, что эффект модификации сохраняется даже после завершения вызываемой подпрограммы. Если эффект непреднамеренный, то такие основные принципы программирования, как коммутативность выражений, может быть нарушена, вызывая некорректное поведение программы во время выполнения. Эффект наложения в сочетании с цепными ссылками также может вызвать непредсказуемое поведение программ.

Объекты первого уровня могут быть построены в виде данных во время выполнения, а затем превращаться в программу. Метапрограммы

воспринимают другую программу, как данные в некотором абстрактном домене для получения свойств программы или интерпретации поведения программы.

Обработка исключений делает программы более надежными и дает возможность постепенного выхода из исключений. Взаимодействие может быть обработано интерфейсным типом данных и абстракциями данных вызывающих и вызываемых программ. Это можно сделать с помощью определяющих языков конкретного интерфейса, либо с использованием общего языка спецификаций и метаданных, который определяет детали преобразования между типами данных двух различных языков программирования.

И, наконец, мы рассмотрели некоторые примеры абстракций данных и управления в различных языках, таких как ADA, C, C++, C#, Java, Modula-3, Ruby, и Scala. Другие примеры из языков с конкретной парадигмой приводятся в следующих главах, посвященным конкретным парадигмам: примеры программирования из Lisp, Scheme, и Haskell в Главе 9 о функциональной парадигме программирования; примеры программирования и механизм обмена информацией из Prolog в Главе 10 о логической парадигме программирования; подробные примеры, показывающие объектно-ориентированные функции в языках C#, C#,, Java, Ruby, Scala, и Modula-3 в Главе 11 о парадигме объектно-ориентированного программирования; примеры программирования из языков сценариев, таких как Python, Perl, и Lua в Главе 14 о языках сценариев.

4.12 ОЦЕНКА

4.12.1 Концепции и Определения

Фактический параметр; совокупность; совмещение имен; утверждение; доска объявлений; блок; вызов по копии; вызов по перемещению; вызов по имени; вызов по необходимости; вызов по ссылке; вызов по результату; вызов по обмену; вызов по значению; вызов по значению результата; вызов по посещению; цепное распределение; класс; совокупность объектов данных; общий язык промежуточного программного обеспечения; общий тип; составной объект данных; условный оператор; абстракция управления; абстракция данных; объявление; деструктивное обновление; обработка исключений; расширяемый объект данных, объект первого класса; формальный параметр; глобальная переменная; защита; защищенные команды; неизменяемый объект; импорт–экспорт; обмен информацией; сокрытие

информации; наследование; итерация; итератор; карта; метаданные; метапрограммирование; модуль; множественное назначение; изменяемый объект; мутатор; взаимная рекурсия; именованный кортеж; недетерминированное программирование; нелокальная переменная; объект; класс объекта; переопределение; передача параметров; сохранность данных; постусловие; предварительное условие, закрытый метод; защищенный метод; открытый метод; вызов по ссылке только для чтения; рекурсия; рефлективный язык; рефлексивность; реификация; саморекурсия; последовательность; последовательное назначение; затенение; побочный эффект; одиночный объект данных; один вход; один выход; файл символов; переключение; кортеж; система типов; если; видимость; слабое предварительное условие; когда.

4.12.2 Решение задач

1. Напишите быстродействующую программу на языках программирования Modula-3, C3, и Java, и сравните различные абстракции, используемые в языках программирования.

2. Напишите программу в Modula-3 и ADA 2012 для выполнения сортировки слиянием в модуле под названием “my_sort,”, а затем используйте его в основной программе для чтения и записи последовательности элементов данных.

3. Гостиница абстрагируется как двухмерный массив hotel[5, 120], где первое измерение показывает этажи, а второе измерение показывает номера в пределах этажа. Каждый объект данных занимает b байт. Дайте уравнение для вычисления смещения hotel[i, j], и примените его, чтобы получить смещение для расположения комнаты номер 18 на 4-м этаже. Предположим, что нумерация начинается с 1, а не 0.

4. Класс моделируется как массив из 30 составных объект данных под названием студенты. Каждый составной объект данных моделируется в виде кортежа формы (*имя, возраст, факультет, год*). Предположим, что имя моделируется как строка фиксированного размера из 32 символов, возраст моделируется как целое число в 4 байта, факультет моделируется как целое число в 4 байта, а год моделируется как целое число в 4 байта. Вычислите смещение различных полей в следующих объявлениях, таких как class[i].name.

5. Предположим, что язык поддерживает четыре типа передачи параметров: вызов по значению, вызов по ссылке, вызов по значению

результата и вызов по результату. Для следующей программы нужно показать трассировку после выполнения каждого утверждения в виде двумерной матрицы, где ряды показывают утверждение, а столбцы показывают различные переменные и формальные параметры. Предположим, что & является фактическим параметром, передающим ссылку, # является фактическим параметром, передающим параметр с помощью вызова по значению результата, а $ фактическим параметром, использующим передачу параметра с помощью вызова по результату.

```
Program main ( )
integer i, j, k, a[6];
{       i = 0; j = 0; k = 2; a[0] = 1
for (i = 1; i =< 5; i++) a[i] = a[i-1] * 2;
messy(a[3], &a[4], &j, &j, #a[3], $a[4]);
}
void messy( integer a, *b, *c, *d, e, f)
{       f = 2;
        a = *b + *c + e;
        *b = *d + f;
        *c = a + *b;
        *d = *c - *d;
        e = *b + *c + e;
        f = e + a;
}
```

```
Программа основные ()
целое число i, j, k, a[6];
{       i = 0; j = 0; k = 2; a[0] = 1
для (i = 1; i = < 5; i ++) a[i] =
a[i-1] * 2; грязный (a[3] & a[4] &
j & j, #a [3], $a[4]);
}

пустой грязный (целое число a, * b * c * d, e, f)
{       f = 2;
        a= * b + * c + e;
```

```
* b = * d + f;
* c = a + * b;
* d = * c - * d;
 e = *
 b +
 * c +
 e; f
 = e
 +a;
}
```

6. Напишите программу для выполнения пузырьковой сортировки в ADA, Modula-3, C#, и Java и сравните абстракции управления и передачу параметров, использованные в четырех программах.

4.12.3 Развернутый ответ

7. Каким образом реализуются расширяемые абстракции данных? Обсудите.

8. Что является преимуществом экспорта-импорта при разработке многократно используемой библиотеки? Объясните.

9. Какова разница между концепцией класса и концепцией модулей? Дайте четкое объяснение.

10. Каковы проблемы *вызова по имени*? Объясните с помощью простых примеров.

11. Что такое *вызов по необходимости*? Как это улучшает эффективность выполнения по сравнению с *вызовом по имени*? Объясните, используя простой понятный пример.

12. Что такое *вызов по обмену*? Какое сходство у него с *вызовом по значению*? Объясните, используя простую схему.

13. Каковы различия между *вызовом по значению* и *вызовом по ссылке*? Объясните с точки зрения механизма и эффективности доступа.

14. Каковы преимущества модулей и экспортно-импортного механизма перед вложенными процедурами, поддерживающими нелокальные переменные? Объясните.

15. При каких обстоятельствах *вызов по результату значения*

предпочтительней *вызова по ссылке* и почему?

16. Объясните и сравните различные механизмы передачи параметров для распределенных вычислений.

17. Каковы различные механизмы для обеспечения взаимодействия разных языков программирования? Объясните.

18. Каков механизм обработки исключения? Объясните.

19. Сравните экспортно-импортный механизм с нелокальными переменными и наследованием.

20. Какова разница между модулем, классом и объектами? Объясните.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Абельсон, Гарольд, Сассман, Джеральд Дж. и Сассман, Джули. *Структура и интерпретация компьютерных программ*, 2-е издание. MIT Пресс. 1996.

Американский Национальный Институт Стандартов. *Язык Программирования Ада*. CSA ISO/IEC 8652:201z. 2012.

Есть в наличии на сайте http://www.adaic.org/ada-resources/standards/ada05/

Биррелл, Эндрю Д. и Нельсон, Брюс Дж. «Реализация процедур удаленных вызовов.» *ACM Транзакции Компьютерных Систем*, 2(1). 1984. 39-59.

Блэк, Эндрю, Хатчинсон, Норман С., Джул, Эрик и Леви, Генри М. «Разработка языка программирования Изумруд». В *Работе над HOPL III Материалами Третьей Конференции ACM SIGPLAN по Истории Языков Программирования*. 2007. 11-1-11-51.

Колинбьерн, Хью. *Книга о языке Рубин*. No Starch Пресс. 2011.

Дейкстра, Эдсгер У. *Дисциплина Программирование*. Прентис Холл. 1976.

Хоар, Чарльз А. Р. «Аксиоматические Основы Компьютерного Программирования». *Передача Данных* в *ACM*, 12(10). 1969. 576-583.

Гудак, Пол, Хьюз, Джон и Джонс, Саймон Р. «История Haskell: Не затрачивая усилий на классы». В *Работе Третьей Конференции ACM SIGPLAN по Истории* Языков *Программирования*. 2007. 12-1-21-55.

Кеннеди, Кен, Колбэл, Чарльз и Зима, Ханс. «Взлет и падение языка

высокой производительности Фортран: Урок исторического объекта.» В *Работе Третьей Конференции ACM SIGPLAN по Истории Языков Программирования*. 2007. 7-1-7-22.

Кляйн, Питер. «Разработка программного обеспечения с Модула-3». *Технический Отчет 94-16*. Кафедра Компьютерных Наук III, Университет Технологий в Аахене. 1994.

Лисков, Барбара и Гуттаг, Джон. *Абстракции и Спецификация в Разработке Программ*. MIT Пресс. 1986.

Лисков, Барбара и Гуттаг, Джон. *Разработка Программы в Ява: Абстракция, Спецификация и Объектно-ориентированное Проектирование*. Эддисон-Уэсли. 2000.

Одерски, Мартин, Спун, Лекс и Веннерс, Билл. *Программирование в Скала: Полное Пошаговое Руководство*, 2-е издание. Artima Инкорпорейшн. 2011.

Страуструп, Бьерн. «Эволюция языка в и для реального мира: C++ «1991-2006». В *Работе Третьей Конференции ACM SIGPLAN по Истории Языков Программирования*. 2007. 4-1-4-59.

Ватт, Дэвид А. *Концепции и Парадигмы Языков Программирования*. Прентис Холл, 1990.

Вирт, Николас. «Модула-2 и Оберон». В *Работе Третьей Конференции ACM SIGPLAN по Истории Языков Программирования*. 2007. 3-1-3-10.

Глава 5

Модели реализации для императивных языков

БАЗОВЫЕ КОНЦЕПЦИИ

Абстракция и обмен информацией (Гл. 4); Абстрактные понятия в вычислениях (Раздел 2.4); Концепции структуры данных (Раздел 2.3); Программы и компоненты (Раздел 1.4); Рекурсия (Раздел 2.2.4); Машина фон Неймана (Раздел 2.1).

В этой главе и следующих главах мы изучаем абстрактную реализацию программ с помощью низкоуровневой абстрактной машины. Абстрактная машина дает абстрактное объяснение пошагового выполнения программных конструкций высокого уровня на промежуточном уровне. С помощью модели абстрактной реализации мы сможем понять и проанализировать динамическое поведение низкого уровня и управлять абстракциями, что необходимо для (1) написания более качественных и эффективных программ и (2) разработки генератора кода для компилятора.

Низкоуровневый преобразованный код может выполняться с использованием четырех типов схем распределения памяти: (1) *статическое распределение*; (2) на основе *распределения стека*; (3) на основе *распределения множества*; и (4) *гибридного распределения*, которое интегрирует схемы первых трех распределений для оптимального выделения памяти. В этой главе мы рассмотрим статическое распределения, распределение на основе стека и гибридное распределение. Сфокусируемся на гибридном распределении, которое (1) распределяет локальные динамические переменные в стеке, (2) распределяет рекурсивные и динамические структуры данных множества, и (3) использует *статическое распределение* для эффективного прямого доступа к памяти статических переменных.

Схемы статического распределения выделяют необходимую память в первую очередь, во время компиляции, и не поддерживают рост памяти во время выполнения. Преимущество этой схемы заключается в том, что каждый объект данных сопоставляется с уникальной ячейкой памяти и доступ к нему можно получить непосредственно через доступ к памяти. Однако, есть и недостатки. Статическое распределение не поддерживает (1) рекурсивные процедуры, поскольку рекурсивные процедуры могут вызывать сами себя неопределенное количество раз, что требует роста памяти во время выполнения; (2) расширяемые рекурсивные структуры данных, такие как связные списки, деревья и векторы, поскольку рекурсивные структуры данных могут быть продлены на неопределенный срок и нуждаются в увеличении памяти во время выполнения; и (3) динамическое создание объектов во время выполнения, поскольку требует сопоставления динамических объектов и ячеек памяти во время выполнения.

Все три ограничения возникают, поскольку рост памяти во время выполнения не поддерживается статическим распределением. Статическое распределение также неэкономно для памяти, потому что каждый объект сопоставляется с уникальный ячейкой памяти, а распределение не допускает перераспределения ячеек памяти после истечения срока службы распределяемого в настоящий момент объекта. Поэтому схема статического распределения не подходит для (1) программ со структурным блоком, поддерживающих несколько вызовов по функции и процедур в рамках подпрограммы, поскольку она не поддерживает повторное использование памяти; (2) парадигмы объектно-ориентированного программирования, создающей динамические объекты; (3) рекурсивных структур данных, которые могут быть расширены во время выполнения; и (4) языковых функций, такие как рекурсивные процедуры, требующие роста памяти во время выполнения.

Распределение на основе стека использует стек, называемый *локальный стек* или *стек управления*. Есть много преимуществ в использовании стека. Распределение на основе стека поддерживает рекурсивные процедуры, поскольку стек растет с вызовом процедуры и не фиксируется во время компиляции. Размер стека ограничен только операционной системой. Локальное хранилище вызываемой процедуры передается стеку управления при вызове подпрограммы во время выполнения, поскольку стек может вырасти. После завершения вызываемой подпрограммы, выталкивается локальное хранилище

под названием подпрограмма, освобождая места памяти, которые *используются повторно*, когда вызывается другая подпрограмма. Основным недостатком распределения на основе стека является отсутствие поддержки для динамически расширяемых структур данных, таких как связные списки, деревья, векторы, или динамически созданные объекты, которые переживают срок службы подпрограммы, которая их создала.

Распределение на основе множества использует общую область памяти под названием множество, которую видят все подпрограммы, с таким же сроком службы, как у программы, и которая может вместить все типы структур данных, включая расширяемые структуры данных и динамически созданные объекты. Распределение на основе множества использует указатель из регистров процессора или из элемента управления стеком для указания на первую ячейку данных в структуре данных, хранящейся во множестве , а затем использует внутренние указатели между ячейками данных для обхода других ячеек данных в той же логической структуре данных. Поскольку каждый указатель является адресом памяти, обход рекурсивных структур данных потребует множественных обращений к памяти. Пространство множества может быть расширено автоматически операционной системой во время выполнения, либо указаний программиста, встроенных внутри программы. После удаления распределенных структурных данных, памяти освобождается и маркируется для повторного цикла. После того, как распределяемая пространство памяти полностью заполнено, то есть два варианта: (1) расширить пространство памяти с помощью операционной системы или (2) утилизировать освобожденную память для перераспределения. Первый подход зависит от доступного пространства памяти в операционной системе. Второй подход производит повторную утилизацию памяти и называется *сбор мусора*.

Все три модели имеют свои преимущества и недостатки. Основным преимуществом статического выполнения является то, что доступ к расположенной в памяти переменной можно получить с помощью однократного доступа к памяти. Основными преимуществами распределения на основе стека являются (1) возможность обработки рекурсивных процедур и (2) повторное использование памяти из-за восстановления памяти после завершения вызываемой подпрограммы. Основными преимуществами распределения на основе множества являются (1) распределение расширяемых информационных объектов и динамически создаваемых объектов и (2) повторное использование

памяти, когда заканчивается срок службы динамически распределяемых и расширяемых структурных данных или динамических объектов. Современные языки программирования используют гибридную модель распределения, использующую преимущества всех трех подходов. Гибридная модель распределения использует статическое распределение для статических переменных и глобальных переменных, распределение на основе стека для динамических локальных переменных в блоке программы, и распределение на основе кучи для рекурсивных структур данных и динамически создаваемых объектов. В этой главе мы обсуждаем переход абстракций управления в низкоуровневый код, схему статического распределения и схему распределения на основе стека. Мы также обсудим, как стеки управления обрабатывают различные механизмы передачи параметров. В следующей главе мы обсудим управление динамической памятью с помощью множества.

5.1 АБСТРАКТНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ МАШИНЫ

Абстрактная реализация программы на машине фон Неймана состоит из пяти основных компонентов: *область данных*, *область кода*, *указатель инструкций для пошаговых инструкций в области кода*, *регистров для чернового вычисления* и *слова состояния программы* (PSW) — набор флажков, хранящихся в специальном регистре. Флажки в PSW заданы соответственно после каждой инструкции и являются важной частью состояния вычисления выполняемой программы. Область памяти, где хранится код, называется *область кода*, а область памяти, где хранятся данные, называется *область данных*. Для языков, которые не поддерживают объекты первого класса, область кода является *фиксированной* и *повторно используемой*, что означает, что каждый раз вызываемая подпрограмма начинается с самого начала в том же первоначальном состоянии.

Область данных является тройной линейной структурой (*множество*, стек *управления, статические области*). Множества и стек управления растут в противоположных направлениях, начиная с разных концов. Они растут навстречу друг другу для максимального использования имеющегося пространства памяти. *Стек управления* представляет собой последовательность кадров, отражающих вызывающий шаблон подпрограмм в порядке последним пришел - первым обслужен. Каждый кадр, также называется *записью активации*, представляющую собой индексируемую последовательность областей памяти, которые хранят

(1) информацию, необходимую для вычисления в текущей подпрограмме (2) информацию, возвращающую элемент управления обратно в вызывающую подпрограмму и (3) замороженное состояние вызывающей программы, когда вызываемая подпрограмма активизируется. Информация в кадре состоит из (1) областей памяти для локальных динамических переменных — переменных, которые не являются статическими переменными; (2) изображения регистров, которые будут обновлены в вызываемой подпрограмме; (3) различных указателей объектов данных в вызывающей программе, области кода и верхней части стека управления; (4) архивированной части программной среды и хранилища вызывающей программы, которые затеняются на текущий момент выполняющей подпрограммой; (5) простых динамические объекты, которые в подпрограмме создаются один раз и чей срок службы не превышает срок подпрограммы; (6) замороженное состояние вызывающей подпрограммы во время вызова текущей подпрограммы; и (7) параметров, передаваемых вызываемой подпрограмме.

Каждая активизация подпрограммы помещает соответствующий кадр на вершину стека управления. После завершения вызываемой подпрограммы, кадр удаляется для высвобождения памяти. Кадр индексируется для поддержки доступа к разным информационным объектам, хранящимся в кадре и для ссылок на другие кадры или множества. После завершения вызываемой подпрограммы отыскивается состояние вызывающей подпрограммы, и выполнение кода области в вызывающей программе возобновляется со следующей инструкции после вызова выполненной подпрограммы.

Область кода — последовательность блоков команд низкого уровня. Каждый блок команд представляет собой последовательность команд низкого уровня, представляющим собой подпрограмму. Управляющая логика использует ступенчатое движение к следующей команде или перескакивает с одной команды на другую с помощью команды перехода. Элемент управления перемещается из одной подпрограммы в другую с помощью операторов перехода. Управляющая логика в области кода осуществляется с помощью *указателя команд*, который похож на *счетчик команд* в командах на уровне компоновки. Существует одно основное различие между указателем команд, описанным здесь, и счетчиком команд, используемым в процессорах: указатель команд прирастает после успешного выполнения текущей команды, в то время, как счетчик команд в процессорах прирастает сразу же после вызова текущей команды. Это различие поясняется для нашего удобства, чтобы

объяснить концепции и уменьшить путаницу при обработке смещений. *Указатель следования* — команда перехода — принимает элемент управления к команде, указанной с заданным пользователем значением, вместо последовательности соседних команд. Команда перехода имеет важное значение для низкоуровневой трансляции абстракций управления, таких как традиционные конструкции и итерационные конструкции. В следующем разделе мы обсудим, как различные абстракции управления транслируются в низкоуровневых абстрактных командах с помощью диаграммы управляющей логики.

Низкоуровневые команды относятся к категории подмножества команд: (1) для загрузки значений, сохраненных в пространствах памяти, (2) для загрузки константы в регистр, (3) для хранения значений из регистров памяти, (4) для хранения значения константы в памяти, (5) арифметические и логические операции для оценки выражений, (6) операции сравнения арифметических и логических выражений, (7) условный и безусловный переход к команде (8) принудительная отправка данных на стек управления, и (9) выталкивание данных из стека управления.

Мы предполагаем три вида доступа к памяти: (1) *прямой доступ к памяти*, (2) *косвенный доступ к памяти* и (3) *доступ к памяти на основе смещения*. *Прямой доступ к памяти* осуществляет доступ к одной памяти для чтения или записи данных из (в) данную ячейку памяти, и является самым быстрым. Прямой доступ к памяти подходит для статических переменных. *Косвенный доступ к памяти* использует указатель для доступа к ячейке памяти, хранящей другую ячейку памяти или значение. Косвенный доступ к памяти может пройти через ряд указателей перед получением доступа к значению и нуждается более чем в одном обращении к памяти. Доступ к косвенной памяти обеспечивает независимость от фиксированных ячеек памяти. Однако это происходит медленнее из-за нескольких обращений к памяти. Много раз указатели для часто используемого косвенного доступа сохраняются в регистрах процессора для повышения эффективности. Доступ к памяти на основе смещения используется для доступа к информационным объектам в совокупности объектов данных или подполе внутри составного объекта данных. Адресация на основе смещения требует двух единиц информации: базовый адрес и выражение, которое показывает смещение информационного объекта или подполя. Базовый адрес — это адрес первой ячейки памяти в ОЗУ, где находится объект.

Флажки состояния в PSW необходимы для обработки результатов

оценки выражения. *Бит отрицания N* имеет значение «1», если сравнение дает отрицательное значение и «0», если сравнение дает положительное значение. *Нулевой бит* имеет значение «1» когда сравнение дает значение ноль и «0», если результат является значение не равное нулю. Оценка логического выражения устанавливает *флажок булево выражение* на «1», если результат *правда*, и 0, если результат имеет значение *ложь*. Операторам условного ветвления нужны эти флажки для того, чтобы сделать условные переходы к области команд. Некоторые операторы условного перехода являются *brlt* (ветвление на менее-чем), *brgt* (ветвление на более-чем), *breq* (ветвление на-равно-чему), *brne* (ветвление на не-равно-чему), *brle* (ветвление на менее-чем-или-равно-чему) и *brge* (ветвление на более-чем-или-равно-чему).

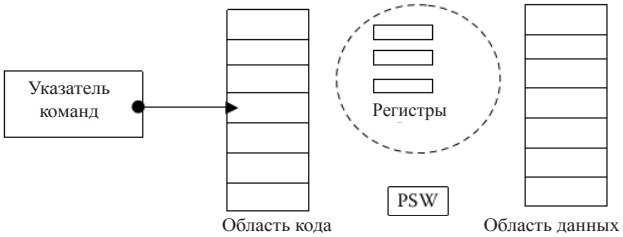


Рисунок 5.1 Схема абстрактной машины для реализации.

Рисунок 5.1 иллюстрирует низкоуровневую абстрактную машину для реализации языков программирования. После компиляции, хранилище переводится в область данных, а абстракции управления переводятся в область кода. Область кода и область данных моделируются в виде одномерных массивов. Ячейка памяти в области кода обозначается как *c[код-индекс],* где при вычислении выражения «индекс кода» выдается смещение с первой команды в области кода. Ячейка памяти в области данных обозначается как *d [данные-индекс],* где индексом данных является выражение, дающее при вычислении смещение с базового адреса области данных, видимых в данный момент.

В зависимости от парадигмы программирования, область данных и область кода можно дополнительно уточнить для выявления конкретной абстрактной машины. Большинству абстрактных машин требуется как минимум *стек управления*, *множество* и область непосредственно доступной памяти для глобальных и статических переменных. В схемах статического распределения базовый адрес после компиляции равен 0 и пропущен.

Однако в схеме распределения на основе стека, *кадры* размещаются в разных ячейках памяти стека, а доступ к элементам данных осуществляется с помощью (1) базового адреса кадра с использованием указателя под названием *указатель кадра* и (2) смещения в отношении кадра базового адреса.

Простейшая абстрактная машина для парадигмы функционального программирования называется *SECD машина*, которая имеет четыре стека: S, *стек для оценки выражения*; E, *стек для хранения программной среды*; C, *стек для хранения командной строки*; и D, *стек для хранения дампа* — последовательный программный вызов процедур в порядке последний пришел-первый обслужен.

