САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МАТЕМАТИКО-МЕХАНИЧЕСКИЙ ФАКУЛЬТЕТ

И. Д. МИРОШНИЧЕНКО

КОНСПЕКТ ЛЕКЦИЙ «ПРИНЦИПЫ ПОСТРОЕНИЯ СОВРЕМЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ»

СПбГУ
2023

Учебное пособие предназначено для обучающихся 2-го курса образовательной программы «Прикладная математика и информатика» дисциплины «Системное программирование». В нем рассматриваются основные теоретические понятия и алгоритмы, относящиеся к технологиям разработки трансляторов, в частности, подробно разбираются особенности всех классических этапов трансляции: анализа проходов транслятора, оптимизации и генерации кодов; рассматриваются классические и современные технологии разработки системных программ: общие принципы разработки и современные концепции.

Авторы: Мирошниченко И. Д. – ст. преподаватель СПбГУ,

Рецензенты: д. ф.-м. наук, профессор Рябов В. М.
д. ф.-м. наук, профессор Аксенова О. А.

ПРЕДИСЛОВИЕ

Учебное пособие «Конспект лекций «Принципы построения современных вычислительных систем» представляет учебный материал курса “Системное программирование” образовательной программы «Прикладная математика и информатика».

В современных информационных технологиях системное программное обеспечение занимает важнейшее место. Системное программное обеспечение можно рассматривать как в узком смысле (имея в виду создание системных продуктов программирования), так и в широком (разработка любых прикладных программных продуктов и комплексов не обходится без применения принципов системного программирования и, собственно, её системной части). Область применения системного программного обеспечения сегодня чрезвычайно широка (для этого достаточно оценить диапазон интересов Института системного программирования имени В. П. Иванникова Российской академии наук, представленного на официальном сайте организации *http://www3.ispras.ru/ru/index.php, http://www3.ispras.ru/ru/directions.php*).

Настоящий конспект лекций как учебное пособие, во-первых, касается описания собственно процесса трансляции, методов реализации её этапов, во-вторых, посвящено обзору и анализу технологий разработки современных систем программного обеспечения.

Значимую часть технологий системного программного обеспечения составляют синтаксические методы. Трансляторы языков программирования (компиляторы, интерпретаторы, конверторы), синтаксические редакторы, автоматический перевод, средства обработки текстовой информации базируются на использовании методов синтаксического анализа. Теоретический фундамент этих методов составляют теория формальных языков и грамматик, теория конечных автоматов, которые рассматриваются в лекциях курса.

Настоящее учебное пособие предназначено для систематизации знаний в процессе изучения подходов к разработке системного программного обеспечения, сделает понятным принципы и методы построения современных вычислительных комплексов, в том числе суперкомпьютерных, базирующихся как на общей памяти, так и на распределенной. Изучение особенностей архитектуры и структуры функционирования суперкомпьютерных вычислительных систем и организации программного обеспечения для параллельного программирования в настоящий момент весьма актуально.

Знание классических методов обработки текстовой информации, как для последовательных, так и параллельных программ, позволяет упростить и процесс разработки программного обеспечения специализированных устройств, работающих с цифровыми сигналами.

Изучение способов передачи данных для многопотоковых задач, разнообразия архитектур вычислительных систем, особенностей преобразования последовательных программ в параллельные, разнообразия параллельных языков — все это весьма важно для того, чтобы ориентироваться в современных системах программирования.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ

Учебное пособие «Конспект лекций «Принципы построения современных вычислительных систем» предназначено для обучающихся образовательной программы «Прикладная математика и информатика» курса «Системное программирование», но также может быть использовано преподавателями в качестве рабочего материала.

Учебное пособие сформировано таким образом, что логически и содержательно-методически соответствует учебной программе по дисциплине «Системное программирование», которая читается в 4 семестре периода обучения. Предлагаемый материал шире и подробнее, чем читается на занятиях, предлагается для более глубокого изучения обучающимися. Методические установки ориентированы, в первую очередь, на то, чтобы познакомить обучающихся с методами реализации этапов трансляции, особенностями оптимизации и генерации кодов, во-вторую, — дать представление о классических и современных технологиях разработки системного программного обеспечения; концепциях, подходах и принципах разработки современных суперкомпьютерных комплексов и системного обеспечения для них, а также принципах разработки мобильных приложений для современных мобильных устройств.

Для более глубокого ознакомления интересующихся архитектурой и программным обеспечением современных суперкомпьютерных комплексов и мобильных устройств, концепциями и методами разработки их системного программного обеспечения приводится список дополнительной литературы

Теоретический материал излагается достаточно подробно, так, чтобы обучающиеся могли самостоятельно разобраться в изучаемой теме.

Первые 15 глав посвящены основным понятиям процесса трансляции, методам их реализации, особенностям оптимизации и генерации кодов, представлению данных на каждом из этапов, глава 15 — современным концепциям разработки трансляторов для новых и модернизируемых платформ.