Парадигма логического программирования также поддерживает *поиск с возвратом* — способ вернуться назад, отменить некоторые из вычислений и попробовать альтернативные пути для поиска решений. Осуществление поиска с возвратом требует дополнительного стека под названием *стек-след*, а также стандартного стека управления. Стек-след отслеживает количество контрольных точек для возврата назад и отмены действий с целью поиска альтернативных действий.

Объектно-ориентированные языки используют такую виртуальную машину на основе стека, как *виртуальная машина* Java, а также используют множество для хранения объектов. Императивные языки поддерживают реализацию на базе одного стека и одного множества.

В этой главе мы обсудим абстрактную машину фон Неймана для парадигмы императивного программирования. В соответствующих главах рассматриваются другие абстрактные машины. Например, машина SECD рассматривается в главе 9, а абстрактная машина Уоррен (WAM) — абстрактная машина для логического программирования — рассматривается в Главе 10.

Схематическая реализация объектно-ориентированных языков рассматривается в Главе 11.

5.2 ПЕРЕВОД АБСТРАКЦИЙ УПРАВЛЕНИЯ

Перевод абстракции управления можно сгруппировать следующим образом: (1) перевод вычисления выражений, (2) перевод оператора присваивания, (3) перевод условных конструкций, 4) перевод итерационных конструкций, и (5) перевод вызова подпрограммы. В этом разделе мы обсудим все другие абстракции управления, кроме вызова подпрограммы, который будет обсуждаться отдельно в Разделах 5.3 и 5.4.

5.2.1 Перевод выражений

Вычисление выражения осуществляется с помощью комбинации регистров процессора и ячеек памяти. Для снижения издержек доступа к памяти, промежуточные значения после вычисления подвыражения сохраняются в регистрах процессора. Общие подвыражения вычисляются один раз и сохраняются в регистрах для использования в будущем; после первого вычисления общих подвыражений, осуществляется распределение регистра в ходе анализа времени компиляции с использованием различных методов оптимизации. Например, вычисление выражения (X + Y + 5) + 2 * (X + Y) будет переводиться следующим образом:

```
load X, R1 % загрузка значения, хранящегося в ячейке памяти (X)
в регистре R1,
add Y, R1, R2, %, добавить значение, хранящееся в ячейке памяти
(Y) в регистре R2,
add #5, R2, R3 %, добавить константу 5 к R3 и сохранить в
регистре R1
multiply #2, R2, R4 % значение-в (регистр R4) = 2 * значение-в
(регистр R2)
add R3, R4, R4 %, добавить значения, хранящиеся в регистрах R3
и R4
```

Приведенный выше пример иллюстрирует прерывание вычисления выражений на последовательности абстрактной команды низкого уровня для загрузки значений переменных из соответствующих ячеек памяти и временного сохранения частичных результатов в регистрах. Обратите внимание, что общее подвыражение (X + Y) вычисляется только один раз и сохраняется в регистре R2 для повторного использования в будущем. В будущем, чтобы обсудить перевод других абстракций управления, мы будем избегать перевода выражения и обозначить его как *вычисление* (*выражения*).

5.2.2 Перевод оператора присваивания

Оператор присваивания <*переменная*> = <*выражение*>эквивалентен последовательности низкоуровневых абстрактных инструкций, которые сначала вычисляют выражение, а затем сохраняют полученное значение в ячейке памяти левосторонней переменной. Например, такой оператор присваивания, как X = Y + 5 эквивалентен следующему:

load Y, R0 % загрузка значения в ячейке Y в регистр R0
load # 5, R1 % загрузка константы 5 в регистр R1
add R0, R1, R0 % добавить значение R0 и R1, и сохранить результат в R0 store R0, X % сохранить значение-в(R0) в ячейке переменной X

Зная, что операция присваивания представляет собой сочетание операций загрузки, вычисления и это слово должно стоять в другом месте хранения, обозначим операцию присваивания простой командой *присвоить* (<*переменная*>,<*выражение*>).

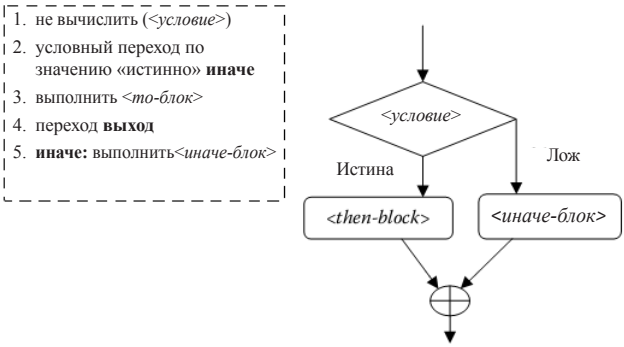


Рисунок 5.2 Схема перевода диаграммы управляющей логики в абстрактную машину.

5.2.3 Переводеконструкциивусловногоиоператора

Управляющая логика условного оператора состоит из трех шагов: (1) вычисление условия булева выражения; (2) на основе вычисления булева выражения, установите флажки состояния на уровне оборудования; и (3) на основе значения флажков состояния ветви, сделайте переходы для выполнения либо блока операторов в части «то» или блока операторов в части «иначе». На рисунке 5.2 показана диаграмма управляющей логики условного оператора и соответствующие схемы для низкоуровневого перевода.

Диаграмма управляющей логики является двумерной плоской фигурой. Её нужно преобразовать в область кода, который является одномерным. Для обеспечения взаимного исключения < *то - блок*> и

<*иначе-блок*> и чтобы избежать пошагового выполнения в <*иначе-*

блок>, требуется оператор перехода после выполнения < *то -блок*>.

После прохождения управляющей логикой через < *то -блок*>, оператор перехода выполняется для обхода <*иначе-блок*> и контроля расположения после <*иначе-блок*>. Элемент управления переходит к <*иначе-блок*>, если условия имеют значение ложь. Поскольку управление логикой движется по умолчанию в направлении вперед, переход необходимо только для перемещения к <*иначе-блок*>, так как элемент управления автоматически будет поступать в < *то -блок*>, если условия являются правдой. Для этого проверяется отрицание условия. Если отрицание условия является правдой, то элемент управления переходит к <*иначе-блок*>. В противном случае, элемент управления проходит через < *то -блок*> по умолчанию.

5.2.4 Переводоператоралварианта

Операторы варианта можно имитировать с помощью последовательности утверждений если-затем -< то -*блок*>, как показано ниже, чтобы выполнить одну из взаимоисключающих возможностей:

1. result = evaluate(<expression>);
2. if (result == value 1) then {<block 1 >; jump-to exit;}
3. if (result == value 2) then {<block 2 >; jump-to exit;}

…
If (result == value N) then {<block N >; jump-to exit;}
<default-block>;
<exit>:

1. результат = вычислить (<*выражение*>);
2. **если** (результат == значение$_1$) **затем** {<*блок$_1$*>; переход-к **выходу**;}
3. **если** (результат == значение$_2$) **затем** {<*блок$_2$*>; переход-к **выходу**;}

…

Если (результат == значение$_N$) **затем** {<*блок$_N$*>; переход-к **выходу**;}

<*по умолчанию-блок*>;

<выход>:

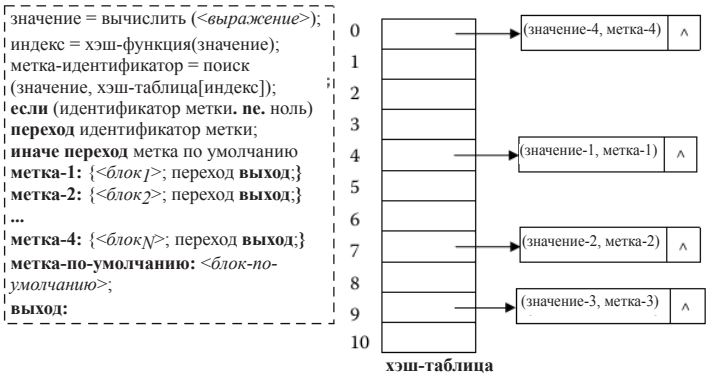


РИСУНОК 5.3 Перевод оператора варианта с помощью хэш-таблицы.

Кроме того, его можно реализовать с помощью хэш-таблицы, которая хранит массив мест, где элемент управления может переходить на основе вычисления <*выражение*>, как показано на Рисунке 5.3. После вычисления выражения, результат приводится в хэш-таблице. Хэш-таблица содержит утроенную форму (*ожидаемое-значение, метка-номер, указатель на следующий триплет*). Если ожидаемое значение совпадает с производным значением, то возвращается соответствующая метка; в противном случае возвращается значение «нуль», и элемент управления переходит к метке по умолчанию. После выполнения блока утверждения, соответствующего ожидаемому значению, элемент управления обходит другие блоки, перейдя на метку «выход.»

5.2.5 Перевод итерационныхдконструкций

Итерационные конструкции преобразуются с помощью перевода конструкции условного оператора. Если мы переводим цикл-пока, используя условный оператор, утверждения выглядят следующим образом:

1. Инициализировать переменные;
2. **цикл: если** (вычислить (**нет** <*выражение*>) = правда) {
3. переход-к **выходу;**
4. **иначе** {выполнение<*блок-пока*>
5. переход-к **цикл;** }

}

6. **выход:**

если мы переводим {if <expression> then {<while-block>; jump-to loop} с использованием предварительных знаний о переводе условного оператора, тогда перевод будет, как на Рисунке 5.4. Вычисляется первое *нет* (<*выражение*>). Если *нет* (<*выражение*>) имеет значение правда, то элемент управления выходит из цикла-пока. В противном случае, элемент управления выпадает из <*блок-пока*>, выполняет его и переходит обратно к началу цикла-пока.

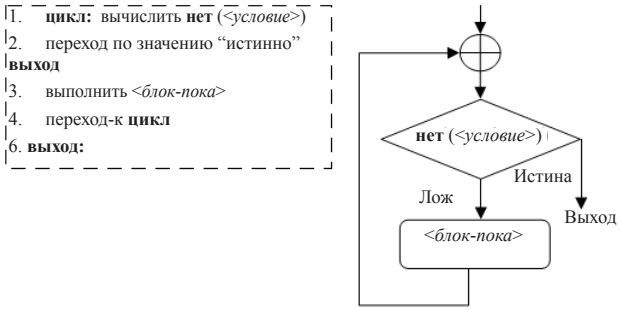


РИСУНОК 5.4 Схема перевода цикла-пока в низкоуровневые абстрактные инструкции.

5.2.5.1 Перевод цикла-для

Конструкция цикла-для имеет следующую форму:

for (i = <initial-expression>';'<final-expression>';'<step-expression>)
<for-loop-block>';',

```
для (i = <начальное-выражение>«;»<конечное-
выражение>';'<шаг-выражение>)
        <для-цикл-блок>';',
```

Цикл-для будет смоделирован с помощью пока-цикла, где окончательное условие является (индекс> вычислить (<*конечное-выражение*>)). Перевод цикла-для с помощью цикла-пока будет следующим:

1) assign(i, evaluate(<initial-expression>);
2) while (not evaluate(<final-expression>))
{3. execute <for-loop-block>;

4. assign(i, i + evaluate(<step-expression>));
}

1) **присвоить** (i, вычислить (<*начальное-выражение*>);
2) **пока** (**не** вычислять (<*конечное-выражение*>))
{3. **выполнить** <*для-цикл-блок*>;
4. **присвоить** (i, i + вычислить(<*шаг-выражение*>));
}

Используя знания об итерационных конструкциях, для-цикл переводится, как показано на Рисунке 5.5.

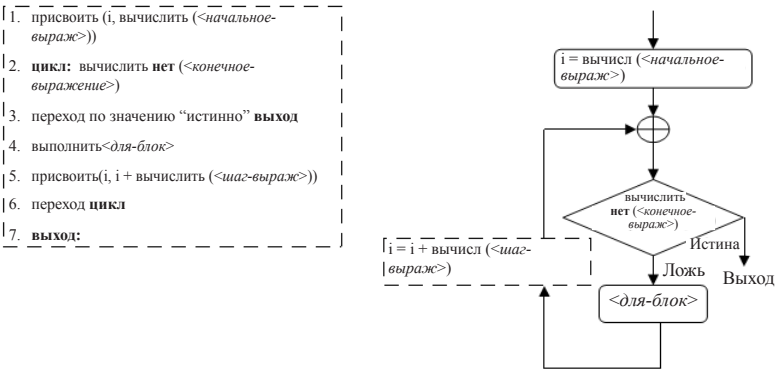


РИСУНОК 5.5 Схема для перевода для-цикла для низкоуровневой абстрактной машины.

Все на Рисунке 5.5 похоже на перевод цикла-пока, за исключением двух операторов: (1) команда присвоения для инициализации индексируемой переменной и (2) обновление индексируемой переменной невидимым для пользователя способом. Блок для приращения индекса был показан с использованием пунктирных линий, чтобы показать, что пользователь не имеет контроля над приращением индексируемой переменной в пределах блока операторов, за исключением указания выражения для размера шага в начале цикла-для.

Подобный перевод может быть сделан для цикла «пока» и оставлен в качестве упражнения для студентов. Итераторы очень похожи на перевод цикла-пока и цикла-для. Единственное отличие заключается в том, что итераторы проходят через совокупность элементов один за другим, а в начале цикла, они проверяют, достигли ли они конца совокупности.

5.3 СТАТИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ

Современные языки поддерживают рекурсивные процедуры, повторное использование памяти, рекурсивные структуры данных и динамические объекты данных. Таким образом, предпочтение отдается гибридной модели. Однако, некоторые концепции распределения статической памяти полезны в понимании реализации на основе стека и гибридного подхода. Возьмем пример Fortran-66, так как более поздние версии Fortran поддерживают распределение на основе стека.

Область данных представлена в виде массива, а также используется запись d [i] для обозначения прямого доступа к *i-той* ячейке памяти в области данных. Сначала выделяются ячейки общей памяти, за ними ячейки памяти для хранения обратного адреса следующей инструкции после вызова инструкцией текущей подпрограммы. Обратный адрес хранится в области данных, поскольку его значение меняется каждый раз, когда подпрограмма вызывается из другого места. Все локальные переменные, которые не используются совместно, распределяются локально в активационной записи подпрограммы.

Различные программные единицы можно компилировать отдельно. Скомпилированные коды *связаны* друг с другом в порядке, указанном пользователем. Области данных и области кода связываются в том же порядке, указанном пользователем, а доступная информация в таблицах символов различных скомпилированных кодов объединяется. Вызывающая программа не будет иметь никакой информации об области данных и области кода вызываемой подпрограммы, если они были скомпилированы отдельно; информация станет доступна только после слияния таблицы символов. Сведения, полученные из таблиц символов, включают (1) информацию об импортируемых и экспортируемых объектах данных, (2) информацию о том, куда можно сделать переход при вызове подпрограммы и (3) информацию о том, куда возвращаться после завершения текущей подпрограммы. Во время компиляции каждая подпрограмма предполагает, что набор недостающей информации будет предоставляться ей во время связывания через таблицу символов соответствующих подразделений программы.

Во время связывания области данных различных скомпилированных кодов объединяются таким образом, что общие объявления блока соединяются вместе, за ними следует обратный адрес, а затем локальные переменные. Давайте предположим, что областью данных скомпилированного кода является (C, R_I (I > 1), L_I (I > 1)), где C обозначает общий блок, R_I обозначает обратный адрес ячейки памяти

области скомпилированного кода *i*, а L_I обозначает набор локальных переменных в *i-той области данных*, которая после связывания области данных будет выглядеть как (C, L_1, R_2, L_2,..., R_I, L_I,... R_N, L_N) в том же порядке, как указано пользователем. Все общие блоки с одинаковым именем в различных скомпилированных кодах объединяются вместе, как один общий блок, и размещаются в начале совместной области данных. Адрес памяти обратных адресов и локальные объявления смещаются во время процесса связывания в результате слияния областей данных. Связывание — это двухэтапный процесс: (1) на первом этапе все области данных объединяются, вычисляются переходы к подпрограммам и возвращения из них, вычисляются смещения областей данных различных программных модулей, а (2) на втором этапе, соответствующие смещения включаются в область кода.

Вызов подпрограммы включает (1) хранение в памяти следующей исполняемой команды вызова подпрограммы в обратном адресе, (2) обмен информацией о параметрах и (3) переход к первой инструкции вызываемой подпрограммы.

Пример 5.1

Рисунок 5.6 иллюстрирует реализацию с использованием схемы статического распределения. Предположим, что основную программу и подпрограмму FIND_MAX скомпилировали отдельно. Как основная программа, так и подпрограмма разделяют общий блок, состоящий из переменной MAX и фиксированный массив *M* размером 4. В основной программе не используемыми совместно переменными являются *I* и *J*. Программа *ОСНОВНАЯ* считывает данные в общем массиве *M*, а затем вызывает подпрограмму FIND_MAX. Подпрограмма FIND_MAX находит максимум массива *M* и использует общий блок для совместного использования значения переменной MAX. Низкоуровневый код использует прямой доступ к памяти, как показывает d [<*m*>], и использует метод смещения для доступа к элементу массива.

В основной программе после компиляции, адресом диапазона переменных от *M* [1] до *M* [4] является соответственно диапазон от d [0] до d [3], а переменной MAX соответствует ячейка памяти d [4], и локальные переменные *I* и *J* в основной программе распределяются в ячейках памяти d [5] и d [6]. В подпрограмме FIND_MAX переменным *M* [1] … *M* [4] соответствуют адреса d[0] ... d[3]; переменная MAX

отображается в ячейке памяти d[4]; а обратный адрес в подпрограмме занимает ячейку памяти d[5], а затем следует локальная переменная *I*, которая занимает ячейку памяти d[6].

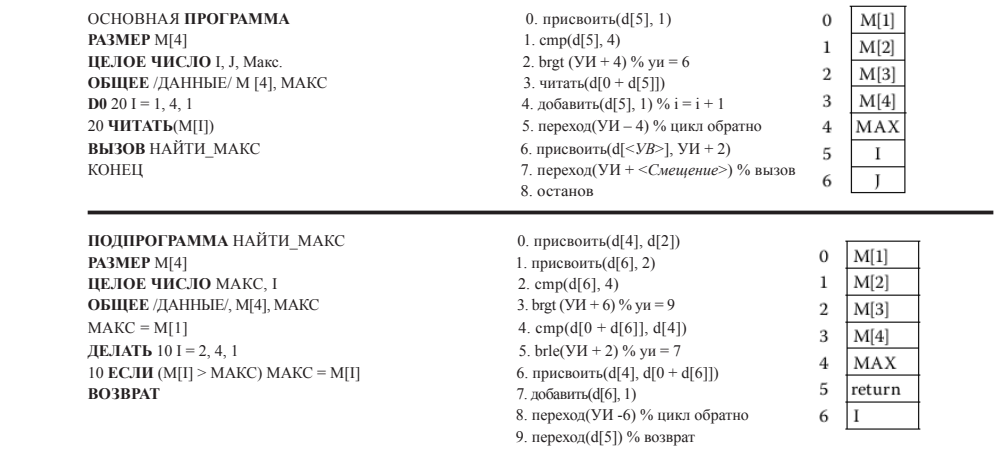


РИСУНОК 5.6 Область кода и область данных для статического выполнения после компиляции.

Эта программа показывает низкоуровневый переход для вызова подпрограммы FIND_MAX и две основные абстракции управления: *цикл «пока»* (вариация цикла-для) и *условных* операторов. Перевод цикла «пока» в основной программе начинается с расположения области кода c[0] и заканчивается на c[5]. Оператор c[0] присваивает значение 1 ячейке памяти d[5], в которой находится переменная *I*. Оператор c[1] сравнивает значение d[5] с константой 4. Оператор c[2] ответвляется к IP + 4, если значение d[5] больше 4. Обратите внимание на использование «IP + 4» для выведения управления из цикла. Метод добавления смещения указателя инструкций, делает код нижнего уровня независимым от областей памяти компьютера, в которые загружается программа для выполнения. Инструкция c[3] использует метод смещения для доступа к ячейке памяти *M*[*I*]: 0 является базовым адресом *M*[0], а d[5] хранит значение переменной *I*. Инструкция c[4] увеличивает значение d[5] до 1, и инструкция c[5] возвращает себе управление обратно на c[1], где начинается следующий цикл итерации. Инструкции c[6] и c[7] связаны с вызовом подпрограммы FIND_MAX. Инструкция c[6] хранит «IP +

2» — адрес следующей инструкции высокого уровня после вызова подпрограммы в неизвестный на данный момент обратный адрес подпрограммы *НАЙТИ_МАКС*. Поскольку информация неизвестна, создается новый символ и вводится в таблицу символов. Во время связывания адрес памяти возвращаемого указателя отыскивается в таблице символов. Инструкция c[7] использует смещение перехода от текущей инструкции к первой исполняемой инструкции в вызываемой подпрограмме FIND_MAX. Однако, смещение также неизвестно до момента связывания и вычисляется во время компоновки.

Скомпилированный код нижнего уровня для цикла DO в подпрограмме FIND_MAX, занимает ячейкм памяти c[1]...c[8], а перевод условного оператора if-then-else, вложенного в цикл DO, занимает ячейкм c[4]...c[6]. Инструкция на c[1] эквивалентна инициализации индексируемой переменной *I* к 2. Инструкция c[2] сравнивает значение индексируемой переменной *I* с верхней границей 4. Инструкция c[3] выводит управление из цикла DO, если индексируемая переменная больше 4. Инструкция c[4] сравнивает значение переменной нижнего индекса *M* [*I*] со значением переменной MAX и переходит к c[7], если *M*[*I*] меньше или равно значению переменной MAX. В противном случае, c[6] присваивает значение *M*[*I*] переменной MAX. Инструкция c[7] увеличивает значение d[6] (локальная переменная *I*) на 1. Инструкция c[8] вычитает 6 из указателя инструкции *ip* для перехода к инструкции c[2] и начала следующей итерации цикла. Последняя инструкция c[9] выбирает адрес, хранящийся в ячейке памяти обратного адреса, и возвращает элемент управления к следующей инструкции в вызывающей процедуре. Поскольку подпрограмма может быть вызвана несколько раз из различных частей области кода, то адрес, хранящийся ячейке памяти обратного адреса будет каждый раз другим.

Как показано на Рисунке 5.7, после связывания общий блок переменных с M[1] до M[4] и переменная MAX объединяются и помещаются ячейки памяти от d[0] до d[4]. Ячейки памяти обратного адреса и локальной переменной *I* в процедуре FIND_MAX размещаются после последней ячейки памяти, выделенной для основной программы в области данных. Используемый обратный адрес отображается на d[7], а локальная переменная I_{sub} отображается на d[8]. Ячейки памяти без совместного использования подпрограммы FIND_MAX смещаются по размеру (запись активации основной

программы) — размер — (общий блок в Подпрограмме FIND_MAX) = 7 − 5 = 2. Это смещение добавляется к ячейкам памяти в часть скомпилированного кода подпрограммы

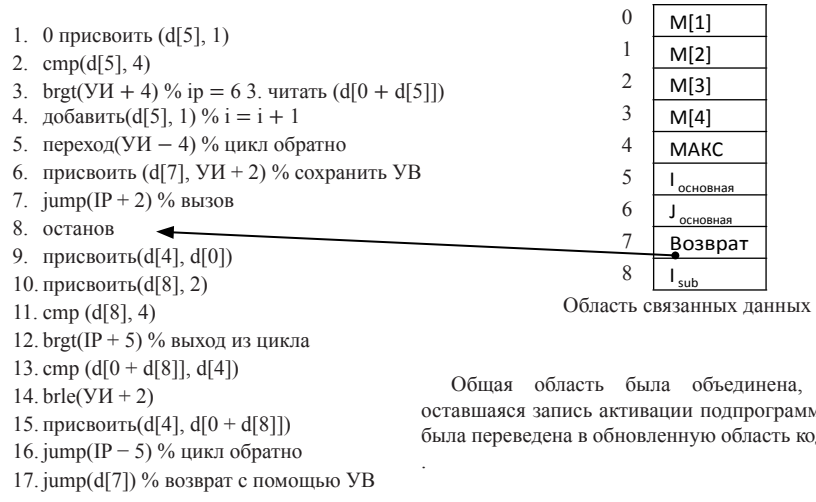


РИСУНОК 5.7 Связывание областей кода и областей данных.

FIND_MAX во время связывания для создания связанного кода. Символ <*R*> в C[6] заменяется на 7-й — адрес памяти указателя возврата — а символ <*S*> заменяется значением смещения 2, чтобы перейти к первой инструкции подпрограммы.