Главы 16-21 посвящены знакомству с основными понятиями параллельных архитектур и парадигмам и параллельного программирования, с принципами и направлениями разработки, с одной стороны, современных суперкомпьютерных комплексов и высокоточных вычислительных систем для моделирования сложных реальных ситуаций и работы с большим объёмом данных, с другой — принципами и направлениями разработки мобильных устройств, концепциям разработки и функционирования соответствующего системного программного обеспечения, методам и алгоритмам их проектирования и реализации.

В конце каждой главы предлагаются вопросы, которые могут быть использованы для самопроверки, а также текущего или дополнительного опроса.

ВВЕДЕНИЕ

Основные понятия учебной дисциплины «Системное программное обеспечение»

1. Роль и место учебной дисциплины «Системное программное обеспечение» в системе подготовки специалиста в области автоматизации систем управления и его последующей практической деятельности.

Дисциплина «Системное программирование» является дисциплиной базовой части профессионального цикла учебного плана образовательной программы «Прикладная математика и информатика».

Она имеет логическую и содержательно-методическую взаимосвязь со следующими учебными дисциплинами: информатика, программирование, операционные системы и технологии разработки специального программного обеспечения, а также другими дисциплинами как математического и естественно-научного, так и профессионального циклов.

2. Цель, задачи, содержание и порядок изучения дисциплины.

Целью освоения дисциплины «Системное программирование» является формирование и развитие у обучающихся личностных и профессионально важных качеств — специалиста в области системного программирования — на основе овладения ими компетенциями в соответствии с требованиями Федерального государственного образовательного стандарта (ФГОС) высшего образования и квалификационных требований по данной специальности.

Для освоения дисциплины «Системное программирование» обучающемуся необходимы знания, умения и навыки, приобретенные в математическом и естественнонаучном, а также профессиональном циклах на первом и втором курсах обучения в результате освоения предшествующих дисциплин: информатика, программирование.

Указанные дисциплины должны обеспечивать следующие требования к «входным» знаниям, умениям и способностям обучающегося.

Обучающийся должен знать:

- назначение и виды информационных технологий;
- базовые и прикладные информационные технологии;
- инструментальные средства информационных технологий;
- теоретические основы информатики: алгебру логики, способы представления информации в компьютере;

- теоретические основы программирования;
- состав и назначение основных устройств ЭВМ.

Обучающийся должен уметь:

- обрабатывать текстовую и числовую информацию;
- применять полученные знания в профессиональной деятельности.

Обучающийся должен владеть:

- навыками подготовки ЭВМ к использованию;
- навыками работы на ЭВМ;
- навыками программирования на базовом языке программирования высокого уровня.

Освоение дисциплины «Системное программирование» необходимо для последующего изучения многих учебных дисциплин, в частности, «Операционные системы и сети», «Параллельные системы» и многих других дисциплин профессионального цикла.

Дисциплина изучается в 4 семестре и обеспечивает 2 зачетные единицы. В данном учебном пособии рассматриваются некоторые теоретические вопросы этой дисциплины.

Системное программное обеспечение (System Software, в дальнейшем СПО) — будем рассматривать в узком смысле, то есть как совокупность программ и программных комплексов для обеспечения работы компьютера, сетей ЭВМ и вычислительных комплексов.

СПО в узком смысле управляет ресурсами компьютерной системы и позволяет разработчикам и программистам, ведущим сопровождение, программировать более выразительными средствами, то есть, на языках более высокого уровня, чем машинный язык компьютера. Состав СПО для такого рода задач мало зависит от характера решаемых задач пользователя проблемной области. Далее, употребляя термин СПО, будем иметь в виду СПО в узком смысле.

Системное программное обеспечение (СПО) предназначено для:

- создания операционной среды функционирования других программ (другими словами, для организации выполнения программ);
- автоматизации разработки (создания) новых программ и комплексов программ;
- обеспечения надежной и эффективной работы самого компьютера, вычислительной сети и вычислительного комплекса;
- проведения диагностики и профилактики аппаратуры компьютера и вычислительных сетей;
- выполнения вспомогательных технологических процессов (копирование, архивирование, восстановление файлов программ и баз данных и так далее).

Данный класс программных продуктов тесно связан с типом компьютера, типами устройств сети и их топологией и является их неотъемлемой частью.

Программные продукты данного класса ориентированы на квалифицированных пользователей — профессионалов в компьютерной области: системного программиста, администратора сети, прикладного программиста, оператора.

Построение дисциплины, контроль усвоения материала, распределение времени по темам.

В результате изучения предлагаемого модуля дисциплины (а именно, настоящего пособия) обучающийся должен:

Знать:

- принципы разработки трансляторов;
- способы представления входных и выходных структур данных на этапах трансляции;
- особенности построения и функционирования лексических, синтаксических и семантических блоков компиляторов;
- суть методов нисходящего и восходящего разборов синтаксических конструкций;
- методы генерации кодов и оптимизации;
- структуру и функциональные средства систем программирования
- принципы распараллеливания в современных многопроцессорных системах;
- перспективы развития современного системного программного обеспечения.