5.4 ГИБРИДНОЕ РАСПРЕДЕЛЕНИЕ

Языки с блочной структурой и языки, которые поддерживают рекурсивные процедуры или динамические объекты, используют гибридное распределение, в которое входит стек управления, множество и статическое распределение, как показано на Рисунке 5.8. В интегрированной модели область данных является триплетом формы (*статическая область, стек* управления, множество). Распределение статической памяти фиксируется в момент компиляции и используется для статических и глобальных переменных. Стек управления и множество изменяются динамически с ходом выполнения программы. Множество и стек управления растут в противоположных направлениях, поскольку,

в зависимости от входных данных, для различного вызова одной и той же программы могут понадобиться разные размеры множества и стека управления. Во многих случаях, размер стека больше, а в других случаях, множеству требуется больше памяти для расширения. При сохранении роста множества и стека управления с двух концов по направлению друг к другу, концы стека управления и множества определяются по объему использования, а область памяти используется до максимальной.

Первый указатель на объект множества находится либо в кадре процедуры, либо в регистре. Распределение множества подробно обсуждается в следующей главе. Тем не менее, в этой главе множество было упомянуто для обсуждения параметра перехода, когда объекты размещены в множестве.

Каждый кадр вызываемой подпрограммы содержит много информации: (1) ячейки памяти для входящих формальных параметров (2) сохраненное состояние для вызова подпрограммы, различные указатели для доступа к стеку управления (включая кадр программы запроса и нелокальные переменные) и для восстановления вычислительного состояния вызывающих подпрограмм, ячейки памяти для локальных переменных, (5) ячейки памяти для чернового вычисления, и (6) ячейки памяти исходящих фактических параметров.

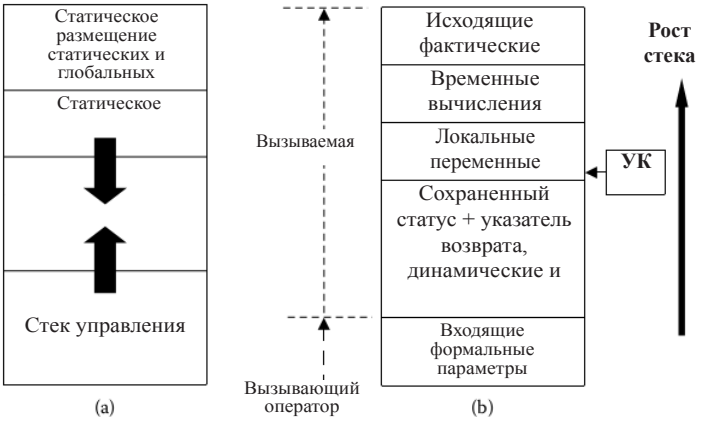


РИСУНОК 5.8 Область данных для блочно-структурированного языка. (a) - Общая область данных, (b) - кадр.

Когда вызывается подпрограмма, сохраняется часть режима вычисления, которая будет изменяться в вызываемой подпрограмме.

Сохраненный режим вычисления включает (1) значения для регистров, которые изменяются в вызываемой подпрограмме, (2) PSW, и (3) различные указатели, которые используются для доступа к области кода и области данных вызывающей подпрограммы. Различные указатели создаются для доступа к: (1) коду и области данных вызываемой подпрограммы (2) ячейкам памяти фактических параметров базовых адресов в кадре вызывающей программы, если параметры передаются вызовом по ссылке, и (3) кадры процедур, при которых внедряется текущая процедура. Для каждой локальной переменной выделяется ячейка памяти. Переменная может быть сопоставлена с объектами данных разного размера. Например, символ может занимать только один байт; а целое число занимает 4 байта на 32-разрядном компьютере. На верхней части кадра есть некоторые ячейки памяти, которые используются для хранения результатов частичных вычислений при вычислении выражений, и, наконец, он содержит фактические параметры, которые должны быть переданы вызываемой подпрограмме.

Вычисляются выражения в фактическом параметре, а результирующее значение сохраняется в новой временной ячейке памяти в области исходящих параметров. Динамические локальные переменные, которые имеют неперекрывающиеся области, могут быть отображены на тех же ячейках памяти для более эффективного использования памяти. После того, как подпрограмма завершается, отбрасывается кадр вызываемой подпрограммы, устанавливаются указатели для доступа к вызывающей программе, а верхняя часть указателя стека сбрасывается для указания на ячейку памяти сразу после кадра вызывающей программы. Таким образом, вся память, используемая вызываемой программой, освобождается для повторного использования в будущем.

5.4.1 Роль различных указателей

Существует пять основных указателей, необходимых для обработки вызываемой подпрограммы. После сохранения вычислительного состояния вызывающей программы и ссылок доступа к кадру вызывающей подпрограммы элемент управления переходит к первой инструкции вызываемой программы. Во время выполнения подпрограммы в распределении на основе стека, необходимо пять указателей: *Указатель кадра*, обозначается как УК и позволяет получить доступ к: (1) область данных выполняющейся в данный момент подпрограммы, (2) местоположение входящих параметров, и (3) сохраненные сведения о состоянии вызывающей подпрограммы. Указатель кадра указывает

на первую ячейку памяти после сохраненного состояния; а ко всем другим ячейкам памяти доступ осуществляется путем добавления смещения базового адреса, хранящегося в указателе кадра. *Указатель вершины стека*, обозначается как УВС, и указывает на следующую свободную ячейку памяти в стеке управления. *Динамическая* ссылка обозначается как ДС и хранит указатель кадра вызывающей программы и используется для восстановления указателя кадра вызывающей программы. *Указатель возврата*, обозначается как УВ, и хранит адрес следующей команды вызывающей программы после команды, которая вызывает подпрограмму. *Статическая* ссылка обозначается как СС и хранит адрес для доступа к кадру подпрограммы, который вложена выполняемая в данный момент подпрограмма. Она используется для доступа к нелокальным переменным. Многие из указателей, такие как УК и УВС, хранятся в регистрах для более быстрого доступа.

5.4.2 Вызов подпрограмм

При вызове подпрограммы происходят изменения указателя в следующем порядке:

1. Исходящая область параметров устанавливается на основе типа передачи параметра и количества параметров. В это время УВС продолжает получать обновления из-за выталкивания значения фактических параметров.

2. Ячейка памяти следующей команды вызова подпрограммы сохраняется в указателе возврата вызываемой подпрограммы.

3. Динамическая ссылка устанавливается равной указателю кадра вызова подпрограммы.

4. Статическая ссылка устанавливает указатель на кадр подпрограммы, под которым вложена вызываемая подпрограмма.

5. Текущее значение указателя вершины стека копируется в указатель кадра (УК), чтобы настроить УК и указать кадр вызываемой подпрограммы.

6. Указатель команды устанавливается на ячейку памяти первой команды в вызываемой подпрограмме.

7. В вызываемой подпрограмме УВС увеличивается на размер

местной среды для распределения на стеке локальных динамических переменных.

Важен порядок, в котором устанавливаются эти указатели. Например, если мы изменим УК перед копированием старого значения УК в динамическую ссылку, то доступ к кадру вызывающей подпрограммы будет потерян. Аналогичным образом, если мы установили сначала УВС, то нельзя устанавливать новый УК.

Порядок, в котором расположены указатели, зависит от реализации. Здесь мы предполагаем, что у УВ есть смещение – 3, у динамической ссылки смещение – 2, а у статической ссылки (если она присутствует) смещение – 1. В случае, если статическая ссылка отсутствует, поскольку язык не поддерживает гнездовые процедуры, то у УВ будет смещение – 2, и у ДС смещение – 1. Таким образом, доступ к указателю возврата производится через d[УК-3] (или D[УК-2], если SL отсутствует); доступ к динамической связи осуществляется через d[УК-2] (или d[УК-1], если СС отсутствует); и к статической ссылке, если она есть, доступ осуществляется через d[УК-1]. Мы также предполагаем, что УК и УВС хранятся в регистре процессора для эффективного доступа к памяти.

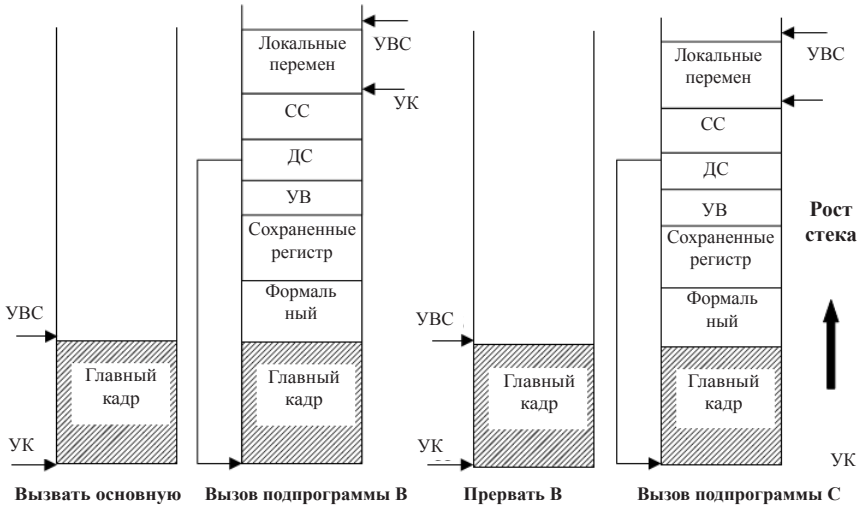


Рисунок 5.9 Движение кадра во время процедуры вызовов.

В понимании большинства концепций мы игнорируем сохраненные регистры и PSW для нашего удобства, без потери какой-либо информации.

Пример 5.2

Давайте рассмотрим сценарий. Программа *основная* вызывает подпрограмму A. Подпрограмма A вызывает подпрограмму B. Подпрограмма B выполняет и завершает работу. Затем основная программа вызывает подпрограмму C. На рисунке 5.9a показан снимок стека управления, когда управление находится в пределах основной программы. Рисунок 5.9b показывает стек управления, когда элемент управления находится в подпрограмме B. Рисунок 5.9c показывает стек управления, когда подпрограмма B завершена, а элемент управления возвращается в основную программу. На рисунке 5.9d показан стек управления, когда программа C вызывается основной программой. Обратите внимание на повторное использование памяти, которая первоначально была занята кадром подпрограммы B.

5.4.3 Создание областей данных и кода

Хранилище вызываемой подпрограммы включает (1) хранилище, созданное глобальными переменными, (2) хранилище, созданное нелокальными переменными, (3) хранилище, созданное локальными переменными (3) хранилище, создаваемое путем передачи параметров, и (4) хранилище, созданное локальными переменными. Глобальные переменные и статические переменные распределяются статически в начале области данных и доступ к памяти осуществляется без каких-либо указателей. Доступ к локальным переменным осуществляется путем добавления смещения указателя кадра в стеке управления. Например, d[УК + 3] предоставляет доступ к ячейке памяти, которая находится на три ячейки дальше адреса, указанного УК.

Статическая ссылка (СС) сохраняет базовый адрес кадра следующего внешнего уровня подпрограммы, а нелокальная переменная может быть объявлена как N ($N \geq 1$) уровнем выше вложенной процедуры. Для доступа к нелокальным переменным используются два метода: (1) цепь статических ссылок и (2) отображение регистров.

Схема цепи статических связей подразумевает, что статическая связь хранится в ячейке памяти, которая имеет в нашем случае фиксированное смещение: d[УК-1]. Этот метод использует последовательные переходы, используя статические связи, хранящиеся в кадре. Если нелокальная переменная объявлена как «*m*» в другом гнездовом уровне, то доступ

к данным нижнего уровня может быть записана как *d(m, смещение)*, а алгоритм доступа к нелокальной переменной будет следующим:

```
while (m > 0) {
                m = m - 1; frame-base = d[SL];
                SL = frame-base − 1;}
                value = d[frame-base + offset]
```

```
пока (m>0) {
          m = m - 1; кадр-
          основа = d[SL];
          SL = кадр-основа -
          1;}
          значение = d[кадр-основа + смещение]
```

Цепь статических связей является простым методом и страдает от издержек, появляющихся от нескольких обращений к памяти, если гнездовой уровень слишком велик. Второй метод использует *регистры* отображения. *Регистры* отображения являются регистрами процессора или кэш памяти, содержащей прямой адрес нелокальных переменных. Регистр отображения заполняется при запуске вызванной процедуры, а доступ к нелокальным переменным эффективно осуществляется без прохождения цепи указателей.

К переменным в кадре вызывающей процедуры обращаются, когда параметры передаются в качестве ссылки. В вызове по ссылке формальный параметр является указателем на первую ячейку памяти объекта данных, на который дается ссылка. Если абстракция данных является составным объектом данных, то доступ к различным атрибутам производится путем добавления смещения атрибута на первую ячейку памяти объекта данных. Аналогично, если абстракция представляет собой совокупность объектов данных, то смещение вычисляется путем умножения индекса на размер отдельного элемента данных, а ячейка памяти i-го объекта данных вычисляется как d[ссылка-связь] + i * размер (отдельного элемента данных), а значение внутри ячейки памяти доступно как d[d[ссылка-связь] + i * размер (отдельного элемента данных)].

Вызываемые подпрограммы осуществляет доступ к ячейке памяти в локальных кадрах, используя только свой указатель кадра и смещение. Когда кадр вызываемой подпрограммы выталкивается в стек управления при запуске подпрограммы, то занимаемая кадром память вызывается

для повторного использования после завершения подпрограммы. Последовательность команд вызова подпрограммы выглядит следующим образом:

1. Вычислить выражения в фактических параметрах и скопировать значения в область исходящих параметров.

2. Сохранить PSW вызова программы и регистры в область сохраненного состояния кадра вызываемой программы. Сохраняются только те регистры, которые изменяются во время сохранения вызываемой подпрограммы.

3. Обновить указатель возврата (УВ) для указания следующей команды в вызывающей программе после возврата управления из вызываемой подпрограммы.

4. Скопировать УК в динамическую ссылку вызываемой подпрограммы.

5. Обновить статическую ссылку, если нелокальные переменные поддерживаются языком.

6. Обновить указатель кадра путем копирования УВС в качестве нового значения УК.

7. Обновить указатель возврата, чтобы он указывал на следующую команду вызова программы.

8. Сделать переход к первой команде вызываемой программы.

9. В области кода вызываемой подпрограммы, первая команда выделяет локальную среду для вызываемой подпрограммы путем добавления размера кадра вызываемой программы в указатель-вершины-стека (УВС).

Возврат из вызываемой подпрограммы выполняет следующие операции для восстановления среды и хранилища вызова подпрограмм:

1. Сохраненные регистры и PSW копируются обратно в соответствующие регистры, чтобы восстановить вычислительное состояние вызова подпрограммы.

2. УВС вызова подпрограммы восстанавливается для возврата памяти, возврата вызываемой подпрограммой. УВС вызова подпрограммы восстанавливается путем вычитания размера входящих параметров, сохраненного состояния и различных указателей из указателя кадра вызываемой подпрограммы.

3. Динамическая ссылка ДС копируется в УК для восстановления кадра вызова подпрограммы.

4. Указатель возврата УВ копируется в УИ, чтобы вернуть управление вызову программы.

После возвращения из вызываемой подпрограммы и перед выполнением следующей команды, результаты формальных параметров копируются из ячеек памяти исходящих параметров в ячейки памяти фактических параметров.

В остальных наших обсуждениях, мы не будем затрагивать сохранение PSW и регистров, не теряя никакую соответствующую информацию, относящуюся к реализации вызова подпрограмм.

Пример 5.3

Рисунок 5.10 иллюстрирует концепцию использования блока структурированной программы. Указатель инструкций (УИ), указатель кадра (УК) и указатель вершины стека (УВС), хранятся в регистрах процессора для обеспечения эффективного доступа. Программа была создана специально, чтобы показать разные абстракции управления, абстракции данных и статические распределения. Основная программа считывает четыре элемента данных в m[4] и вызывает подпрограмму find_max. Снимок показывает сценарий, когда элемент управления находится внутри подпрограммы find_max.

Команды brgt (ответвление, если более-чем), переход (безусловная ветвь), brle (ответвление, если меньше или равно), и cmp (сравнить) являются интуитивными. Команда add обновляет первый аргумент, добавив значение, хранящееся во втором аргументе. Например, команда “add(TOS, 1)” эквивалентна «“TOS = TOS + 1”. Многие абстрактные команды при переводе на команды на уровне сборки отображаются отображаются в последовательности команд. Например, команда “assign(d[FP], IP + 2)” соответствует паре операторов “add(IP, 2, R1), assign(d[FP], R1)”. Аналогичным образом, оператор чтения сошлется

на вызов считывания данных операционной системой. Команда «push» помещает аргумент в стек управления и увеличит УВС на единицу.

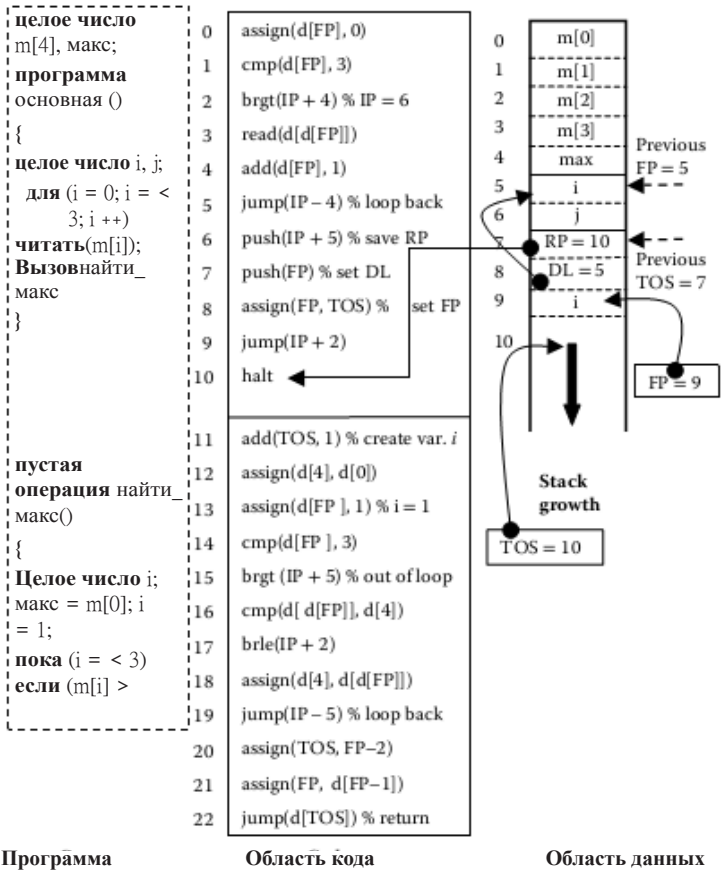


РИСУНОК 5.10 Генерация кода и данных в реализации на основе стека.

Область данных имеет три раздела: статическая область для глобальных переменных, кадр основной программы, и кадр вызываемой подпрограммы find_max. Глобальные переменные состоят из массива m[4] и переменной Max. Глобальные переменные распределяются статически: *m*[0] ↦d[0], m[1] ↦ d[1], m[2] ↦ d[2], m[3] ↦ d[3], и Max ↦ d[4]. Кадр основной программы находится между d[5] и d[6], а кадр

вызываемой процедуры swap (*обмен)* между d[7] и d[9]. В кадре основной программы не содержится указатель возврата и динамическую ссылку. Ячейки d[7] и d[8] сохраняют указатель возврата и динамическую ссылку, чтобы вернуть управление основной программе.

В кадре *основной* программы содержатся ячейки памяти для двух локальных переменных: *i* ↦

d[5] и *j* ↦ d[6]. В кадре подпрограммы find_max содержится указатель возврата (УВ)

↦ d[7], динамическая ссылка (ДС) ↦ d[8] и локальная переменная *i* ↦ d[9]. УК указывает на ячейку d[9], УВС указывает на ячейку d[10], а ДС указывает на базу кадра основной программы; то есть d[5]. Указатель возврата (УВ) указывает на ячейку c[11] в области кода, которая соответствует команде “halt”.

Область кода использует указатель кадра (УК) для доступа к ячейкам памяти в рамках выполняемой в данный момент процедуры. Команды от c[0] до c[5] соответствуют выполнению цикла «for» (для). Команда c[0] присваивает значение 0 ячейке памяти, на которую указывает УК и является низкоуровневым переводом оператора “i = 0”; команда c[1] сравнивает значение d[УК] с константой 3. Выполнение этого оператора задает соответственно PSW, и команда c[2], используя флаги PSW для выхода из цикла while (пока) переходит на оператор c[6]. Команда c[3] считывает значение переменной нижнего индекса *m*[*i*] с помощью двух обращений к памяти: первое читает значения d[УК] (значение *i*), а затем читает индексированную память. Чтение активизирует вызов на системном уровне. Команда c[5] вычитает 4 из указателя команд для перехода обратно к c[1] — начальной точке цикла «for» (для).

Команды области кода c[6] и c[9] используются для настройки различных указателей и перехода к первой команде вызываемой подпрограммы. Команда, расположенная в c[6], устанавливает указатель возврата, помещая «УИ + 5» на вершину стека (на d[7]), и УВС увеличивается на 1. Команда, расположенная в c[7], настраивает динамическую ссылку , помещая значение УК на вершину стека (на d[8]), и УВС увеличивается на 1. Значения регистров, при их изменении, также помещаются в стек. Команда, расположенная на c[8], устанавливает УК для доступа к кадру подпрограммы find_max путем присваивания УК для УВС. Смещение управляется ячейками памяти, необходимыми для сохранения вычислительного состояния вызывающей подпрограммы; оно изменится, если будут сохранены регистры, PSW и статическая ссылка. Команда c[9] передает управление первой команде подпрограммы

find_max путем поиска смещения в таблице символов: смещение в этом случае равно 2. Последняя команда c[10] соответствует команде высокого уровня “halt”.

В подпрограмме find_max команда c[11] устанавливает УВС, добавляя количество ячеек памяти, выделенных для локальных переменных: в нашем случае с размером 1. Команда c[12] соответствует команде высокого уровня “max = m[0]”: max ↦ d[4] и m[0] ↦ d[0]. Команда c[13] соответствует команде высокого уровня «i=1»: i ↦ d[УК]. Команды между ячейками c[14] и c[19] соответствуют циклу-while (пока), а команды между ячейками c[20] и c[22] соответствуют возврату из подпрограммы find_max. Команда в ячейке c[14] сравнивает переменную *i* с константой 3. Команда в ячейке c[15] выходит из цикла-while , если значение переменной *i* больше 3. Команда в ячейке c[16] соответствует команде высокого уровня *m* [*i*] > max: *i* ↦ d[УК], max↦ d[4], и d[d[УК]] ↦ *m* [*i*]. Команда в ячейке c[17] соответствует незаполненной части условного оператора else (иначе) и передает управление команде в ячейке c[19]. Команда в ячейке c[18] является эквивалентом команды высокого уровня max=*m*[*i*]. Команда в ячейке c[19] вычитает 5 из «УИ» для перехода обратно к началу цикла- while (пока).

Возврат из подпрограммы сбрасывает указатели в следующем порядке:

1. Команда в ячейке c[20] копирует значение УК-2 в УВС для освобождения памяти, используемой в кадре подпрограммы find_ max. Значение 2 является суммой размера информации сохраненного состояния, в нашем случае 2: указатель возврата и динамическая ссылка.

2. Команда, расположенная в c[21], восстанавливает указатель кадра основной программы путем копирования динамической ссылки (d[УК-1]) в УК.

3. Команда, расположенная в ячейке c[22], передает управление обратно к основной программе путем копирования адреса, хранящегося в УВ (d[УВС]) в УИ. После изменения УВС в c[20], УВС указывает на ту же ячейку памяти, как и указатель возврата вызываемой подпрограммы.

Для доступа к переменным массива требуется два обращения к памяти: один, чтобы получить значение индексируемой переменной, а второй - для получения значения ячейки памяти, соответствующей элементу массива. Этот дополнительный доступ к памяти сокращается

компиляторами, путем сохранения в регистрах индексируемых переменных. Использование кэш-памяти также улучшает издержки доступа к памяти.

5.5 РЕАЛИЗАЦИЯ ПЕРЕДАЧИ ПАРАМЕТРОВ

В зависимости от типа механизмов передачи параметров, отличается часть реализации соответствующего обмена информацией. Например, для вызова по значению требуется копирование вычисляемого значения выражения в ячейку памяти формального параметра. Вызов по ссылке должен сохранить базовый адрес фактического параметра в ячейке памяти формального параметра.

Пространство памяти для фактических параметров в вызывающей подпрограмме называется *пространством исходящего параметра*, а ячейке памяти для формальных параметров в кадре вызываемой подпрограммы называется *пространство входящего параметра*. Пространство исходящего параметра принадлежит к кадру вызывающей подпрограммы, а пространство входящего параметра относится к кадру вызываемой подпрограммы. Однако, *пространство исходящих* и *входящих параметров* накладываются одно на другое так, что вызывающая и вызываемая подпрограммы могут получить к нему доступ. Вызываемая подпрограмма сохраняет вычислительные состояние в информации сохраненного состояния сразу после пространства входящего параметра. УК указывает сразу после информации сохраненного состояния. Смещение формального параметра является отрицательным относительно УК вызываемой подпрограммы. Однако, смещение оставшихся локальных переменных является положительным относительно УК.

5.5.1 Реализация вызова по значению

Во время вызова по значению, выражения значения фактического параметра оцениваются и хранятся в пространстве исходящих параметров. Вызываемая подпрограмма сохраняет PSW, измененные в вызываемой подпрограмме регистры, а также различные указатели. Доступ к входящим параметрам осуществляется через d[УК – смещение] для единичных информационных объектов и d[УК- база-адрес-смещение + d[УК + индекс-смещение]] для доступа к элементу массива, где смещение-база-адрес является смещением первого элемента в массиве, а смещение-индекс является смещением индексируемой переменной

i для доступа к элементу массива *a*[*i*]. Доступ к составным объектам множества также осуществляется путем копирования ссылки на объект в ячейки памяти формального параметра.

Пример 5.4

На Рисунке 5.11 показана абстрактная реализация вызова по значению. Основная программа копирует значения двух переменных *i* и *j*, в соответствующие формальные параметры *x* и *y*. Подпрограмма swap меняет местами значения формальных параметров *x* и *y*. Результаты вычислений в подпрограмме swap не передаются обратно.

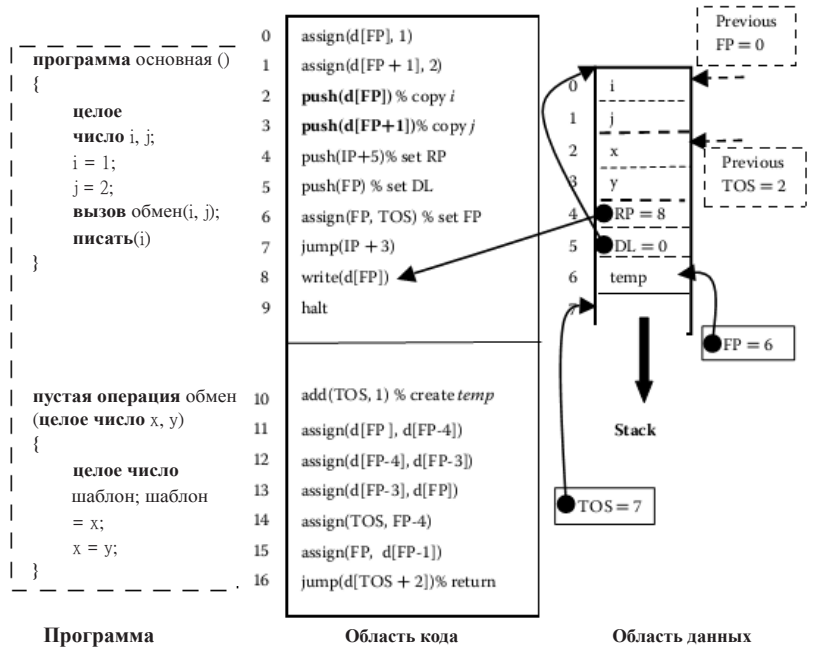


Рисунок 5.11 Схема реализации вызова по значению.

Кадр основной программы состоит из ячеек от d[0] до d[3], а кадр вызываемой подпрограммы состоит из d[2] до d[6]. Перекрывающиеся ячейки d[2] и d[3] соответствуют формальным параметрам *x* и *y*. Ячейки d[2] и d[3] рассматриваются основной программой, как пространство исходящего параметра, а вызываемой процедурой swap – как

пространство входящего параметра. В ячейках d[4] и d[5] сохраняются указатели УВ (указатель возврата) и ДС (динамическая ссылка): в ячейке указателя возврата d[4] хранится адрес следующей команды write(i) (запись); а в ячейке d[5] хранится динамическая ссылка. Переменная temp сопоставляется с ячейкой d[6].

До вызывающей процедуры swap (обмен), элемент управления находится в основной программе, УК указывает на d[0], а УВС указывает на d[2]. УК подпрограммы swap указывает на ячейку d[6]. D[УК-1] является ячейкой динамической ссылки, и в d[УК-2] хранится обратный адрес следующей команды, которая должна быть выполнена в основной программе. Статическая ссылка отсутствует.

Передача параметров использует УК для копирования значений пространства исходящего параметра. Команды, расположенные в ячейках локализации кода c[2] и c[3] копируют значение фактических параметров в d[УК] и d[УК+1] в ячейки исходящих параметров, используя команды push(d[FP]) и push(d[FP + 1]). Оставшиеся в основной программе команды используются для установки различных указателей и перехода к подпрограмме swap, они уже обсуждались в Примере 5.3.

Записи d[УК-4] и d[УК-3] используются для доступа к значению формальных параметров *x* и y, соответственно. Смещения *4* и *3* вычисляются путем известного размера информации сохраненного состояния и размера входных параметров. Команда на c[11] копирует значение переменной *x* в память ячейки переменной temp . Команда в ячейке кода c[12] копирует значение переменной *y* в *x*, а команда в ячейке кода c[13] копирует значение переменной temp в ячейку памяти *y*. Остальные три команды, расположенные между ячейками кода c[14] и c[16], являются командами передачи управления обратно вызывающей программе, мы их обсуждали в Примере 5.3. В команде c[16], ячейку указателя возврата можно определить во время компиляции, зная размер ячеек исходящего параметра. В нашем случае это размер 2: фактические параметры *i* и *j*.

5.5.2 Реализация вызова по ссылке

Во время вызова по ссылке, вызывающая подпрограмма хранит базовый адрес объекта данных в пространстве исходящего параметра, который становится пространством входящего формального параметра для вызываемой подпрограммы. Доступ к объектам ссылки осуществляется через d[d[УК – смещение параметра] + смещение индекса], где смещение параметра — это смещение базового адреса входящего формального

параметра вызываемой подпрограммы, а *смещение индекса* является смещением локального индекса переменной в кадре вызываемой подпрограммы. Второй элемент можно заменить соответствующим механизмом адресации, если переменная индекса отличается от локальной переменной. Ничего особенного не нужно делать во время возвращения из вызываемой подпрограммы, так как вызов по ссылке дает фактической переменной возможность непрерывной мутации, поэтому не нужно явным образом передавать обратно какой-либо результат.

Пример 5.5

На Рисунке 5.12 в ячейках памяти формальных указателей *x* и *y*, хранится адрес ячеек памяти переменных *i* и *j*, соответственно. Так как *x* и *y* хранят адреса фактических параметров *i* и *j*, то передача параметра в ячейке c[2] вставляет адрес памяти УК вместо r-значения d[УК], а ячейка c[3] вставляет адрес памяти УК + 1 вместо r-значения d[УК + 1].

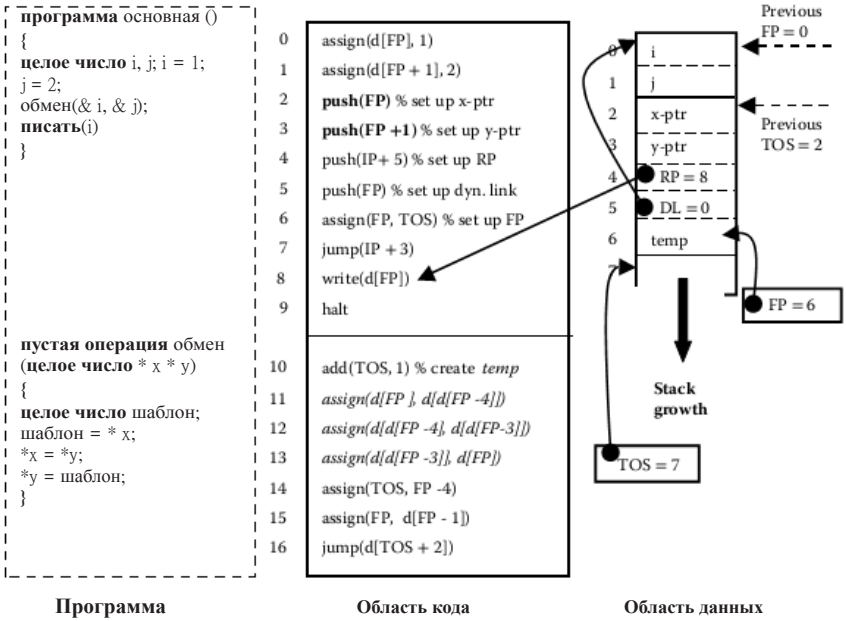


РИСУНОК 5.12 Схема реализации вызова по ссылке.

Команды в ячейках c[11], c[12] и c[13], которые используют формальные параметры *x* и *y* для непрямого доступа к фактическому параметру, имеют два обращения к памяти, обозначаемые как d[d [УК-4]], для доступа к значению фактического параметра переменной *i*, и d[d[УК-3]] для доступа к значению фактического параметра переменной *j*. Обратите внимание, что d[УК-4] предоставляет адрес памяти фактического параметра *i*, а d[УК-3] предоставляет адрес памяти фактического параметра *j*. Остальные команды, обсуждаемые ранее, похожи на вызов по значению.

5.5.3 Реализация вызова по значению результата

Во время вызова по значению результата, вызывающая подпрограмма оценивает значение выражений для фактических параметров и сохраняет полученное значение в соответствующих ячейках памяти в пространстве исходящих параметров. После завершения вызываемой подпрограммы, результат, в случае необходимости, копируется обратно в вызывающую подпрограмму из пространства исходящего параметра в ячейки соответствующих переменных. Поскольку пространство исходящего параметра вызывающей программы и пространства, входящего параметров вызываемой подпрограммы накладываются, то результат автоматически передается обратно вызывающей подпрограмме.

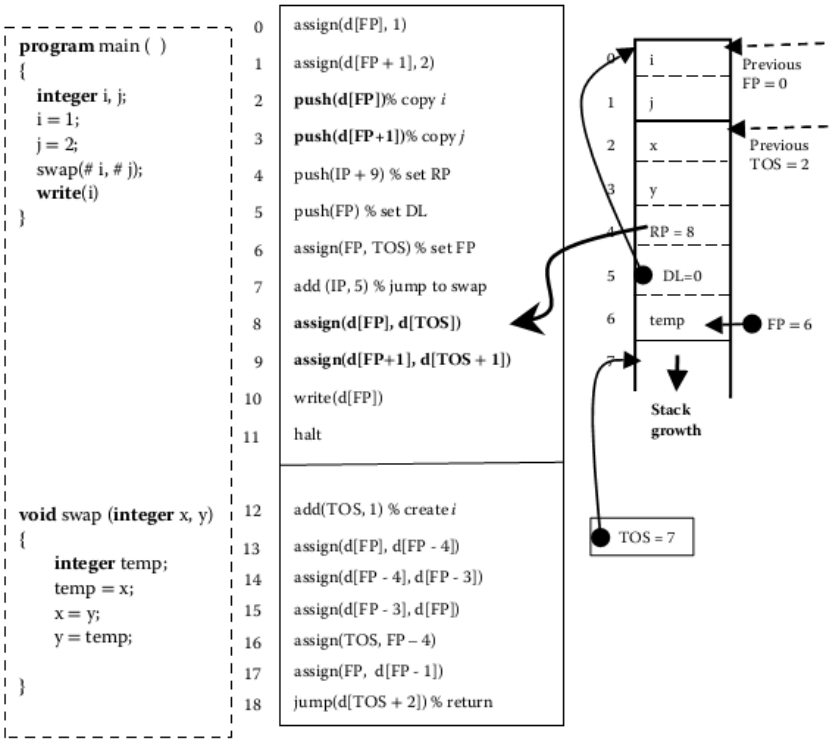


РИСУНОК 5.13 Схема реализации вызова по значению результата.

Как вызов по значению, так и формальные параметры рассматриваются, как локальные переменные, и доступны через d[УК – смещение] для единичных информационных объектов, а d[УК- смещение базового адреса + d[УК + смещение индекса]] для доступа к элементу массива, где смещением базового адреса является смещение первого элемента данных в массиве, а смещением индекса является смещение индексируемой переменной *i* для доступа к элементу массива *a*[*i*].

Пример 5.6

На рисунке 5.13 показана та же программа с переходом параметра вызов по значению результата. Фактические параметры вызова по значению результата были помечены с помощью символа «#». Область

данных является такой же, что в вызове по значению. Есть еще две дополнительные команды в области кода: команда в ячейках c[8] и c[9] копирует результат обратно из пространства исходящего параметра в фактическую ячейку памяти переменных *x* и *y*, соответственно. После того, как элемент управления возвращается в область кода, результат формального параметра копируется обратно в ячейки фактической памяти перед выполнением следующей команды высокого уровня. Обратите внимание, что пространство исходящего параметра используется многократно, несколькими вызовами различных подпрограмм, и не может использоваться в качестве изображения конкретных переменных. Все другие команды уже обсуждались в Разделе 5.5.1 и похожи на вызов по значению.

5.6 НИЗКОУРОВНЕВОЕ ПОВЕДЕНИЕ РЕКУРСИВНЫХ ПРОЦЕДУР

Рекурсивные процедуры вызывают активацию новой записи для каждого рекурсивного вызова. Обратный адрес в кадрах каждого экземпляра указывает на то же расположение области кода, а статическая ссылка кадров каждого экземпляра рекурсивной процедуры указывает на кадр той же процедуры, согласно которой рекурсивная процедура является вложенной. Однако, динамические ссылки экземпляра $m_{\text{го}}$ ($m > 2$) указывают на базовый адрес кадра $(m-1)_{\text{й}}$ экземпляра, а первый экземпляр рекурсивной процедуры указывает на кадр процедуры, вызвавшей рекурсивную процедуру.

Пример 5.7

Разница между статическими и динамическими ссылками в вызове рекурсивных процедур объясняется на Рисунке 5.14, показывая ссылками процедур *факториальной* подпрограммы - программы, вызываемой основной программой.

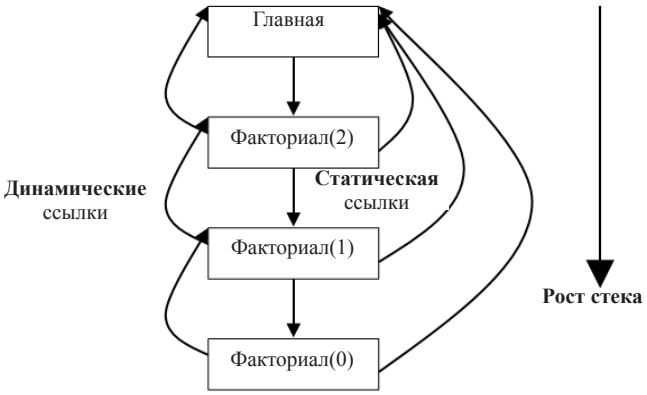


РИСУНОК 5.14 Динамические и статические ссылки в рекурсивных процедурах.

Основная программа вызывает *факториал*(2). *Факториал*(2) вызывает *факториал*(1) и приостанавливает себя, ожидая значение из *факториала*(1). Динамическая ссылка *факториал*(1) указывает на базу кадра запуска *факториал*(2). Аналогичным образом, *факториал*(1) вызывает *факториал*(0). Динамическая ссылка кадра процедуры вызова для *факториал*(0) указывает на основание кадра *факториала*(1). Однако, статические ссылки в кадрах всех вызовов *факториал*(2), *факториал*(1) и *факториал*(0), указывают на основание кадра Основной программы, под которой *факториальная* программа является вложенной.

5.7 РЕАЛИЗАЦИЯ ОБРАБОТЧИКА ИСКЛЮЧЕНИЙ

Обработчики исключений выполняются с помощью операторов перехода. После того, как программа возвращается от выполнения оператора с обработчиками исключений или из вызываемой процедуры, устанавливается флажок «успешного завершения». Если инструкция выполнена успешно без каких-либо исключений, то элемент управления переходит к инструкции после обработчиков исключений, используя оператора ветви. В противном случае, элемент управления проходит через обработчики исключений. Он проверяет условия в каждом обработчике исключений и выбирает ячейку команды из соответствующего местоположения в рамках процедуры, если условие имеет значение истина. Если объявлено исключение, то у записи активации есть вход в кадр процедуры, которая хранит месторасположение выполняемой программы. После выполнения обработчика исключений, элемент

управления возвращается обратно, чтобы проверить следующее условие исключения.
Если обработчик исключений выполняется успешно, то элемент управления остается в выполняющейся на данный момент подпрограмме. В противном случае, если ни один из обработчиков исключений не совпадает с условием исключения, то команда ветвления принимает элемент управления для последовательности возврата, и элемент управления возвращается к вызывающей подпрограмме, которая проверяет флажок исключения и повторяет процесс.
Другим подходом к реализации является сохранение отдельного стека исключений под названием *стек исключений*. Адрес первой команды обработчика исключений и самый верхний кадр активации вставляется в стек выполнения. По завершении успешного выполнения обработчика исключений, все кадры выше кадра успешного обработчика исключений удаляются из элемента управления стеком.

5.8 ЗАКЛЮЧЕНИЕ

В этой главе мы изучали модель абстрактной реализации императивных языков программирования. Конкретная реализация является более подробной. Однако эта дискуссия дает нам возможность лучше понять реализацию и доступ различных данных, абстракций управления на промежуточном уровне.
В этой главе обсуждалась концепция абстрактной машины и её поведение с точки зрения области данных, области управления, указателя команд и регистров. Область данных постоянно изменяется, используя фиксированный код, за исключением языков, поддерживающих объекты первого класса. Область данных состоит из трех частей: статической области, управления стеком и множества . Статическая область фиксируется во время компиляции и не растет или уменьшается во время выполнения. Стек и множество могут увеличиваться и уменьшаться во время выполнения. Стек растет линейно. Множество — это общая область памяти, видимая для всех подпрограмм, и ячейки данных могут быть выделены по требованию в любое время. Физические ячейки логической структуры данных множества связаны посредством цепочек указателей.
Статическое распределение является схемой распределения фиксированной памяти, которая распределяет память во время компиляции так, что доступ к ячейкам памяти осуществляется напрямую, без использования какого-либо указателя. Статическое распределение

используется для объектов данных, которым необходимо сопоставление фиксированной памяти, без изменения ячейки памяти во время выполнения. Глобальные и статические переменные распределяются с помощью статического распределения. Локальные динамические переменные и объекты данных с ограниченным сроком службы в рамках модулей программы или блока, где они объявлены, распределяются в стеке элемента управления. Структуры рекурсивных данных и объекты динамических данных, у которых срок службы больше программного блока, где они объявлены, распределяются в множестве.

Диаграмма управляющей логики используется для перевода абстракций управления высокого уровня в команды абстрактного уровня. Отрицание условного выражения, за которым следуют команды условного ветвления, такие как *brlt*, *breq*, *brne*, *brgt*, *brge* и *brne*, используются для переноса управления на <else-statement> (оператор «иначе») в конструкции условного оператора и выхода из цикла в итерационных конструкциях.

Статическая реализация использует статическое распределение и прямой доступ к данным. Для вызова процедур необходимо сохранить адрес памяти следующей выполняемой команды и сделать переход с помощью команды ветвления для передачи управления вызываемой подпрограмме. После завершения всех команд, вызываемая подпрограмма присваивает значение указателя возврата указателю команд для передачи управления обратно следующей команде вызывающей подпрограммы.

Реализация на основе стека использует несколько указателей и отдельные кадры для каждого вызова подпрограммы. Указатель кадра вместе со смещением используется для доступа к ячейке памяти локальной переменной или объекта данных. Динамическая ссылка используется для получения обратно кадра вызывающей подпрограммы после завершения вызываемой подпрограммы. Статическая ссылка или регистры отображения используются для доступа к нелокальным переменным. Перед вызовом подпрограммы, параметры передаются в пространство исходящего параметра, которое становится пространством входящих параметров для вызываемой подпрограммы. Регистры, указатели и статус слова PSW вызывающей подпрограммы сохраняются; указатели изменяются для доступа к кадру вызываемой процедуры, а управление передается первой команде вызываемой подпрограммы. После завершения вызываемой подпрограммы, используется обратный процесс для восстановления среды вызывающей программы, передачи значения параметров, в зависимости от параметров механизма передачи, и передачи обратно управления к следующей расположенным в

множестве вызывающей подпрограммы.
Чтобы поделиться составным объектом, расположенным в множестве используется вызов по значению, позволяющий получить доступ к объекту данных, расположенному в множестве. Во время вызова по ссылке, указатель на фактический параметр сохраняется в ячейке формального параметра. Для поиска значения переменной r потребуется два обращения к памяти: одно для доступа к ячейке памяти фактического параметра, а второе для доступа к r-значению. Вызов по значению результата копирует фактические параметры исходящей области, которая становится областью входящего параметра в вызываемой подпрограмме. После возвращения из вызываемой подпрограммы, результаты из области исходящего параметра копируются обратно в ячейки фактической памяти переменных.

Рекурсивные функции создают новый кадр для каждого вызова. Статическая и динамическая ссылки рекурсивной функции указывают по-разному: динамическая ссылка указывает на базу кадра вызывающей процедуры, а статическая ссылка всех вызовов указывает на базу кадра процедуры, согласно которой вложена рекурсивная функция.
Обработка исключений реализуется с помощью одного из двух методов: (1) сохранения местоположения процедур обработки исключений в кадре подпрограммы, или (2) отдельного стека исключений. При первом методе, если все обработчики исключений дают сбой, то элемент управления возвращается к вызывающей программе, и процесс повторяется. При втором методе, если обработчик исключений выполняется успешно, то все кадры выше успешного кадра удаляются из стека исключений.

5.9 ОЦЕНКА

5.9.1 Концепции и Определения

Абстрактные инструкции; абстрактная машина; активное пространство; вызов по значению; вызов по ссылке; вызов по значению результата; область кода; стек управления; область данных; регистры отображения, динамическая ссылка; обработка исключений; стек исключений, кадр; указатель кадра; множество; область ожидания; рекурсивные структуры данных; рекурсивные процедуры; указатель возврата; распределение на основе стека; статическое распределение; SECD машина; статическая ссылка; указатель вершины стека; след стека, WAM (абстрактная машина Уоррена).

5.9.2 Решение Задач

1. Написать последовательность низкоуровневых кодов для реализации утверждения X = Y + 2 + 3 + 5.

2. Нарисуйте диаграмму управляющей логики для оператора варианта, а затем напишите низкоуровневый абстрактный код для реализации оператора варианта.

3. Напишите код области и область данных для следующей, похожей на Fortran 66 программы, используя статическое распределение.

PROGRAM MAIN	**ПОДПРОГРАММА** СОРТИРОВАТЬ	**ПОДПРОГРАММА** МАКС
ЦЕЛОЕ ЧИСЛО I, J, K[20]	ЦЕЛОЕ ЧИСЛО M[20], I, J	ЦЕЛОЕ ЧИСЛО M[20], I, J, K
ОБЩИЙ/B1/K[20]	ОБЩИЙ/B1/M [20]	ОБЩИЙ/B1/M [20]
ДЕЛАТЬ 20 I = 1, 20, 1	ДЕЛАТЬ 10 I = 1, 20, 1	ДЕЛАТЬ 30 I = 1, J, 1
20 ЧИТАТЬ(K[I])	J = 20 − I + 1	ЕСЛИ (M [I] .GT. M[I + 1])
ВЫЗОВ СОРТИРОВАТЬ	10 ВЫЗОВ МАКС	K = M[I]
ДЕЛАТЬ 30 I = 1, 20, 1	ВОЗВРАТ	M[I] = M[I + 1]
30 ПИСАТЬ(K[I])		M[I + 1] = K
КОНЕЦ		КОНЕЦЕСЛИ
		ВОЗВРАТ

4. Напишите функцию, которая обнаруживает *факториал*(n) и покажите стек управления для вычисления *факториал*(2), когда элемент управления находится в вызове, вычисляющем *факториал*(0).

5. Напишите область кода и область данных для следующей программы, используя распределение на основе стека для следующего блочно-структурированного языка. Предположим, что блочно-структурированный язык поддерживает три типа передачи параметров: вызов по значению, вызов по ссылке и вызов по значению результата. Вызов по ссылке обозначается тег «&» в фактическом параметре, а вызов по значению результата обозначается знаком «#» перед фактическим параметром. В вызываемой подпрограмме доступ к значению фактического параметра осуществляется через знак «*» перед формальным параметром.

```
программа основная ()
целое число значение макс;
{       целое число i, j, a[20]
        для (i = 0; i = < 19, i++)
              читать(a[i]);
              сортировать(ссылка
              &a); значение макс =
              d[19]; i = 0;
              делать
                   писать(a[i])
              пока (i ++ = < 19);
              писать(макс);
}
```

```
пустая операция
сортировать
(целое число *d)
целое
число i, j
j = 20;
пока (j>1)
  {макс (d # j);
возврат
```

```
пустая операция макс
(целое число * d, j)
{целое число i,
    шаблон; i = 0;
пока (i = < j) {
  если (* d [i] < *d[i+1]) {
       шаблон = *d[i];
            *d[i] = *d[i+1];
            *d[i+1] =
            шаблон;
    }
  }
  j = j − 1;
}
```

6. Напишите простую программу для пузырьковой сортировки, которая вызывает процедуру find -max с помощью обмена соседними значениями вместо хранения временного макс. Она передает массив с помощью вызова по ссылке. Покажите области данных и кода, и выделите параметр передачи кода. Четко обозначьте указатели, когда элемент управления находится в основной программе, и когда он находится в процедуре find–max. Предположим, что размер массива равен 5.

5.9.3 Развернутый ответ

7. Каковы основные недостатки статического распределения? Как схема гибридного распределения, состоящая из статического распределения, распределения на основе стека и на основе множества решает задачу?

8. Объясните роль кадров в реализации на основе стека. Почему необходимо сохранять видимость, ограниченную, главным образом, местным кадром? Объясните.

9. Каковы преимущества наложения области исходящего и входящих параметров? Объясните, с помощью использования схем.

10. Исходя из того, что у вас имеется управление стеком и множеством, разработайте механизм передачи параметра, который действует как вызов по ссылке только для чтения, во время выполнения вызываемой подпрограммы, если значение в настоящее время читается и передает любые изменения фактическому параметру в конце вызываемой процедуры для обновления части фактического параметра, которая изменилась во время выполнения вызываемой процедуры. Обсудите

преимущества и недостатки такой схемы, по сравнению с вызовом по ссылке и вызовом по значению результата.

11. Просмотрите литературу для реализации итераторов, и напишите различные методы реализации итераторов и проблемы внедрения итераторов.

12. Что такое обработчики исключений, и как с ними работать в реализации на основе стека?

13. Нарисуйте схему управляющей логики для выбора конструкции с несколькими условными выходами в ADA , и напишите код нижнего уровня для конструкции.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Брент Р. «Эффективная реализация первой подходящей стратегии для распределения динамического хранения». *Транзакции ACM на Языке Программирования и Системы*. 11(3). Июль 1989 года. 388-403.

Диль, Стефан, Хартель, Питер и Сештофт, Питер. «Абстрактные машины для реализации языков программирования». *Компьютерные системы будущего поколения*, 16.2000. 739-751.

Хансон, Дэвид Р. «Быстрое распределение и перемещение памяти на основе срока службы.» *Опыт и Практика Программного Обеспечения*, 20(1). Январь 1990 года. 5-12.

Джонс, Ричард, Хоскинг, Антоний и Мосс, Элиот. *Справочник Коллекции Мусора*. CRC Пресс / Группа Тейлор и Фрэнсис. 2012. 481.

Джонс, Ричард и Линс, Рафаэль Д. *Коллекция Мусора: Алгоритмы Автоматического Управления Динамической Памятью*. Нью-Йорк: Джон Уайли. 1996.

Уилсон, Пол Р., Джонстон, Марк С., Нили, Майкл и Боулз, Дэвид. «Распределение динамического хранения: Обзор и критический анализ.» *Работа IWMM ' 95 Материалы Международного Семинара по Менеджменту Памяти*. Спрингер Ферлаг. 1 – 116.

Глава 6

Управление динамической памятью

БАЗОВЫЕ КОНЦЕПЦИИ

Абстрактные понятия в вычислениях (Раздел 2.4); Абстрактная реализация (Глава 5); Абстракции данных (Раздел 4.1); Концепции структур данных (Раздел 2.3); Принцип локальности (Раздел 2.4.8).

Повторное использование памяти является важным требованием для выполнения крупномасштабных программ. Различные подмодули программы и динамически созданные объекты имеют время существования. Выделенная память освобождается, когда соответствующие блоки программы завершили выполнение, или рекурсивные структуры данных или объект динамических данных высвобождены вручную действием программиста. Память выделяется в *стеке управления* или множестве. Память из стека управления высвобождается, когда отбрасывается кадр вызываемой процедуры, и ячейка памяти может быть повторно использована для распределения другого кадра. Множество — это общая область, используемая, в основном, для рекурсивных структур данных и динамически создаваемых объектов данных в объектно-ориентированных языках программирования, таких как C++, Java и C#. Структуры данных, размещенные в множестве, расширяются динамически на основе запроса программиста во время выполнения, и имеют время жизни большее, чем у процедур, в которых они были созданы. Множество видимо на протяжении всей жизни программы. Выделенная множеству память используется повторно только после высвобождения объектов данных.

Управление динамической памятью касается распределения, перемещения и переработки ячеек памяти, необходимых для структур данных, выделенных множеству во время выполнения. Если объекты будут расширяться по частям, то к ним будет трудно получить доступ в кадре процедуры, поскольку различные распределения должны идти по цепочке и доступны по порядку, в то время, как доступ к объектам

данных в кадре осуществляется с помощью индекса и смещений. Структуры данных и объекты, построенные один раз и удаленные до истечения срока службы подпрограммы, которая их создала, могут быть распределены в стеке управления для обеспечения эффективного доступа и лучшего восстановления памяти. Язык, который поддерживает распределение сложных динамических объектов в стеке управления, должен поддерживать *вызов по ссылке* для совместного использования объекта с вызываемой подпрограммой. Если объекты динамических данных размещаются в множестве, то стек управления сохраняет только указатель на первую ячейку памяти динамического объекта, и *вызов по значению* (или *вызов по обмену*) используется для перевода указателя на доступ к объекту в вызываемой подпрограмме.

6.1 Структура множества

Множество состоит из линейного массива, в котором размещаются динамические объекты данных. У множества есть три типа блоков памяти: *выделенный*, *освобожденный* и *свободный. Выделенными* блоками являются активные блоки, используемые процессом. *Освобожденные блоки* не используются процессом и должны быть переработаны, чтобы оказаться в свободном блоке. *Блоки свободной памяти* могут быть выделены для процесса, запрашиваемого во время выполнения. Каждый тип блока памяти содержит еще информацию о своем размере. Множество можно моделировать как последовательность четверок, где элемент последовательности может быть (*выделенный,* <<block-size> (размер блока)>, <start-address> <end-address> (начальный и конечный адрес) или освобожденный размер блока, <start-address> <end-address> (начальный и конечный адрес) или (*свободный,* размер блока, <start-address> <end-address> (начальный и конечный адрес).

Свободные блоки группируются вместе либо с помощью цепочки указателей, либо, используя индексируемую структуру, которая группирует блоки памяти аналогичных размеров с помощью цепочки указателей. Правильную цепочку можно найти с помощью схемы эффективного поиска, такого как бинарный поиск, или хэш-функций. После соединения цепи определяются блоки аналогичного размера, а оптимальный блок может быть идентифицирован последовательно, следуя по цепочке.

Первые указатели объектов множества хранятся в регистрах процессора, управлении стеком, или дескрипторе — кортеже ссылок на ресурсы, используемые процессом. Доступ к ячейкам данных такой

же структуры данных, разбросанных в множестве, осуществляется через цепь указателей, соединяющих блоки рассеянной памяти, принадлежащие к той же структуре данных. Во время перехода параметра можно получить доступ к объекту данных в вызываемых подпрограммах, путем копирования первого указателя на динамический объект. Для доступа к объектам множества требуется только вызов по значению для копирования ссылки на объекты.

Пример 6.1

На Рисунке 6.1 множество начинается в ячейке 13,500 и заканчивается в ячейке 0. Оно хранит два вида объектов данных. Объект данных 1 распространяется на три блока памяти: 13500-11,501, 8000-6001 и 5000 – 3001; Объект данных 2 распространяется на два блока памяти: 10,500–8001

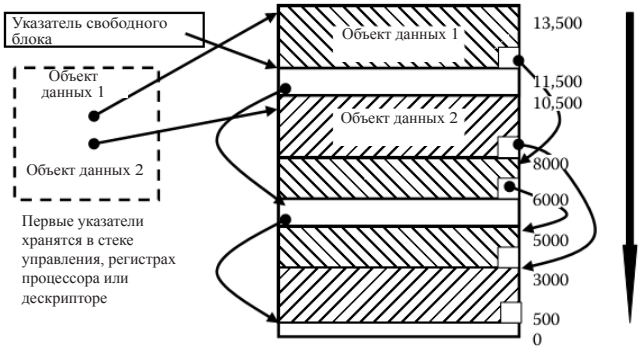


РИСУНОК 6.1 Схема устройства множества.

и 3000 – 501. Первый указатель объекта данных 1 указывает на базовый адрес 13,500, а первый указатель объекта данных 2 указывает на базовый адрес 10,500. Свободные блоки также перемежаются в трех диапазонах памяти: 11,500-10,501, 6000-5001 и 500-0, причем указатель свободного блока указывает на базу первого свободного блока, расположенного в 11,500.

Блоки множества выделяются на основании запроса во время выполнения *процесса* — активная версия программы. Блок множества представляет собой набор ячеек данных, имеющий три типа полей: *поле заголовка*, *информационные поля* и *поля*, *указывающие на другие ячейки данных*. *Поле заголовка* содержит многочисленную информацию, такую как флажки для пометки

ячеек по мере распределения, ряд полей и их смещений памяти от начала ячейки, ряд ячеек памяти в ячейке, и ряд указателей, указывающих на ячейку. Информационное поле содержит тип информации и значения. Например, связной список представляет собой парную последовательность (заголовок, инфо-поле, указатель следующей ячейки), а ячейка бинарного дерева — четверной кортеж формы (заголовок, инфо-поле, левый-указатель, правый-указатель), как показано на Рисунке 6.2. Ячейки в структуре логических данных отслеживаются с помощью цепочки указателей, соединяющих блоки множества в одной и той же структуре данных.

Свободное место множества может быть организовано четырьмя различными способами: (1) как цепь свободных блоков, как показано на Рисунке 6.1; (2) как индексированные цепочки указателей, связывающие блоки аналогичных размеров вместе, как показано на Рисунке 6.3; (3) как группа индексируемых стеков, как показано на Рисунке 6.4; или (4) как битовые маркеры распределения. Третья

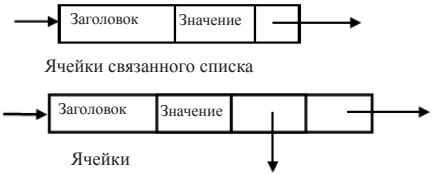


РИСУНОК 6.2 Распределение ячеек в блоках связанных списков и деревьев.

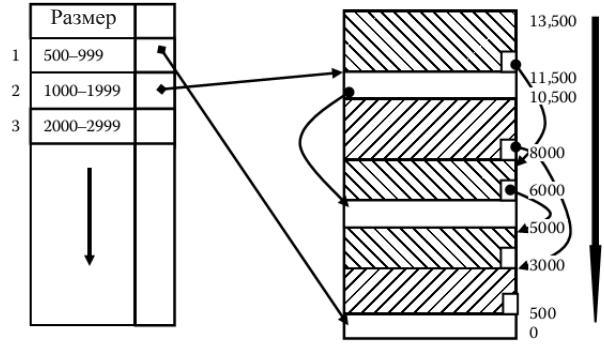


РИСУНОК 6.3 Раскладка блоков множества по размеру для быстрого поиска оптимального блока.

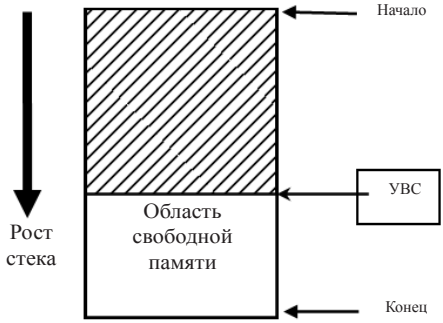


РИСУНОК 6.4 Распределение множества на основе стека.

схема похожа на стек управления, изучаемый в Главе 5. Четвертая схема популярна в системе программирования для отображения файлов на физических блоках во вторичной памяти.

На Рисунке 6.3 показана схема организации множества для эффективного поиска блока свободной памяти оптимального размера. Индексированная таблица составляется из блоков одного размера. Соответствующая цепь свободных блоков определяется с помощью схем эффективного поиска.

В распределении на основе стека, множество представлена как кусок ячеек памяти, которые помечены как начальный адрес и конечный адрес. *Указатель вершины стека* (УВС) начинает с начального адреса и движется в направлении конечного адреса, после каждого выделения запрашиваемого блока памяти во время выполнения. УВС используется для выделения следующего запрашиваемого блока памяти и запуска процесса сбора-переработки мусора освобожденных блоков памяти, который возвращает их в накопитель свободной памяти. Когда значение УВС проходит мимо конечного адреса стека, начинается сборка мусора.

В *битовой схеме* пространство множества разделяется на блоки фиксированных размеров, которые сопоставляются с битовым блоком таким образом, чтобы смещение адреса бита от начала битового блока представляло собой индекс блока данных множества. Если бит имеет значение «1», то выделяется блок с таким же индексом, как у бита со смещенным адресом; в противном случае он остается свободным. Список свободных блоков можно определить путем (1) обхода битового блока для выявления бит с нулевым значением, или (2) сохраняя последовательность смещений адреса бит с нулевым значением.

6.2 РАСПРЕДЕЛЕНИЕ ОБЪЕКТОВ ДИНАМИЧЕСКИХ ДАННЫХ

Объекты данных распределяются с помощью конструкторов в объектно-ориентированном программировании, таких как C++, C# или Java, при создании объектов динамических данных, или программно, с помощью явного запроса выделения памяти как в языке C. При появлении запроса, множество проверяется на наличие доступной памяти. При наличии, выделяется память. В противном случае, задействуются освобожденные блоки памяти, восстановленные процессом сбора-переработки мусора. Есть три популярных схемы распределения памяти: *первый пригодный*, *следующий пригодный* и *самый пригодный* (или *оптимально пригодный*).

В алгоритме *первый пригодный*, выделяется первый свободный блок, который больше, чем выделенный блок запрашиваемой памяти, а заголовок нового выделенного блока и цепочка указателей подключения свободного блока, настраиваются соответствующим образом. Указатель в предыдущем блоке обновляется, чтобы указывать на следующий за последним адресом, адрес заново выделенного блока памяти.

Схема следующий-пригодный является вариантом схемы первого-пригодного. Она основана на запоминании позиции последнего свободного блока памяти в ходе последнего поиска. В следующем запросе, поиск продолжается с этой позиции. По достижении конца множества, он начинается снова, с самого начала множества. Преимуществом этой схемы является более равномерное распределение выделенных блоков. При равномерном распределении блоков памяти, уменьшается вероятность и объем *фрагментации*. *Фрагментация* происходит из-за оставшихся небольших блоков памяти, у которых недостаточно пространства смежной памяти для выполнения запроса блоков памяти. Однако, это не простое преимущество, потому что у разных подпрограмм разное время жизни, и у разных программ разный объем распределения и перемещения объектов множества. Подпрограмма, у которой много распределений и перемещений объектов множества, в большей степени способствует фрагментации. Распределение может быть произведено либо с помощью блоков фиксированного размера, либо с помощью блоков переменного размера. Идея заключается в том, чтобы избежать фрагментации. Если размер блока памяти после распределения очень мал, то вместо того, чтобы сохранять фрагмент свободного блока, выделяется полностью весь блок, чтобы избежать фрагментации. После выделения памяти устанавливается указатель с расширяемого объекта данных на новые выделенные ячейки данных. Если по запросу

выделяется цельный свободный блок, то свободный блок удаляется из связного списка свободных блоков. В противном случае, цепочка свободных блоков корректируется соответствующим образом, чтобы указывать на оставшийся более мелкий свободный блок.

Схема самый-пригодный находит наиболее оптимальный размер, подходящий больше всего запрашиваемому размеру. Больше всего схема самый-пригодный подходит индексируемой структуре множества, когда множество устроена из группы свободных блоков схожего размера, соединенных через связные списки. Схемы эффективного поиска идентифицируют цепочку указателей с ближайшим диапазоном и просматривают цепочку указателей, чтобы найти наиболее подходящий размер, соответствующий запросу распределения. И, опять же, если вычисление показывает, что размер фрагмента оставшегося блока очень мал, то выделяется полностью весь блок, а цепочка указателей настраивается соответственно. Если оставшийся размер блока больше минимально допустимого фрагмента, то вычисляется размер остатка свободного блока, а новый свободный блок стирается из текущего связного списка указателей и вставляется в связной список свободных блоков, отражая, надлежащим образом, размер маленького свободного блока. В случае, если разыскиваемая цепочка указателей оказывается пустой, то поиск проводится в следующем диапазоне большего размера до тех пор, пока не будет найдена незанятая цепочка указателей. Если незанятая цепочка указателей распознана, то начинается процесс переработки памяти с помощью *сборки мусора*.

6.3 ОСВОБОЖДЕНИЕ ОБЪЕКТОВ ДИНАМИЧЕСКИХ ДАННЫХ

Освобождение осуществляется двумя способами: (1) удаление структуры данных программистом вручную; или (2) автоматическое удаление объектов по завершении вызываемой подпрограммы, и время жизни динамически созданного объекта ограничивается временем жизни

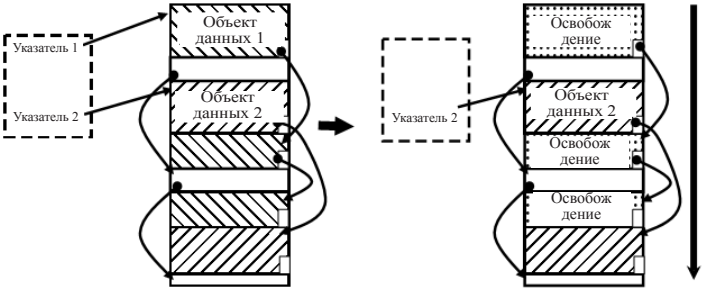


РИСУНОК 6.5 Высвобождение объекта из множества.

вызываемой подпрограммы. После удаления объекта динамической памяти, освобождается блок занятой памяти, готовый к возвращению в накопитель свободных блоков. Для освобождения объекта не требуется проверять каждую ячейку памяти, занятую динамическим объектом данных. Вместо этого первый указатель, который указывает на базовый адрес первого блока памяти динамического объекта данных, сбрасывается на *нуль*. В процессе сборки мусора обход начинается со сбора первых указателей на сохраненные динамические объекты данных множества. Ячейки памяти, недоступные через цепочку указателей, после сбора первых указателей, считаются мусором и собираются как свободные блоки. Поэтому, для освобождения целого динамического объекта данных достаточно удаления указателя из набора первых указателей. перемежающихся показано на Рисунке 6.5.

6.4 ФРАГМЕНТАЦИЯ

Фрагментация является процессом формирования перемежающихся блоков маленького размера, которые нельзя выделить по запросу в блок памяти большего размера. По мере того, как процесс выделения и освобождения продолжается, размер свободных блоков продолжает сокращаться, а число свободных блоков очень маленького размера продолжает увеличиваться. Эти, скомбинированные вместе фрагменты, могут выделяться при запросах блоков с большим объемом памяти. Однако, по отдельности эти свободные блоки слишком малы, чтобы быть пригодными.

Пример 6.2

Давайте рассмотрим пример множества, состоящей из следующих

блоков: {(выделенный, 200, 0, 199), (свободный, 400, 200, 599), (выделенный, 300, 600, 899), (выделенный, 500, 900, 1399), (свободный, 200, 1400, 1599)}. Общая сумма пространства комбинированной свободной памяти, разбросанной по фрагментам, составляет 600 байт. Запрос поступает на 500 байт. Ни один из свободных блоков не может обработать запрос индивидуально.

Можно возразить, что запросы можно было бы разбить на мелкие запросы и соединить в цепочку для размещения большого запроса. Однако, существует еще одна проблема с фрагментацией. Если структура данных разбросана по всему множеству, то ее можно разбить на разные страницы в области виртуальной памяти. Принцип локальности утверждает, что программа доступа к данным, или объект управления, работают в физической близости к программе. Компьютерное оборудование использует принцип локальности для (1) заполнения кэша блоком физически смежных ячеек памяти и (2) загрузки физически смежных страниц с жестких дисков в оперативную память когда происходит ошибка страницы. Если структура данных разбросана по множеству, то кэш не получит следующую ячейку объекта данных из-за физической удаленности, а недействительная страница не выдаст смежные ячейки объекту данных ввиду значительного расстояния между физическими адресами. Это означает, что затраты на заполнение кэша или ошибки страницы будут нарастать по мере увеличения фрагментации множества.

Существует два подхода к решению проблемы *внешней фрагментации*: (1) избегать внешней фрагментации; или (2) соединить внешние фрагменты во время сборки мусора для формирования более крупных блоков свободной памяти. Появления *внешних фрагментов* можно избежать, если использовать блоки фиксированного размера и выделять несколько блоков фиксированного размера при появлении запроса. Однако, использование блоков фиксированного размера приводит к *внутренней фрагментации*, так как $M \times$ *блок-фиксированного-размера* $-$ *запрашиваемая память* ≥ 0. Такая бесполезная память в последнем блоке выделенной памяти называется *внутренними фрагментами*. Внешние фрагменты можно соединить, если (1) смежные блоки идентифицированы как свободные блоки во время сборки мусора; или (2) все блоки с заполненным состоянием передвинуты на новую ячейку памяти. При единовременном копировании одной структуры логических данных, логически смежные структуры данных также становятся физически смежными.

6.5 СБОРКА МУСОРА — ВОССТАНОВЛЕНИЕ ПАМЯТИ МНОЖЕСТВА

Ячейки памяти множества проходят через три состояния, как показано на Рисунке 6.6. Ячейка памяти становится *активной* после её выделения объекту данных процесса. Процесс использует блок активной памяти и, в конце концов, освобождает блок. После ухода объекта данных, занимающего память, ячейка памяти становится освобожденной. Освобожденная память все еще недоступна для выделения другому объекту данных, потому что не является частью цепочки блоков свободной памяти. Память восстанавливается с помощью программного процесса, называемого *сборка мусора*, который берет освобожденную память и делает её частью блоков свободной памяти. Ячейка памяти считается *свободной*, если она становится частью накопителя свободной памяти и может быть выделена процессу.

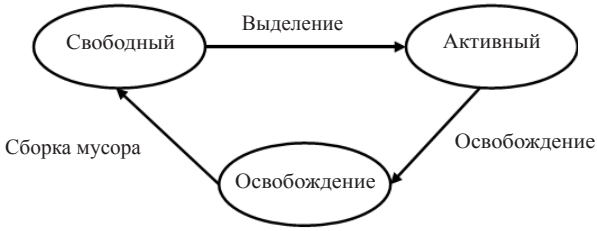


РИСУНОК 6.6 Состояния ячеек памяти множества

После того, как объект высвобожден из множества, можно предпринять разные действия, в зависимости от методов восстановления памяти:

1. Освобожденный объект собирается и немедленно перерабатывается. Такая сборка мусора называется *непрерывной сборкой мусора*.

2. Освобожденный объект помечается для переработки. Однако, восстановление памяти откладывается на будущее, когда множеству не хватит блоков свободной памяти.

Процесс сборки мусора начинается с набора первых указателей (коренные указатели) и проходит с помощью цепочки указателей, для маркировки действующих объектов данных. После идентификации действующих объектов данных можно использовать два подхода:

1. Действующие объекты данных перемещаются в новые смежные ячейки, высвобождая пространство старой памяти.

2. Область неактивной памяти множества сцепляется вместе как свободное пространство и используется повторно.

У сборки мусора есть время выполнения и затраты памяти, тормозящие выполнение программы. Затраты на выполнение современной сборки мусора составляют около 20-30% от общего времени выполнения. В схемах сборки мусора, приостанавливающих выполнение программы на сборку мусора, теряются важные события или данные, происходящие в реальном времени, вследствие задержки, вызванной временной приостановкой программы. Например, система мониторинга ядерного завода может пропустить важную радиоактивную утечку; или на космическом корабле могут упустить важные показания датчика; или робот-маляр может неожиданно остановить покраску в мастерской компании, выпускающей автомобили, что приведет к сбоям в работе сборочной линии. В интерактивных программах такая задержка приводит к ощутимому ухудшению производительности. Например, рендеринг (обработку) видеоклипа невозможно произвести в реальном времени.

6.5.1 Подходы к сборке мусора

Процесс сборки мусора может быть (1) *стартстопным* или (2) *в реальном времени*. Стартстопный алгоритм временно приостанавливает выполнение программы, когда начинается сборка мусора. Сборка мусора в реальном времени позволяет чередовать выполнение программы и сборку мусора. *Сборка мусора в реальном времени* не приостанавливает программу полностью во время сборки мусора и лучше обрабатывает собятия реального времени. Существует множество подходов для осуществления сборки мусора в реальном времени, таких, как (1) *инкрементная сборка мусора,* (2) *непрерывная сборка мусора*, (3) *параллельная сборка мусора*, и (4) *жесткая сборка мусора в режиме реального времени*.

Инкрементная сборка мусора дает возможность выполнения программы во время сборки мусора, путем деления сборки мусора на множество мелких сборок, чередующихся с выполнением программы. Однако, во время этого более короткого периода сборки мусора,

выполнение программы все же приостанавливается. Чередование корректно притормаживает выполнение программ в ходе процесса сборки мусора, обеспечивая ограниченное взаимодействие с внешним миром.

Непрерывная сборка мусора сразу же собирает освобожденную память и помещает ее в цепочку свободного пространства. Однако, возникают затраты на освобождение и помещение в свободное пространство. Сборка мусора с подсчетом ссылок (*Примечание переводчика: подсчёт ссылок - алгоритм сборки мусора, ведущий для каждого объекта программы подсчёт текущего числа других объектов, которые на него ссылаются. Если это число становится равным нулю, память соответствующего объекта можно безопасно освободить, а общее число ссылок каждого из ссылающихся на него других объектов уменьшить на единицу, однако, если у какого-то из них счётчик ссылок тоже обнулится, то процедура просмотра повторяется и может повторяться неопределённое число раз, поэтому данный алгоритм не рекомендуется для систем реального времени. У этого алгоритма возникают также проблемы с объектами, ссылающимися друг на друга по кольцу, так как у них число ссылок никогда не бывает меньше 1) является популярной схемой непрерывной сборки мусора. Параллельная сборка мусора позволяет собирать мусор вместе с выполнением программы, используя несколько потоков или несколько модулей обработки: отдельно модули, производящие сборку мусора и отдельно - выполняющие программу. Процессы утилизации освобожденной памяти и процесс выделения памяти возможны, если они соответствуют следующим условиям:

1. Процесс сбора пространства памяти и процесс выделения памяти ведутся раздельно.

2. Два процесса синхронизируются при работе над одной и той же ячейкой памяти.

При использовании многоядерных и многопроцессорных компьютеров, параллельная сборка мусора становится очень популярной. Чередование выполнения программы и сборки мусора означает, что новые ячейки памяти также выделяются во время сборки мусора. Хотя схемы инкрементной сборки мусора и параллельной сборки мусора помогают корректно сократить временную задержку снижения производительности, есть еще много работ в реальном времени -

таких, как управление космическим аппаратом, автоматизированный контроль управления, контроль двигателя в современных автомобилях, управление роботом и много других критических задач, с жесткими сроками завершения работ. Схемы жесткой сборки мусора в реальном времени стараются решить эту задачу на уровне планировщика путем временной приостановки сборки мусора, если необходимо выполнить первоочередное прерывание или процесс.

У сборки мусора есть два вида процессов: *мутатор* и *коллектор*. *Мутатор* - это часть выполнения программы, которая может выделить новые ячейки памяти или освободить структуру данных во время сборки мусора. *Коллектор* - это процесс сборки мусора, конвертирующий блоки освобожденной памяти в блоки свободной памяти. Структуры данных множества могут быть смоделированы как направленный циклический *граф* с тремя видами узлов: *активным,* освобожденным и *свободным*. Активные узлы являются объединением всех узлов в нескольких структурах данных. У каждой структуры данных есть граф узлов. Такие структуры данных сцепляются вместе и стартуют с корневого указателя. Так как есть множественные структуры данных, то есть и множественные корневые указатели. Набор корневых указателей является отправной точкой сборки мусора. Процесс сборки мусора вовлекает сбор ряда активных узлов с использованием цепочки указателей. Как только активные узлы помечены, оставшиеся узлы считаются узлами мусора и собираются. Многие структуры данных разделяют часть своих структур данных. Поэтому, может оказаться более одного входящего соединения узла.

6.6 СТАРТСТОПНАЯ СБОРКА МУСОРА

Есть два основных подхода к стартстопной сборке мусора: (1) помечать все активные ячейки и затем собирать все оставшиеся ячейки как свободные, сцепляя их вместе; (2) перемещать активные ячейки в другую область памяти, освобождая старую область памяти.

Оба подхода были использованы: алгоритмы *маркировка-и-сканирование* используют первый подход, а *копирование сборки мусора* использует второй подход. В этом разделе дается описание трех популярных стартстопных методов, используемых в современных однопроцессорных машинах.

6.6.1 Алгоритм маркировка и сканирование

Алгоритм *маркировка-и-сканирование*, также известный как алгоритм *маркировка-и-поиск*, имеет два этапа: *этап* разметки и *этап поиска*. На этапе разметки, начиная от первых указателей, хранимых в регистрах процессора, стеках управления или дескрипторах, алгоритм рекурсивного спуска обходит все ячейки объекта данных, отслеживая цепочку указателей. Этап поиска перебирает последовательно весь массив множества, по одной ячейке памяти за раз, и собирает все ячейки памяти, для которых не установлена соответствующая битовая метка. После прохождения активной ячейки, он сбрасывает битовую метку для улучшения сбора ячеек памяти, освобождаемых в следующем цикле.

Обход использует поиск в глубину. Адрес информационного узла, содержащего более одного неисследованного указателя, помещается в стек, и неисследованный крайний левый указатель проверяется в глубину до тех пор, пока не будет найден лист - ячейка без исходящего указателя. После прохождения листа, самый верхний указатель выталкивается из стека, и узел снова проверяется в глубину оставшейся неизученной части графа, пока не останется больше непройденных узлов . Процесс повторяется для каждой структуры данных. Во время обхода заголовок каждой активной ячейки устанавливает свою *битовую метку* - битовый флажок в заголовке каждой ячейки данных в структуре данных. *Битовая метка* сообщает этапу поиска, что ячейка является активной и её не нужно собирать.

В ходе этапа поиска, куски смежной памяти группируются вместе в один большой блок памяти. Такие свободные блоки либо сцепляются в единую цепочку свободных блоков, либо в качестве более коротких индексируемых множественных цепочек собираются в свободные блоки схожего размера, как объяснялось в Разделе 6.1.

У алгоритма маркировки и сканирования есть много недостатков, как описано ниже

1. Алгоритм маркировки и сканирования обходит дважды активные ячейки: один раз во время этапа разметки активных ячеек, и затем во время этапа поиска для сбора ячеек, которые не были отмечены во время этапа разметки.

2. Алгоритм маркировки и сканирования использует алгоритм поиска в глубину для прохождения структур данных. Поиск в глубину является рекурсивным по своей природе. У любого рекурсивного алгоритма есть расходы памяти при использовании стека. Размер стека, по меньшей

мере, настолько же глубок, как и глубина структуры проходимых данных. Связные списки являются линейными и не нуждаются в стеке для прохождения узлов. Однако, для обхода дерева и графа стек необходим. Запись (*N*) является глубиной для сбалансированных бинарных деревьев, где *N* является количеством ячеек данных в дереве. Однако, чаще всего вследствие смежной вставки и стирания, деревья не сбалансированны, и глубина колеблется между записью (*N*) и *N*.

3. Традиционный алгоритм маркировки и сканирования является стартстопным и не подходит для языков, поддерживающих события в реальном времени.

4. Алгоритм маркировки и сканирования объединяет только смежные свободные блоки. Однако, свободные блоки разбрасываются вместе с вложенными активными блоками. С течением времени, размер свободных блоков становится меньше, а фрагментация увеличивается. Фрагментация вызывает (1) рост числа прерываний из-за отсутствия страницы и (2) увеличение частоты вызова сборки мусора; так как запросы памяти не могут быть выполнены в фрагментированном пространстве памяти. Фрагментация приводит к снижению скорости выполнения из-за (1) отсутствия удачного обращения в кэш; (2) избытка прерываний из-за отсутствия страницы; и (3) частого вызова сборки мусора из-за отсутствия утилизации очень маленьких блоков памяти.

5. Этап поиска алгоритма маркировки и сканирования зависит от размера множества, а не от количества свободных ячеек, так как он сканирует последовательно каждую ячейку памяти.

6.6.2 Копирование сборки мусора

Копирование сборки мусора собирает все разбросанные свободные области в одну большую область, не страдая от проблем фрагментации и зависимости от размера множества в алгоритме маркировки и сканирования. Есть три основных преимущества копирования сборки мусора:

1. Копирование сборки мусора сжимает все свободные блоки памяти в одно смежное пространство памяти, полностью исключая фрагментацию.

2. Копирование сборки мусора копирует за раз одну структуру данных, удерживая их в физически смежном пространстве памяти,

улучшая коэффициент совпадения кэша и сокращая прерывания из-за отсутствия страниц, что происходит ввиду перемежающихся структур данных.

3. Копируются только активные ячейки.

На рисунке 6.7 показана основная схема копирования сборки мусора. При копировании сборки мусора, множество делится на два пространства: *активное пространство* (также именуемое *из-пространство*) и *неактивное пространство* (также именуемое *в-пространство* или *неиспользуемое пространство*). В любое время для распределения объектов данных используется только активное пространство. Распределение производится с помощью организации множества на основе стека.

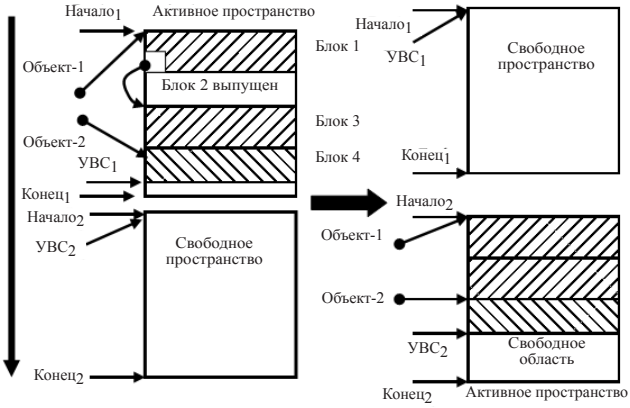


РИСУНОК 6.7 Копирование сборки мусора.

Когда *указатель вершины стека* достигает конечной метки, начинается сборка мусора. Во время сборки мусора активные структуры данных проходятся по одной за раз, начиная с первого указателя, хранящегося в регистрах, стеке управления или дескрипторах. Все перемежающиеся блоки памяти, соответствующие единичным структурам логических данных, копируются последовательно в физически смежное пространство для эффективной утилизации согласно принципу локальности. После того, как все структуры данных скопированы, пространства переключаются: активное пространство (из-пространство) становится неактивным пространством (в-пространство), а неактивное

пространство (в-пространство) становится активным пространством (из-пространство), и возобновляется выполнение программы.

У копирования сборки мусора есть следующие указатели: начальный-адрес, конечный-адрес, УВС и прямой указатель. Роль указателя начального-адреса, конечного-адреса и УВС такая же, как и используемая при распределении стека. Прямой указатель используется для копирования совместно используемых данных. Условно, множество растет в обратном направлении: от высшего к низшему адресу.

На левой стороне Рисунка 6.7 показан снимок множества до сборки мусора, а на правой стороне показан снимок множества после сборки мусора. Активное пространство до сборки мусора показывает два объекта данных: объект данных 1 и объект данных 2. Объект данных 1 занимает блок памяти 1 и 3, а объект данных 2 занимает блок памяти 4. Блок 2 является освобожденным блоком, который требует утилизации, вместе с маленьким фрагментом памяти на вершине стека. УВС очень близко расположен к конечному маркеру активного пространства.

Запрос на распределение блока памяти, такой как УВС - размер (запрашиваемый блок памяти) < конечная-метка, запускает следующую сборку мусора. Во время сборки мусора, блоки 1 и 3, принадлежащие объекту данных1, копируются и размещаются смежно, а за ними следует блок 4, принадлежащий объекту данных 2. Свободное пространство объединяется на другом конце.

Прямой указатель используется для указания от из-пространства на новую ячейку памяти в в-пространство, куда были перемещены данные. Каждая ячейка памяти в из-пространстве, которая была скопирована в в-пространство, перезаписывается с (1) тегом, что ячейка уже была скопирована и (2) прямой указатель показывает на соответствующие ячейки памяти в в-пространстве. Прямой указатель создается для того, чтобы избежать копирования ячеек данных, совместно используемых двумя структурами данных в двух разных ячейках памяти.

Многие структуры данных совместно используют часть ячеек памяти. Как только совместно используемая часть скопирована во время копирования первой структуры данных, совместно используемое пространство не может быть снова продублировано во время копирования второй структуры данных. Чтобы избежать дублирования совместно используемой структуры данных во время копирования, в ячейке старой памяти из-пространства хранится прямой указатель. После того, как во время обхода замечается тег прямой-указатель, то прямой указатель копируется в поле указателя последней посещенной ячейки второй

структуры данных, и процесс копирования второй структуры данных останавливается. Использование прямого указателя показано на Рисунке 6.8.

Пример 6.3

На Рисунке 6.8 показаны два объекта данных, которые совместно используют данные ячейки: «s-1» и «s-2». После копирования объекта данных$_1$ в оставленный интервал, нет необходимости копировать ячейки общего пользования «s-1» и «s-2.» Скопированные ячейки памяти переносят прямой указатель в

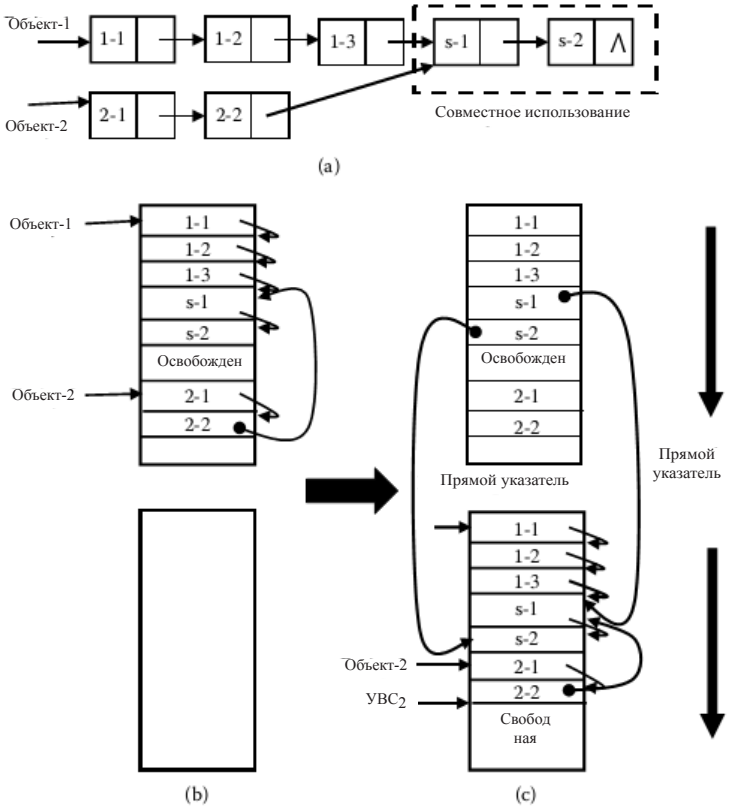


РИСУНОК 6.8 Роль прямого указателя во время копирования мусора. (a) Два связных списка совместного использования структуры данных;

(б) снимок перед копированием сборки мусора; (в) снимок во время копирования сборки мусора.

ячейки памяти, куда они были скопированы. После копирования ячейки «2-2» объекта данных2, третья ячейка «s-1» становится совместно используемой ячейкой. Такой прямой указатель, хранящийся в ячейке «s-1» в из-пространстве, копируется в поле указателя ячейки «2-2» в в-пространство. Нет необходимости копировать какие-либо дополнительные ячейки после установки указателя, так как совместно используемые ячейки уже находятся в в-пространстве.

У примитивного копирования сборки мусора есть следующие недостатки:

1. Это стартстопный алгоритм, не подходящий для программ, работающих с событиями в реальном времени.
2. Оно использует поиск в глубину, у которого излишний расход памяти пространства стека.
3. Только 50% пространства множества перерабатывается за раз.
4. Принцип локальности утверждает, что у старых структур данных меньшая степень вероятности быть использованными. Такие старые структуры данных продолжают копироваться между из-пространством и в-пространством каждый раз при сборке мусора, способствуя перерасходу времени выполнения с излишним копированием.

Существует два основных подхода к решению задач в примитивном копировании сборки мусора:

1. При первом подходе заменяются излишние расходы памяти стека с использованием *поиска в ширину*, рассматривая полное пространство-получатель как очередь. Использование поиска в ширину удаляет излишние расходы стека, а использование пространства-получателя в качестве очереди, удаляет присущие памяти излишние расходы с помощью очереди в поиске в ширину..
2. При втором подходе используются пространства нескольких версий вместо двух полупространств, сохраняя одно из пространств неактивным в конкретный момент времени. Такой подход улучшает утилизацию памяти на 100*(n−1)/n%. С пространствами трех версий, эффективность утилизации памяти составляет 67%, а с пространствами четырех версий, эффективность утилизации памяти возрастает до 75%.

6.6.3 Модифицированное копирование сборки мусора Чейни

На Рисунке 6.9 показан алгоритм, впервые разработанный Чейни, убирающий излишние расходы памяти стека путем использования поиска в ширину и утилизации свободного пространства-получателя в качестве очереди. Использование пространства-получателя в качестве очереди, позволяет избежать любых излишних расходов унаследованной памяти в отдельной очереди. Элементы данных пространства-получателя могут быть в трех состояниях: *черные ячейки*, *серые ячейки* и *белые ячейки. Черные ячейки* являются активными ячейками, которые уже посетили и скопировали в пространство-получатель. *Серые ячейки* были скопированы в пространство-получатель , но их дочерние записи находятся в пространстве-отправителе. *Белые ячейки* являются недостижимыми ячейками, к которым нельзя обратиться с любого из первых указателей. Схема использует дополнительный указатель, называемый скан-указатель, который начинается с указателя начала$_2$ в пространстве-получателе и достигает УВС в период сбора мусора. Скан-указатель действует как указатель, который указывает на проход к следующей ячейке в очереди, а УВС выступает в качестве указателя конца очереди, куда должны быть вставлены новые элементы данных. Ячейки памяти, на которые указывает скан-указатель, проверяются на наличие значений данных или указателя.

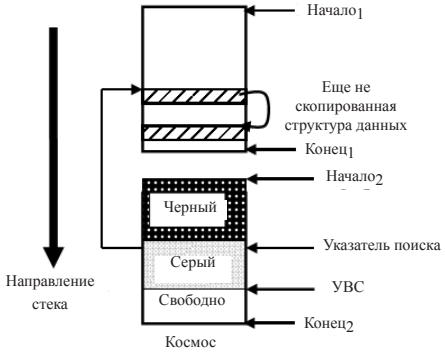


РИСУНОК 6.9 Схема сбора мусора Чейни, использующая поиск в ширину.

В зависимости от того, содержит ли ячейка данные или указатель, возможны следующие три действия:

1. Если область памяти содержит указатель на ячейку данных в пространстве-отправителе, то на вершине стека пространства-получателя появляется новая ячейка, значение ячейки памяти в пространства-отправителя копируется в эту новую ячейку и указатель следующей позиции пространства-отправителя переводится на эту новую ячейку. В скан-указателе добавляется одно место нахождения.

2. Если область памяти содержит элемент данных, то скан-указатель увеличивается на 1.

3. Если область памяти содержит указатель на место в памяти принимающей ячейки, то указатель поиска увеличивается на 1.

После того, как скан-указатель сходится с указателем вершины стека, все ячейки одной структуры данных копируются из пространства-отправителя в пространство-получатель. Процесс повторяется для каждой структуры данных, начиная с первого указателя. Когда все структуры данных были скопированы и скан-указатель достиг указателя вершины стека, процесс сбора мусора завершается.

6.6.4 Сборка мусора по поколениям

Сборка мусора по поколениям использует более двух пространств. Однако только одно пространство остается свободным. Пространство, которое заполняется полностью, копируется в свободное пространство. Основным преимуществом этой схемы является то, что (1) увеличивается степень использования памяти; (2) затраты на копирование в свободное пространство снижаются, поскольку размер порции меньше; (3) частота сбора мусора для объектов структур данных нового поколения увеличивается из-за принципа локальности и того факта, что доступ к наиболее последним используемым пространствам осуществляется чаще и они заполняются первыми; и (4), затраты на копирование более старых структур данных, которые не использовались, минимизированы путем хранения старых данных в более старых областях памяти.

Принцип локальности предполагает, что данные, доступ к которым осуществлялся недавно, получают доступ чаще, за исключением случаев, когда локальность меняется вызываемыми подпрограммами. Разделение множества на несколько небольших версий пространств использует это свойство выполнения программы: разные версии пространств (составных единиц множества), как правило, собирают структуры данных по их возрасту. Каждая версия пространства обозначается номером версии в

зависимости от его возраста. Общая схема показана на Рисунке 6.10.

Новые структуры данных, в большинстве своем, имеют короткий жизненный цикл и собираются до следующего цикла сбора мусора. Меньший процент переживает более одного цикла и его доля с возрастом постепенно уменьшается. Это означает, что очень старые структуры данных занимают гораздо меньшие версии пространств и увеличиваются очень медленно. Роль сбора мусора состоит из двух операций: (1) копировать новые структуры данных в свободное пространство и (2) если новая ячейка была выделена для очень старой структуры данных, она переносится в более старую версию пространства; где находится остальная часть старой структуры данных. Когда старая версия пространства становится полной, она собирается так же, как и любая другая версия пространства.

Одной из проблем, которую имеет схема нескольких версий пространств, является то, что, в отличие от копирования в процессе сбора мусора с двумя полупространствами, где все структуры данных хранятся в одном полупространстве, одна логически последовательная структура данных разбрасывается по нескольким версиям пространств. Большой процент таких структур данных является устаревшими данными. Существует необходимость смены указателей, когда часть структуры данных из одной версии пространства копируется в свободную версию пространства.

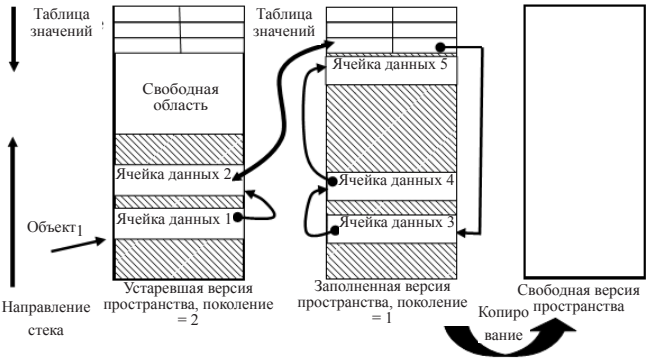


РИСУНОК 6.10 Сборщик мусора, учитывающий поколения объектов, и использующий таблицы значений.

Чтобы устранить проблему смены указателей в рассеянной по нескольким версиям пространств структуре данных, в каждой версии

пространства используется таблица указателей. Таблица указателей имеет две функции:

1. Она переводит указатели возврата с предыдущей ячейки той же структуры данных на следующую ячейку структуры данных, хранящуюся в соответствующей версии пространства.

2. Таблица указывает на следующую ячейку, хранящуюся в версии пространства. Преимуществом этой схемы является то, что, когда ячейки данных сжимаются при копирующем сборе мусора, адрес изменяется только в таблицах значений, а не в другой версии пространства.

Помимо всех указателей, используемых при копирующем сборе мусора, существует три дополнительных указателя для сбора мусора по поколениям: (1) указатель следующей позиции* (прямой указатель) в таблице значений, (2) указатель предыдущей позиции из таблицы значений* (обратный указатель) и (3) УВС для таблицы значений. Прямой указатель используется для указания из последней ячейки структуры данных в версии пространства на расположение таблицы значений, которая хранит адрес следующей ячейки в другой версии пространства. Обратный указатель указывает на предшествующую ячейку из таблицы значений, которая хранит адрес следующей ячейки. Если больше одной структуры данных совместно используют следующую ячейку, имеется связанный список обратных указателей, указывающих на все предыдущие ячейки, которые указывают на следующую ячейку в версии пространства. Таблица прямых и обратных указателей находится в верхней части версии пространства, как показано на Рис. 6.10. Таблица является расширяемой, растет как стек, в направлении, обратном росту множества и имеет необходимость в другой вершине стека.

Во время сбора мусора, когда таблица значений версии пространства копируется в неактивную версию пространства, должны быть осуществлены два действия:

1. Прямой указатель из предыдущей ячейки должен быть изменен, чтобы указать на новую запись таблицы в пространстве-получателе. Это делается с помощью обратного указателя, чтобы вернуться к предыдущей ячейке, затем сохранить адрес нового местоположения в пространстве-получателе, куда была скопирована запись таблицы.

2. Уплотнение выполняется для каждой таблицы значений, поскольку

каждая таблица значений указывает на первую ячейку частичной структуры данных, находящейся в версии пространства .

Пример 6.4

Рисунок 6.10 показывает множество, состоящее из трех версий пространств. Две левые версии пространств активны, а крайняя правая версия пространства – неактивная. Левая версия пространства старше, с возрастом = 2, а средняя версия пространство новее, с возрастом = 1. Ввод данных$_1$ затрагивает обе версии пространств. Версия пространства с поколением = 1 заполняется быстрее, в соответствии с принципом локальности и копируется чаще в свободное пространство. Вторая ячейка данных в более старой версии пространства (поколение = 2) имеет прямой указатель на запись таблицы, которая указывает на третью ячейку данных той же структуры данных в более новой версии пространства (поколение = 1). Та же запись таблицы в новой версии пространства имеет обратный указатель, указывающий на вторую ячейку в более старой версии пространства (возраст = 2).

Когда средняя версия пространства заполняется, начинается сбор мусора и средняя версия пространства уплотняется и копируется в крайнюю правую версию пространства. Адрес таблицы значений и ячейки данных меняются. Прямой указатель из второй ячейки изменяется. сначала с помощью обратного указателя получает адрес второй ячейки, а затем обновляет поле указателя второй ячейки.

В то время как сбор мусора по поколениям улучшает примитивный копирующий сбор мусора путем улучшения эффективности использования пространства памяти, не копируя старые структуры данных, он имеет дополнительные затраты вычислительных ресурсов при обслуживании таблицы значений для обработки структур данных, разбросанных по версиям пространств. По мере увеличения числа версий пространств, также увеличиваются затраты вычислительных ресурсов.

6.7 ИНКРЕМЕНТНЫЙ (ПОЭТАПНЫЙ) СБОР МУСОРА

Инкрементный сбор мусора (см. Рис. 6.11) объединяет в себе сбор мусора и работу основной программы. После начала сбора мусора, сборщик мусора выполняет частичный сбор мусора, сопровождаемый некоторым выполнением программы. Процесс сбора мусора после своего запуска переходит в следующую последовательность*: чсм, рп, чсм, рп, чсм, рп* ..., где *чсм* означает *частичный сбор мусора* и *рп* означает

работу программы. Период сбора мусора достаточно мал. Сбор мусора запускается довольно часто. В ходе сбора мусора,

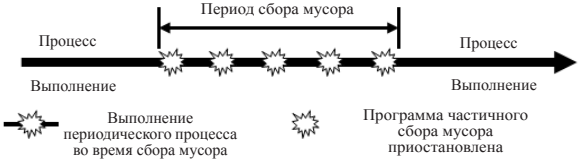


РИСУНОК 6.11 Инкрементный сбор мусора.

перерабатывается освобожденной памяти больше, чем выделяется новой памяти во время выполнения программы. После окончания сбора мусора, регулярные вычислительные процессы возобновляются.

6.7.1 Алгоритм Бейкера

Бейкер улучшил алгоритм Чейни периодически собирая ячейки K (K>>1) из пространства-отправителя в пространство-получатель перед помещением затребованной памяти в ячейку-получатель во время сбора мусора. Как только начинается сбор мусора, пространство-отправитель запечатывается и запрашиваемое выполняющимся процессом новое распределение памяти во время периода сбора мусора выполняется только в пространстве-получателе. Если запрашивается блок памяти размера *m*, то сначала ячейки *m* * *K* копируются из пространства-отправителя в пространство-получатель перед размещением ячеек *m* в пространстве-получателе. Новые ячейки располагаются на другом конце полупространства, чтобы обеспечить физическую целостность структуры данных, копируемых из пространства-отправителя в пространство-получатель, как показано на Рис. 6.12.

Как показано на Рис. 6.12, существует еще два указателя — верхний указатель и нижний указатель — в дополнение к скан-указателю. Нижний указатель указывает на следующее свободное место в свободном пространстве для копирования структур данных из пространства-отправителя, а верхний указатель указывает на следующую свободную область для новых перемещений. Нижний указатель и верхний указатель двигаются в противоположных направлениях. После копирования объектов данных из пространства-отправителя, нижний указатель фиксируется, и только верхний указатель двигается, до тех пор, пока не встретится с нижним указателем, в этот момент начинается новый цикл

сбора мусора.

Существует три типа ячеек: *черные*, *серые* и *белые*. *Черные ячейки* - это те ячейки, которые были скопированы из пространства-отправителя в пространство-получатель вместе с их дочерними узлами. В стеке пространства-получателя они находятся между маркером начала и скан-указателем. *Серые ячейки* - это те ячейки, которые были скопированы из пространства-отправителя в пространство-получатель. Однако, они до сих пор могут содержать указатели на другие клетки пространства-отправителя, которые не были скопированы в пространство-получатель. В стеке пространства-получателя они находятся между скан-указателем и нижним указателем. *Белые ячейки* – это ячейки пространства-отправителя, которые были освобождены и не должны копироваться в пространство-получатель. Недавно размещенные ячейки рассматриваются как черные ячейки.

Поскольку перемещаемые данные - это те же копируемые данные из пространства-получателя, не требуется никакой дополнительной памяти. Общая выделенная память обозначается как $N + N/K$,, где

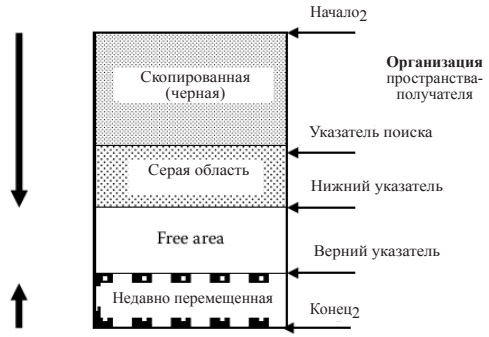


Рисунок 6.12 Пространство-получатель во время инкрементного копирующего сбора мусора.

N - местоположение активной памяти в пространстве-отправителе, а N/K является новой памятью, располагающейся в пространстве-получателе во время сбора мусора. Остающаяся память в пространстве-получателе представляет собой $M - (N + N/K)$. При условии стабильного состояния, при котором скорости размещения и удаления из памяти объектов равны, оставшаяся память в пространстве-получателе должна быть такой же, как и активная память, копируемая из пространства-отправителя. Таким образом, $N = M - (N + N/K)$, что ведет к $M = 2N + N/K$. Используя это уравнение, можно вывести размер множества, с учетом

размера активной области памяти и коэффициента *K*.

6.8 СБОР МУСОРА МЕТОДОМ НЕПРЕРЫВНОГО ПОДСЧЕТА ЧИСЛА ССЫЛОК

Еще одна популярная схема для сбора мусора - сбор мусора методом подсчета числа ссылок, которая является схемой непрерывного сбора мусора, решающей проблему зависших указателей при наличии совместно используемых структур данных. Как было описано ранее, часть структуры данных может совместно использоваться с другими структурами данных. Общая часть структуры данных не может быть утилизирована, за исключением всех совместно используемых структур данных, которые удалены логически. Сбор мусора методом подсчета числа ссылок решает эту проблему путем хранения числа ссылок в заголовке каждой ячейки данных. Счетчик ссылок увеличивается при создании новой ссылки и уменьшается после удаления ссылки. Ссылка может быть создана путем формирования новой структуры данных, которая имеет общую часть в структуре данных. Ссылку можно уничтожить путем удаления структуры данных.

После удаления структуры данных все ячейки данных из первого указателя (корневой указатель) удаленной структуры данных уменьшается на 1. Декрементирование счетчика ссылок на единицу обнуляет счетчик ссылок, если количество ссылок ранее равнялось единице. Все ячейки данных, имеющих количество ссылок = 0, собираются и связываются друг с другом в качестве свободного блока памяти. Процесс уменьшения количества ссылок останавливается после достижения счетчиком количества ссылок ячейки > 1, потому как данная структура используется по меньшей мере, еще одну структуру данных.

Пример 6.5

Рисунок 6.13 показывает два объекта данных, представленных в виде связанных списков. Объект данных 1 состоит из цепочки ячеек данных: #1, #2, #3, #6 и #7. Объект данных состоит из цепочки ячеек данных #4, #5, #6, и #7. Ячейки данных #6 и #7 являются совместно используемыми, и на ячейку данных #6 указывают два указателя: указатели из ячеек данных #3 и #5. Счетчик ссылок ячеек #1, #2, #3, #4, #5 и #7 равен 1, и счетчик ссылок ячейки # 6 равен 2.

Рассмотрим сценарий, когда объект данных 1 удален. Счетчик ссылок ячейки #1, #2 и #3 становится 0, а значение счетчика ссылок ячейки #6

становится равным 1. Три

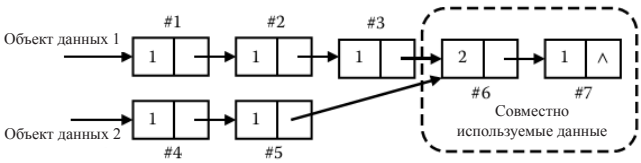


Рисунок 6.13 Счетчики ссылок для общих структур данных.

ячейки - ячейки #1, #2 и #3 собраны и утилизированы. Однако ячейки #6 и #7 имеют счетчик ссылок 1 и не собраны.

Существует много проблем с примитивным сбором мусора методом подсчета числа ссылок, таких как:

1. Сбор мусора методом подсчета числа ссылок не направлен на сжатие свободного пространства, так как является схемой непрерывного сбора мусора и таким образом страдает от фрагментации.
2. Сбор мусора методом подсчета числа ссылок не способен эффективно утилизировать циклические структуры данных. В случае с циклическими структурами данных, как это показано на Рисунке 6.14, счетчик ссылок первой ячейки равен 2, а не 1, как и в случае с ациклическими структурами данных. При удалении структуры данных счетчик ссылок уменьшается до 1 для первой ячейки данных и процесс декрементирования останавливается. Поскольку ни одна из ячеек не имеет счетчика ссылок = 0, высвободившиеся структуры данных не могут быть собраны, за исключением случаев, когда цикл не обнаружен. Обнаружение цикла влечет за собой значительные затраты памяти и времени выполнения программы и избегается в реальных сборщиках мусора. Явление невозможности утилизировать освобожденные ячейки памяти называется *утечкой памяти*, так как эти ячейки памяти никогда не смогут быть освобождены для других структур данных во время жизненного цикла программы. Поскольку опыт показывает, что лишь небольшой процент структур данных являются циклическими, утечка памяти позволяет сохранять эффективность выполнения сбора мусора.
3. Декрементирование счетчика ссылок большой структуры данных имеет значительные потери во времени выполнения программы. Чтобы уменьшить эти издержки, декрементирование счетчика ссылок откладывается до тех пор, пока есть запросы на выделение памяти. Вместо того, чтобы непрерывно декрементировать счетчик ссылок,

адрес первой ячейки освобожденной структуры данных помещается на вершину стека, называемого *стек обратного отсчета* (TBD). Когда все запросы на распределение областей памяти *m* выполнены, адрес извлекается из стека TBD, и области памяти «*m*» собираются из соответствующих освободившихся структур данных. Процесс сбора начинается, если после декрементирования, счетчик ссылок равняется 0 и заканчивается, когда счетчик ссылок после декрементирования больше или равен 1.

4. Хранение счетчика ссылок несет за собой излишние расходы памяти. Использование всего 4 бит для максимум 16 ссылок несет за собой значительные расходы памяти. Часто 4 бит недостаточно, кроме того, может появиться большее количество ссылок. Были предложены несколько вариантов сборщиков мусора методом подсчета числа ссылок для решения проблемы переполнения, вызванного чрезмерным количеством ссылок.

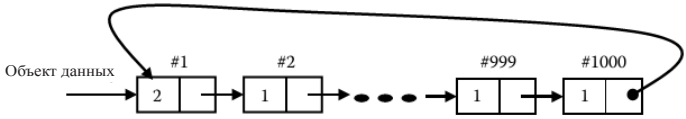


РИСУНОК 6.14 Проблема циклических структур данных в сборе мусора методом подсчета числа ссылок

6.9 КОНКУРЕНТНЫЙ СБОР МУСОРА

Инкрементный сбор мусора в значительной степени устраняет проблему временного прекращения выполнения программы. Однако инкрементный сбор мусора по-прежнему страдает от небольших приостановок программы. Конкурентный сбор мусора позволяет осуществлять более подробный сбор мусора путем запуска сборщика мусора и выполнения программы одновременно на двух отдельных процессорах или с использованием двух разных потоков на однопроцессорном компьютере. В то время как один процессор выполняет выполнение программы, второй процессор выполняет сбор мусора. Существует два вопроса в конкурентном сборе мусора: как обработать общие структуры данных, поскольку сборщик мусора (также именуемый коллектором) и выполнение программы (также именуемое мутатором) не может одновременно выполнять операции в одной ячейке

данных. Процесс освобождения или выделения ячейки данных является операцией с несколькими потоками команд. После запуска, операция должна быть обязательно завершена без передачи управления. Это свойство называется *атомарностью*, и операция называется *атомарной операцией*. Как атомарность, так и подробная синхронизация имеют решающее значение для конкурентного сбора мусора. Это ограничение накладывает дополнительные затраты в сравнении с последовательными схемами сбора мусора.

6.9.1 Конкурентный копирующий сбор мусора

Конкурентный копирующий сбор мусора позволяет копирование и сжатие живых объектов из пространства-отправителя в пространство-получатель без кокой-либо приостановки выполнения программы. Поток коллектора и поток мутатора продолжают выполняться одновременно. Общая схема показана на Рисунке 6.15.

Она использует две особые абстракции данных: *карта перемещений* и *журнал изменений данных*. *Карта перемещений* хранит записи форм (*область памяти пространства-отправителя*, *область памяти пространства-получателя*), чтобы отметить места в памяти, которые были скопированы из пространства-отправителя в пространство-получатель, а журнал изменений данных хранит все изменения, внесенные в объекты пространства-отправителя во время сбора мусора, которые должны быть включены в пространство-получатель перед переключением полупространств. В отличие от «старт-энд-стоп» копирующего сбора мусора, указатель следующей позиции не создается в пространстве-отправителе. Чтобы избежать копирования общей части структур данных используется карта перемещений. Карта перемещений внедряется путем использования хэш-таблицы с адресом в пространстве-отправителе в качестве ключевого. Каждая запись в карте перемещений эквивалентна неразрушающей версии

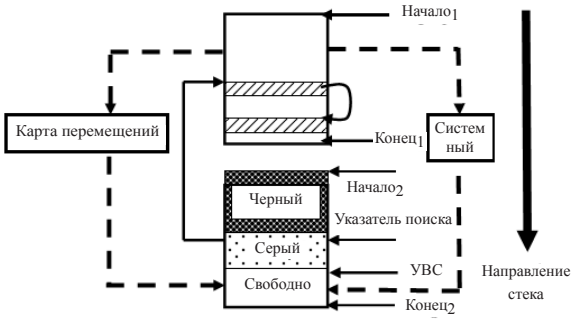


Рисунок 6.15 Конкурентный копирующий сбор

прямого указателя в инкрементном сборе мусора. Цель карты перемещений идентична назначению прямого указателя «старт-энд-стоп» копирующего сбора мусора. Перед копированием области памяти из пространства-отправителя в пространство-получатель, проверяется карта перемещений. Запись в карте перемещений указывает на то, что область памяти распределяется между двумя структурами данных, и область памяти не копируется.

Конкурентный сбор мусора не позволяет мутатору (выполнению программы) использовать повторяющиеся объекты в пространстве-получателе до тех пор, пока все живые объекты не будут полностью скопированы. Новые изменения, произведенные мутатором, собраны в *журнале изменений данных*. Сборщик продолжает проверять системный журнал и обновляет в пространстве-получателе соответствующий повторяющийся образ.

Общий алгоритм состоит в том, что в каждый момент времени из пространства-отправителя копируется одна структура данных в пространство-получатель, используя при этом модификацию Чейни. Когда область памяти копируется из пространства-отправителя в пространство-получатель, ввод данных (*адрес from-space, адрес to-space*) вводится в карте перемещений. Любое новое изменение пространства-отправителя во время сбора мусора также вносится в журнал. Исходя из предположения, что сборщик работает гораздо быстрее, чем мутатор, копирование живых объектов и обновления на основе журнала будут скопированы в пространство-получатель в реалистичные сроки.

Одним из основных вопросов в конкурентном сборе мусора является решение конфликта одновременной модификации мутатором структуры данных, которая в то же самое время копируется сборщиком. Используется

схема *атомарной операции* и структуры данных копируются по одной ячейке в один момент времени. Если необходимо немедленно передать управление программам операционной системы, то все незавершенные процессы и атомарные операции вновь возобновятся с самого начала, чтобы получить корректные результаты. Чтобы обеспечить атомарные операции, используются блокировки. Блокировки являются аппаратно поддерживаемой областью памяти специального назначения, которые обеспечивают исключительный контроль ресурсов компьютера над процессом. Контроль срабатывает только после сброса блокировки.

6.9.2 Конкурентный сбор мусора в режиме реального времени

Как описано в предыдущем разделе, конкурентный сбор мусора должен пройти через атомарные операции на уровне ячейки информации и синхронизацию, чтобы обеспечить правильное чередование мутатора и сборщика. Для конкурентного копирующего сбора мусора в режиме реального времени, были использованы два типа схем: (1) использование блокировок на уровне информационных ячеек для обеспечения атомарных операций на уровне ячеек и (2) не использующий блокировки сбор мусора, учитывающий, что частота сборки и выполняемых мутатором операций на одной и той же ячейке происходит редко. В случае столкновения мутатора и сборщика, копирование прерывается. Недопущение использования блокировок повышает производительность.

6.10 ПРОБЛЕМЫ В ИСПОЛЬЗОВАНИИ СБОРА МУСОРА

Сбор мусора требует определенного количества информации для успешной утилизации памяти. Ему необходимо некоторое количество информации о том, является ли ячейка памяти объектом данных или указателем. Аналогичным образом, каждая ячейка должна иметь битовую метку, чтобы пометить, является ли область памяти выделенной или освобожденной. Сбор мусора методом подсчета числа ссылок требует дополнительных битов памяти для поддержания счетчиков ссылок. Существуют значительные затраты времени в процессе выполнения сбора мусора. Они составляют около 20-30% от общего времени выполнения программы. Механизм сбора мусора «старт-энд-стоп» приостанавливает выполнение программы, что может привести к потерям событий в реальном времени и внезапному ухудшению характеристик, в особенности в интерактивных программах. Для поддержания вычислений в реальном масштабе времени и во избежание внезапного

ухудшения производительности программы, сбор освобождающейся памяти и выполнение программы чередуются между собой.

Инкрементный сбор мусора решает проблему долговременных приостановок. Однако все еще сохраняются вставки кратковременных приостановок, которые вызывают задержки в обработке событий жесткого реального времени. Для решения этой проблемы, сбор и выполнение программы осуществляются одновременно на многопроцессорных системах и многоядерных компьютерах. Одновременное выполнение в значительной степени решает проблему незначительных приостановок в традиционных инкрементных сборах мусора. Однако одновременное выполнение имеет свои собственные проблемы с обработкой: (1) *атомарные операции* представляют собой несколько потоков команд, рассматриваемых как единая большая команда; (2) обработка общих сегментов памяти, доступных как со стороны сборщика мусора, так и со стороны выполнения программы; и (3) обработка событий в системе жесткого реального времени. Для того, чтобы обработать события жесткого реального времени, современные системы позволяют приостановить процесс сбора мусора диспетчером задач в операционной системе, когда возникает необходимость обработать события в реальном времени.

Процесс сбора мусора начинается с обхода ячеек данных, запускаясь с корневого указателя. Смысл состоит в том, чтобы определить указатель, так как числа и адреса являются комбинациями нулей и единиц. Многие языки программирования, такие как C дают указателю арифметически обрабатываемые адреса как целочисленные переменные для перемещения по сегментам памяти. Это очень затрудняет разделение между целыми числами и указателями. Целые числа представляют собой данные, а указатели – адреса других ячеек данных, которые должны быть собраны после того, как структуры данных освобождены. Как правило, указатели можно определить, зная структуру ячейки данных. Однако, если указатель пропущен, и во время сборки мусора рассматривается как некое количество данных, то следующие освобожденные ячейки данных являются недоступными, не помечены как живые, и собираются как ячейки мусора, вызывая неправильную реакцию программы. Если ячейка данных рассматривается как указатель, тогда ячейки мусора могут быть помечены как живые ячейки и не могут быть собраны во время сбора мусора, что служит причиной тому, что эта ячейка не будет использована в будущем, что и приводит к *утечке памяти*. Утечкой памяти является ситуация, когда ячейка освобожденных данных недоступна для

перераспределения. Сборщики мусора достаточно консервативны и предпочитают утечки памяти некорректному поведению программы. Предпринимаются многие другие меры предосторожности для выделения различий между указателями и данными. Например, любая ячейка данных, которая имеет значение, не указывающее на область множества рассматривается как ячейка данных, а ячейки данных выровнены по границе слов и инициируется, как нулевое значение, чтобы отделить их от указателей. Еще одним важным вопросом является распределение данных в множестве или в стеке. Эти динамические данные, время жизни которых ограничено областью вызываемой процедуры принято выделять в стек для

более быстрого доступа и восстановления памяти.

6.11 КРАТКИЕ ВЫВОДЫ

В этой главе мы обсудили управление памятью динамических данных структур и повторения рекурсивных данных с помощью множества. Множество — это общая область, видимая для всех подпрограмм, и все подпрограммы могут выделить динамические данные в множество. Это множество отличается от множества, которые вы изучали в структурах данных. Свободная область памяти множества может быть организована несколькими путями: (1) с помощью одной цепочки указателей блоков свободной памяти; (2) использование связанной группы свободных блоков, отсортированных по аналогичным размерам; и (3) сбор стеков. Данные могут быть выделены автоматически либо при создании или программном запросе нового динамического объекта.

Стратегия распределения может быть *первым пригодным*, *следующим пригодным* или *наилучшим*. Ячейки памяти множества находятся в трех состояниях: *активное, освобожденное* и *свободное*. Сборка мусора утилизирует память из *освобожденного состояния* в *свободное состояние*. Платформа динамических данных ячейки может быть освобождена автоматически или программно. В процессе непрерывного выделения и освобождения свободное пространство продолжает получать перемежающиеся изолированные мелкие куски памяти. Многие из блоков памяти настолько малы, что они не могут самодостаточно использоваться для выделения эффективной памяти. Такое формирование перемежающихся малых блоков называется *фрагментацией*, и это отрицательно сказывается на времени выполнения работы сборщиков мусора и выполнении программы путем уменьшения коэффициента попадания в кэш и увеличивая частоты сбоев страниц и

сборки мусора.

Сборка мусора может происходить непрерывно, как в счетчике ссылок сборки мусора, или периодически, после того, как больше отрезки памяти не могу быть выделены в пространство множества. Периодические сборщики мусора могут быть «старт-энд-стоп» или возрастающими. «Старт-энд-стоп» сборщики мусора могут приостановить выполнение программы полностью во время сборки мусора, и непригодны для обработки событий реального времени, так как сборка мусора вызывает значительную задержку из-за излишних затрат памяти и времени выполнения. Чтобы избежать этой проблемы, были опробованы многие подходы, такие как инкрементный, параллельный, а также непрерывный сбора мусора. В инкрементном сборе мусора один большой период сбора мусора делится на несколько небольших периодов, чередующихся с небольшим периодом выполнения программы, таким образом, уровень сбора быстрее, чем скорость выделения памяти. Параллельная сборка мусора запускает выполнение программы и сборку мусора одновременно. Однако они должны решать вопросы атомарности нескольких инструкций, участвующих в общих атомарных операциях и обеспечивать синхронизацию при совместном использовании памяти между сборщиком мусора и выполнением программы. Схемы инкрементного и параллельного сбора мусора призваны смягчить проблему программ реального времени, таких как показ видео, плавным снижением производительности. Однако они не в состоянии решить проблему событий жесткого реального времени, связанных с датчиками или критическими операциями. Для того, чтобы регулировать события жёсткого реального времени, планировщик процесса в операционной системе должен отложить и приостановить процесс сбора мусора, когда происходят события с высоким приоритетом в реальном времени.

Периодические сборщики мусора работают двумя способами: (1) маркировка активного пространства и сборка оставшегося места множества, как свободной памяти и (2) сбор активной памяти и копирование его в другую область памяти для устранения фрагментации. Маркировка активных пространств страдает множеством недостатков, таких как переполнение стека, излишние расходы на обход активного пространства дважды и фрагментации. Копирование сборки мусора страдает от потери памяти. Для того, чтобы решить эту проблему, используются несколько версий пространств. Каждая версия пространства имеет возраст, связанный с ним. Как только одна версия пространства становится неактивной, все другие версии пространства

активируются. Благодаря принципу локальности, старые структуры данных расположены в старых версиях пространства, и молодые структуры данных находятся в молодых версиях пространства. Более молодая версия пространства заполнится быстрее старой версии пространства. Увеличение числа версий пространства также позволяет избежать чрезмерного копирования старых, менее доступных структур данных. Недостаток версий пространств — излишние расходы на содержание таблицы прямых и обратных указателей для подключения структур данных, перемежающихся через несколько версий пространства.

Подсчет ссылок сборки мусора является методом непрерывного сбора мусора, который полезен в обработке структуры данных, так как сохраняет счетчик ссылок. Когда структура данных освобождается, подсчет ссылок соответствующих ячеек уменьшается, и все ячейки данных, с количеством ссылок = 0, собираются. Она сталкивается с проблемой обработки циклических структур данных, которая составляет 2-3% от общего объема. Для поддержания эффективности, счетчик ссылок мусора собирает схемы, предпочитая *утечки памяти* затратному сбору мусора циклических структур данных.

В механизмах сбора мусора в режиме реального времени, наименьшая единица собирания является одной единицей информации или объектом одновременно. Это можно сделать с помощью атомарных операций. Сборщик использует блокировку для копирования одного узла сведений или одного объекта за раз.

Существует много проблем со сборкой мусора. Порой трудно провести различия между ячейками данных и указателями. Если указатели рассматриваются как ячейки данных во время сборки мусора, то следующие ячейки памяти не отмечаются, в результате неправильного сбора активных ячеек, как мусора. Если ячейки данных трактуются как указатели, то дополнительные ячейки помечаются как активные и никогда не собираются, что приводит к *утечке памяти*. Программы сбора мусора используют консервативный подход и предпочитают утечки памяти в случае возникновения сомнений. Эмпирическое исследование показывает, что 2-3% памяти теряется на утечке памяти.

Есть много вариаций категорий сборщиков мусора, описанных в этой главе. В последние годы было приложено огромное количество усилий в разработке параллельных версий сборщиков мусора для различных параллельных архитектур. Есть также рассредоточенные версии сборщиков мусора для интернет-языков.

«Java» использует параллельную версию сборки мусора в ее

реализации. Различные реализации C++ используют вариации сборки мусора через разметку и сканирование, а также копирование сборки мусора. C# использует сборку мусора по поколениям с тремя версиями пространства. Scala построен на «Java» и использует сборщик мусора «JVM». «ADA» не сборщик мусора по умолчанию. «Haskell» использует параллельный сбор мусора по поколениям. Текущие реализации Ruby, используют вариант сборки мусора разметки-и-сканирования.

6.12 ОЦЕНКА

6.12.1 Понятия и определения

Блоки активной памяти; обратный указатель; алгоритм Бейкера; наилучшее распределение; черные ячейки; алгоритм Чейни; параллельный сбор мусора; одновременное копирование сбора мусора; непрерывный сбор мусора; копирование сбора мусора; запись таблицы; первое пригодное размещение; прямой указатель; фрагментация; свободный блок памяти; пространство-отправитель; сбор мусора; сбор мусора по поколениям; серая ячейка; сбор мусора в режиме жесткого реального времени; множество; неактивное пространство; инкрементная сборка мусора; маркировка бит; алгоритм разметки и сканирования; выделение памяти; освобождение памяти; журнал мутации сборка мусора в режиме реального времени; счетчик ссылок мусора; карта перемещения; «скан» указатель; полупространства; TBD-стек; пространство-получатель ; версия пространства; белые ячейки.

Решение Задач

1. Рассмотрим организацию индексированной цепочки свободного пространства, где каждый диапазон памяти составляет 1000 байт. Например, диапазоны 0-999, 1000-1999, 2000-2999, 3000-3999, 4000-4999 и так далее. Отображение организации индексированного множества с размерами свободного пространства 1024, 512, 1024, 2048, 1536, 4096, 32, 16 и 128. Производится запрос на выделение памяти для двух блоков размером 480 байт и 1000 байт. Покажите организацию индексированного множества для множества после выделения запрошенных блоков с помощью алгоритма распределения, который собирает первые свободные блоки в цепочке свободных блоков в диапазоне лучшего соответствия.

2. Опишите представление структуры данных для индексируемой организации свободного пространства множества и напишите алгоритм высокого уровня для выделения запрашиваемых данных с использованием

индексированной цепочки для оптимального распределения. Наглядно покажите, как корректируется таблица после выделения.

3. Множество имеет следующие свободные блоки: 1200, 3000, 340, 560, 790 и 4002 байт. Предположим, что блоки соединяются, используя одну цепочку указателей. Выделите три запроса блока памяти на 2000, 450 и 740, используя стратегии первого подходящего и следующего подходящего и показывая оставшиеся свободные блоки после каждого выделения.

4. Напишите алгоритм высокого уровня для простейшего алгоритма разметки и сканирования.

5. Напишите алгоритм высокого уровня для простейшего старт-стопного копирования сбора мусора.

6. Напишите алгоритм высокого уровня для копирования сборки мусора по Чейни.

7. Напишите алгоритм высокого уровня для алгоритма инкрементного копирования с улучшением Бейкера.

8. В алгоритмах Бейкера предположим, что размер полупространства составляет 200 КБ, а соотношение собранных ячеек к выделенным ячейкам во время сбора мусора = 8. Каким будет количество ячеек памяти, выделенных в процессе сбора мусора, если предположить, что процент живых объектов данных в полупространстве составляет 50% на момент начала сборки мусора?

9. Существует два объекта данных: объект данных$_1$ (размером 2000 байт), на который указывает указатель P, и объект данных$_2$ (размером 1000 байт), на который указывает указатель Q, эти объекты совместно используют 400 байт общего пространства. Если предположить, что удаление использует стек TBD, покажите записи в TBD стек для следующих действий: удалить(P), удалить(Q), новый(A1), новый(A2), новый(A3). Где A1 — это объект, который требует 200 байт, A2 требует 60 байт и A3 требует 1440 байт. Объясните ваш ответ, используя рисунок и различные снимки TBD стека.

6.12.3 Концептуальный тип

Какие проблемы стартстопного сбора мусора? Объясните.

10. Каковы проблемы наивного алгоритма разметки и сканирования? Как они могут быть исправлены?

11. Объясните проблемы с наивными копированием сбора мусора. Как они могут быть исправлены?

12. Объясните проблемы со счетчиком ссылок сборки мусора. Как они могут быть исправлены?

13. Объясните подробно улучшение Чейни традиционного стартстопного копирования сборки мусора с четким объяснением роли поиска в ширину и скан-указателя в уменьшении затрат памяти на стек.

14. Объясните роль прямых и обратных указателей в таблицах указателя, хранящихся в сборке мусора по поколениям .

15. Объясните улучшение Бейкера в алгоритме Чейни для инкрементных сборщиков мусора.

16. Каковы различные шаги, предпринятые для обработки событий в реальном времени во время сборки мусора? Объясните.

17. Объясните различия между алгоритмом Бейкера и одновременным копированием сборки мусора.

18. Как возраст и таблица указателей влияют на издержки в процессе сборки мусора по поколениям?

Приложение I

Поддерживаемые в языках парадигмы

C^T — параллельный задаче поток; **C^P** — параллелизм данных; **C^D** — распределен; **C^S** — синхронный параллелизм; **E** — на основе событий ; **F** — функциональный; **I** — обязательный; **L** — логика; **M** — мультимедиа; **O^I** — объектно ориентированный с наследованием класса; **O^F**— объектно-ориентированный с плоскими объектами; **V** — визуальный; **W** — веб.

Примечание: Некоторые языки имеют не четкую стандартную версию. Некоторые даты являются приблизительными, на основе последней разработки компиляторов.

Язык	Редакция	Парадигмы
Algol W	1996	**I**
ALICE	2000	**I, E, M, V**, образовательный мультимедийный язык
Ada	2012	**I,C^T, O^I**
C	1999	**I,C^T**(библиотека), **C^P**(вариации)
C++	2011	**I, C^T** (библиотека), **O^I, V (**вариации**)**
C#	2010	**I, E, C^T, M, O^I, V, W** (с .NET**)**
Chapel	Эволюция	**I, C^P,O^I (** для массивных параллельных вычислений**)**
Claire	2009	**I, F, L, O^I, W**

Clojure	2011	**F, C^{T}**, язык сценариев
COBOL	2002	**I, O^{I}**, бизнес язык программирования
ECL^{i}PS^{e}	1997	**L**, язык программирования с ограничительной логикой
Emerald	1994	**I, C^{D}, O^{F}**, язык распределенных вычислений
" Estrel "	1991	**I, C^{S}**, язык моделирования синхронного оборудования
F #	2005	**I, F, O^{I}, E, V, W**
FORTRAN	2008	**I, O^{I}, C^{P}** (различные варианты, например HPF)
Haskell	2010	**F, C^{T}**
Java	Продолжается	**I, E, O^{I}, C^{T}, M, V, W**,
JavaScript	1995	**I, E, O^{F}, W**, язык сетевых сценариев
Семейство Lisp	1994	**I** (ограниченная)**, F** (основной)**, O^{I} (**вариации), **C** (ограниченная площадь)
Lua	2006	**I, F, O, C^{T}, W** (язык сценариев игры)
ML	1998	**I** (ограниченная поддержка), **F (**основной**)**
Modula-3	1991	**I, O^{I}, C^{T}**
Oz/ Mozart	2007	**I, F, L, O^{I}, C^{T}**
Perl	Варианты	**I, F** (ограниченная), **O^{I}**, язык сценариев
PhP	2004	I, OI, W, язык сетевых сценариев
Prolog	Различия	**L, O^{I} (**библиотека**), V (**вариации), **C^{T}** (вариации)
Python	Продолжается	**I, F, O^{I}**, регулярный язык используется для работы со скриптами
Ruby	2012	**I, F, O^{I}**, интегрированная мультипарадигма
Scala/ EScala	2012	**I, F, O^{I}**, **C^{T}**, интегрированная мультипарадигма
SMIL	2008	**I, M, W (**для веб вычислений**)**
VRML/ X3D	1996/2005	**I, M, W**, языки 3D моделирования
X10	Развивается	**I, C^{P},O^{I}** (для массивных параллельных вычислений)
на основе XML/ HTML	Вариации	**M, V, W**, для сетевого вычисления

Приложение II

Сводка абстрактных данных

1. Изменяемые объекты данных
2. Неизменяемые объекты данных
3. Независимые явные указатели/ссылки
4. Основные типы дополнительной точности
5. Строки
6. Массивы/индексированные последовательности
7. Хэш-таблица/карты/ таблица ключевых значений
8. Именованные кортежи/запись/структура
9. Порядковый номер типа
10. Наборы
11. Рекурсивные типы данных/списки
12. Универсальный полиморфизм/подтип
13. Объекты
14. Класс и наследование (одно- или множественное наследование)
15. Модули/пакет/пространства имен

■ — Встроенный в/встроенный в библиотеке; **A**— псевдонимы, используемые как ссылки; **Δ** — ограниченный; **D** — с динамическим контролем типов; **L** — через библиотеку; **M** — медиа-объекты; **R** — ссылка вместо указателя; **P** — диапазон, **S** — моделируется другой структурой данных; V — вариации языковой поддержки; **X** — отсутствует

Языки	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
АЛИСА	■	X	X	X	■	■	X	X	X	X	■	X	M	■	X
ADA 2005	■	Δ	■	■	■	■	■	■	■	■	■	■	■	■	■
C	■	X	■	■	■	■	X	■	■	S	■	X	X	X	■
C++	■	X	■	■	■	■	■	■	■	■	■	■	■	■	■
C#	■	X	■	Δ	■	■	■	■	■	S	■	■	■	■	W
Часовня	■	X	X	■	■	■	■	■	■	■	X	■	■	■	■
Клэр	■	■	X	X	■	■	X	■	п	■	■	■	■	■	■
Clojure	Л	■	P	Л	■	■	■	■	п	■	■	■	■	■	■
ECL^{i}PS^{e}	X	■	■	D	■	■	■	■	S	■	■	■	■	■	■
Изумруд	■	■	X	X	■	■	X	■	■	■	■	■	■	X	■
"Эстрель"	■	X	X	X	X	X	X	X	X	X	X	X	X	X	■
F #	■	■	P	■	■	■	■	■	■	■	■	■	■	■	■
Фортран 2008	■	X	■	■	■	■	X	X	■	X	■	■	■	■	■
Haskell	X	■	X	■	■	■	S	■	■	■	■	■	■	■	■
Ява	■	X	P	■	■	■	■	X	■	■	S	■	■	■	■
JavaScript	■	Δ	X	D	■	■	■	X	S	X	X	X	■	X	X
Лисп	■	■	X	D	■	■	■	S	B	S	■	■	B	B	X
Lua	■	■	X	X	■	■	■	X	S	S	■	B	■	S	■
ML	■	■	■	■	■	■	■	■	■	■	■	■	X	X	■
Модула-3	■	X	■	■	■	■		■	■	■	■	■	■	■	■
Perl	■	X	Л	Л	■	■	■	S	■	S	S	■	■	■	■
PhP	■	X	X	X	■	■	■	X	X	X	X	Л	■	■	■
Python	■	■	A	■	■	■	■	■	■	■	■	■	■	■	■
Рубин	■	■	P	D	■	■	■	X	п	■	п	■	■	■	■
Скала	■	■	X	■	■	■	■	■	■	■	■	■	■	■	■
СМИЛ	■	X	X	X	■	X	X	X	X	X	X	X	M	X	■
X10	■	■	X	■	■	■	X	■	X	Л	■	■	■	■	■

Приложение III

Сводка абстракций управления

1. Деструктивное (разрушающее) назначение
2. Множественное/параллельное назначение
3. Условный оператор (if-then-else)
4. Операторы case/switch (оператор выбора/оператор переключения)
5. Множественное/параллельное назначение
6. Условный оператор (if-then-else)
7. Операторы case/switch (оператор выбора/оператор переключения)
8. Для цикла/определенная итерация
9. Во время цикла/бесконечный цикл
10. Цикл с условием продолжения (повторять до выполнения условия)
11. Итераторы
12. Функции/подпрограммы
13. Охранники
14. Обработка исключений
15. Пользовательские потоки/задачи

16. Отслеживание/взаимное исключение использование блокировок/ защищенные объекты/синхронизированные методы

17. Другие конструкции параллельности

18. Распределённые вычисления

■ — встроенная поддержка; **C** — кластерные вычисления; D — общий цикл и выход; **L** — сопряжен с библиотекой; **L**$^{\mathbf{T}}$ — Временная петля; **M** — внешняя библиотека передачи сообщений, как PVM или MPI; **N** — не является частью текущей спецификации; **R** — смоделирован с использованием концевой рекурсии; **S** — можно моделировать с помощью существующих конструкций; **T** — как особенность, характерная черта; **U** — «пока-не», цикл с условием завершения; **V** — поддерживает изменение языка; **W** — поддерживает веб-программирование; **X** — не поддерживается в стандартах

	Управляемые абстракции														
Язык	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
АЛИСА	■	X	■	X	■	■	X	X	■	X	X	■	X	■	X
Ада	■	■	■	■	■	■	■	■	■	■	■	■	■	■	B
C	■	X	■	■	■	■	■	X	■	X	X	Л	Л	B	M
C++	■	■	■	■	■	■	■	■	■	X	■	Л	Л	X	X
C#	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
Часовня	■	X	■	■	■	■	■	■	■	X	N	■	■	■	C
Клэр	■	X	■	■	■	■	Ю	■	■	X	■	X	X	■	X
Clojure	■	■	■	■	■	■	X	■	■	X	■	■	■	■	■
ECL$^{\text{i}}$PS$^{\text{e}}$	■	■	■	X	■	P	P	■	■	X	Δ	Δ	■	X	■
Изумруд	■	X	■	X	■	■	D	X	■	X	X	■	■	X	■
"Эстрель"	■	X	■	■	■	■	■	X	■	X	■	X	X	■	X
F# 3.0	■	X	■	■	■	■		■	■	X	■	■	■	■	W
Фортран 2008	■	X	■	■	D	D	D	X	■	X	Δ	B	B	■	M
Haskell	X	X	■	■	■	S	S	S	■	■	■	■	■	■	■
Ява	■	Δ	■	■	■	■	■	■	■	X	■	■	■	X	■
JavaScript	■	■	■	■	■	■	■	■	■	X	■	X	X	X	W
Лисп	■	■	■	■	■	■	■	■	■	X	■	X	X	■	X
Lua	■	■	■	X	■	■	■	■	■	X	■	Л	Л	■	W
ML	■	■	■	■	S	■	X	■	■	B	■	■	■	■	■
Модула-3	■	X	■	■	■	■	■	X	■	X	■	■	■	X	■
Perl	■	■	■	■	■	■	■	■	■	X	■	■	X	■	B
PhP	■	■	■	■	■	■	■	■	■	X	■	X	X	X	W
Python	■	■	■	■	■	■	X	■	■	X	■	■	■	X	■
Рубин	■	■	■	■	■	■	Ю	■	■	X	■	■	■	X	■
Скала	■	■	■	■	■	■	X	T	■	■	■	■	■	■	W
СМИЛ	■	X	X	X	■	X	X	X	X	X	X	X	X	■	W
X10	■	X	■	■	■	■	■	X	■	Л	■	■	■	■	C

Приложение IV

Веб-сайты для языков

Язык	Веб-сайты
Алиса	http://www.alice.org/
Ада	http://www.sigada.org/education/ and http://www.ada-auth.org/arm.html C Несколько доступных в Интернете
C++	Несколько доступных в Интернете, включая Microsoft Visual Studio C# Несколько доступных в Интернете, включая Microsoft Visual Studio Chapel http://chapel. cray.com/
Клэр	http://www.claire-language.com/ Clojure http://clojure.org/
ECLiPSe	http://www.eclipseclp.org/MT
Изумруд	http://www.emeraldprogramminglanguage.org Esterel http://www-sop.inria.fr/meije/esterel/esterel-eng.html
F #	http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/ http://msdn.microsoft.com/en-us/library/dd233154.aspx
ФОРТРАН	http://j3-fortran.org/ Haskell http://www.haskell.org Java http://www. java.com and http://www.oracle.com/us/technologies/java/overview/index.html
JavaScript	http://msdn.microsoft.com/en-us/library/ie/d1et7k7c(v = vs.94).aspx Lisp Несколько включая сайт GNU Lisp
Lua	http://www.lua.org
ML	http://www.smlnj.org Modula-3 http://www.modula3.org
Perl	Несколько доступных в Интернете
PhP	http://www.php.net/
Пролог	http://www.sics.se/software; http://www.gprolog.org; http://www.swi-Prolog.org Python http://www.python.org/
Рубин	http://www.ruby-lang.org
Скала	http://www.scala-lang.org
СМИЛ	http://www.w3.org/AudioVideo/ and http://www.w3.org/TR/2008/REC-SMIL3-20081201/
X10	http://x10-lang.org/

Принцип локальности

Программа выполняется в пределах ограниченного подмножества локальной среды в течение небольшого промежутка времени. Эта, в настоящее время активная часть локальной окружающей среды, называется *участок*. Принцип локальности утверждает, что программа будет использовать *тот же участок* в ближайшем будущем. Локальность изменяется потоком управления: (1) обрабатывает различные части большой структуры данных, (2) перемещается в разные части большой программы или (3) вызывает другую подпрограмму.

Принцип локальности имеет важное значение для эффективного выполнения и более эффективного использования пространства выполнения программ, поскольку современные операционные системы выполняют несколько процессов одновременно — активная часть программы — и выделение ограниченного объема оперативной памяти для каждого активного процесса. Из-за ограниченности места в оперативной памяти только малая часть программы вносится в активную память, и эта активная память должна управляться непрерывно, чтобы избежать бездействия блоков памяти большой структуры данных. Когда участок выполнения изменяется, новые части пространства данных вносятся в выделенное пользователем пространство, вызывая проблемы донесения информации от более медленных устройств вторичного хранения, таких как жесткие диски.

Виртуальная память и ошибки страниц

Приложение VI

Виртуальная память и ошибки из-за отсутствия страниц

Программист пишет программу логического пространства, предполагая, что все пространство данных и пространство кода непрерывно. Однако, в действительности, на низком уровне, программы и данные хранятся в оперативной памяти и на жестких дисках: активная часть программы хранится в оперативной памяти, в то время как остальные части программы и файлы данных хранятся на жестком диске. Существует два пространства памяти: логически смежное, по мнению программиста, пространство памяти, называемое *виртуальной памятью* и разбросанное физическое хранилище, где фактически будут сохранены программа и данные. Виртуальная память отображается в физической памяти операционной системы. Блок памяти может быть переменного или фиксированного размера, в зависимости от операционной системы. Блоки переменного размера называются *сегментами* и имеют размер вызываемой подпрограммы. Блоки фиксированного размера называются *страницами*.

Страница (или сегмент) - вносится из вторичного хранилища в оперативную память, если адрес доступа оперативной памяти в виртуальную память отсутствует в оперативной памяти. Процесс импорта недостающих страниц с жестких дисков в оперативную память (RAM) называется *ошибкой страницы* и вызывает значительные затраты на передачу данных. По мере увеличения числа сбоев страниц, фактическое использование ЦП уменьшается. Планировщик процессов добавляет больше процессов, для улучшения использования ЦП. Выполнение нескольких процессов вызывает больше ошибок страниц

и еще меньшее использование центрального процессора. В конечном счете, центральный процессор прекращает делать любую полезную работу. Это явление называется пробуксовка, и его следует избегать.

Приложение VII

Корректность и полнота работы программы

Программа *правильна*, если набор сгенерированных решений является подмножеством всех возможных решений проблемы. Программа *завершена*, если множество всех возможных решений проблемы входит в набор решений. Существует возможность неправильного решения в программе, которое заботится только о *завершенности* программы. Аналогичным образом, существует возможность того, что программа окажется неполной, если программа пытается принять только правильные решения. Мы должны искать правильность и полноту решений, когда мы пишем программу.

В реальной жизни трудно написать большое программное обеспечение, которое является правильным и полным. Большинство крупных программ не являются ни правильными, ни полными. Программное обеспечение имеет множество логических ошибок, называемые *ошибками программного обеспечения*. Число этих ошибок значительно возрастает с увеличением размера программного обеспечения.

Существует несколько способов удаления ошибок программного обеспечения: (1) программист проходит через логику снова и снова (2) программист запускает программу с образцами различных входных данных с помощью отладчиков для обнаружения и исправления ошибок каждый раз, когда программа не выдает нужное выходное значение и (3) использует средства для автоматизированного анализа программ для получения правильности программы. Методы 1 и 2 являются довольно распространенными. Однако метод 3 это редкость для большого программного обеспечения, так как требует вовлечения сложных вычислений.

Приложение VIII

Сложность алгоритмов

Во время перевода высокоуровневых конструкций в промежуточный код низкого уровня и во время выполнения программ должна решаться проблема эффективности программ. Эффективность программ зависит от многих факторов, таких как: (1) расположение структур данных; (2) Частота получения данных, хранящихся на жестком диске; (3) отсутствие эффективных алгоритмов; и (4) затраты на передачу данных, особенно при использовании нескольких процессоров для решения сложной задачи.

Одним из способов изучения эффективности заключается в изучении того, работают ли основные алгоритмы эффективно, когда размер входных данных увеличивается. Например в алгоритме линейной сложности, время выполнения пропорционально увеличивается с увеличением размера входных данных. Алгоритм имеет *квадратичную сложность*, если время выполнения программы увеличивается на коэффициент $O(m^2)$ при увеличении размера входных данных на коэффициент m. В *кубических* алгоритмах время выполнения увеличивается на $O(m^3)$, когда размер входных данных увеличивается в m раз. Алгоритмы с коэффициентом, превышающим кубический, не предназначены для использования в данных большого размера и быстро становятся слишком медленными для выполнения своих задач. Такие алгоритмы, где коэффициенты мощности ограничены такой константой как верхняя граница, называются *Полиномиальными алгоритмами*, и на практике значение коэффициента мощности > 4 не считается хорошим решением. К классам эффективных алгоритмов относятся алгоритмы *постоянной сложности,* сложности log(log(N)), сложности log(N) и линейной сложности *(O(N))*, так как темпы увеличения времени выполнения меньше или равны размеру входных данных. Некоторыми примерами сложности log(N) являются алгоритмы двоичного поиска, а примерами

алгоритмов *постоянного времени* - методы хэширования.

Интересный класс алгоритмов представляют *экспоненциальные алгоритмы*, в которых время увеличивается экспоненциально с увеличением размера входных данных. При увеличении размера входных данных, экспоненциальные алгоритмы быстро достигают уровня, где время выполнения слишком велико для выполнения на любом существующем компьютере. Если мы не знаем, имеет ли данная проблема решение известной полиномиальной сложности и класс таких НП проблем может быть преобразован к этой проблеме, то проблема называется НП -*полная* (недетерминированно полиномиально полная). В реализации языков программирования мы хотим избежать НП-полная проблем поскольку их известные реализации являются неэффективными. В настоящее время НП-полные алгоритмы используют аппроксимацию, эвристики (интеллектуальное угадывание на основе математических функций или известных решений в прошлом) и ограничение входных данных и параметров для повышения эффективности выполнения алгоритмов.

Вопрос эффективности времени выполнения является довольно сложным и не может быть просто смоделирована исследованиями алгоритмической сложности. Есть другие вопросы для рассмотрения: (1) расходы на извлечение данных с жестких дисков в оперативную память, (2) использование принципа локальности , (3) превышение пропускной способности шин и (4) затраты на передачу данных. Эти вопросы рассматриваются в курсе «*Операционные системы*»

Содержание