Уметь:

- строить деревья разбора синтаксических конструкций;
- разрабатывать алгоритм анализа рекурсивных грамматик;
- применять алгоритмы построения конечных автоматов по заданным формальным грамматикам и программировать их;
- применять алгоритм построения обратной польской записи для арифметических выражений;
- разрабатывать алгоритмы с использованием статической и динамической памяти (строки, структуры списки);
- анализировать структуру разрабатываемой программы с точки зрения оптимизации по заданным критериям;
- разбираться в особенностях реализации параллельных конструкций при разработке параллельной программы или преобразовании последовательной в параллельную.

Принципы построения современных вычислительных систем

Глава 1. Принципы организации и функционирования систем машинного перевода (трансляторов и интерпретаторов)

Учебные цели:

- дать определение общего понятия трансляции;
- дать представление об особенностях функционирования компиляторов, интерпретаторов, декомпиляторов;
- дать представление о принципах функционирования транслятора.

1.1. Трансляторы, компиляторы, интерпретаторы. Общие свойства и различия

Рассмотрим классический механизм создания системного программного продукта — трансляцию программы.

Понятие *собственно* ***трансляции*** включает два понятия:

- либо компиляции,
- либо интерпретации.

Компиляция включает в свой состав несколько основных этапов:

- трансляцию с получением промежуточного кода;
- связывание программ в промежуточных кодах (использование редактора связей);
- оптимизацию кода;
- генерацию исполняемого кода.

Интерпретатор анализирует программу по частям, выполняя частично программу до контрольной точки, и, возможно, выдавая промежуточные результаты.

Компиляция отличается от интерпретации тем, что при компиляции полностью осуществляется лексический и синтаксический, а также семантический проход (иногда и контекстно-свободный анализ). При этом результатами компиляции является

- либо список ошибок компиляции и невозможность дальнейшего выполнения программы;
- либо объектный код программы без ошибок компиляции, который возможно сохранить в отдельном файле и в дальнейшем (или непосредственно за компиляцией) запустить на выполнение.

Понятно, что для каждого языка нужен свой транслятор. А для некоторых языков существует много реализаций.

Например, реализаций Паскаля[1] и Си[2] для ЭВМ типа IBM/PC на рынке существует больше десятка.

Обилие *реализаций* зависит как от архитектуры ЭВМ, для которой пишется транслятор, так и от стандарта представлений объектов языка (написание, например, в кавычках или без, с учетом пробелов или без и тому подобное).

Кроме этого, постоянно растущая потребность в новых компиляторах связана с бурным развитием архитектур ЭВМ.

Это развитие идет по различным направлениям.

- *Совершенствуются старые архитектуры*, как в концептуальном отношении, так и по отдельным линиям.

  Так, например, GCC (сокращение от GNU[3] Compiler Collection — набор компиляторов для различных языков программирования) поддерживает более 40 архитектур процессоров. В рамках последовательных машин возникает большое число новых архитектур, в которых процессоры
  - *ориентированы на приложения*;
  - являются процессорами *с длинным командным* словом;
  - представляют собой *нейрокомпьютеры, транспьютеры*;
  - представляют собой *суперскалярные* процессоры.

  Естественно, что для каждой новой системы команд требуется полный набор новых компиляторов с распространенных языков.
- Развиваются *различные параллельные* архитектуры (сейчас особенно). Среди них есть векторные, многопроцессорные. Параллельные архитектуры встречаются сегодня во всех классах ЭВМ — от супер-ЭВМ до PC[4].
- Естественно, для каждой из этих машин создаются трансляторы для ***многих*** языков программирования.

Итак, дадим определения.

---

[1] Паскаль (англ. Pascal) — один из наиболее известных императивных языков программирования, созданный Никлаусом Виртом в 1968—1969, используется для обучения программированию в старших классах и на первых курсах вузов, является основой для ряда других языков. Целью создания такого языка являлось построение небольшого и эффективного языка, способствующего хорошему стилю программирования, использующему структурное программирование и структурированные данные.

[2] Си или C (англ. C) — компилируемый статически типизированный язык программирования общего назначения, разработанный в 1969—1973 годах сотрудником Bell Labs Деннисом Ритчи, был разработан для реализации операционной системы UNIX, но впоследствии был перенесён на множество других платформ. Нашел применение в различных проектах, как ориентированных на язык ассемблера (например, в операционных системах), так и в различном прикладном программном обеспечении для множества устройств — от суперкомпьютеров до встраиваемых систем. Язык программирования Си оказал существенное влияние на развитие современного программного обеспечения, а его синтаксис стал основой для таких языков программирования, как C++, C#, Java и Objective-C.

[3] GNU/Linux — версия свободной Unix-подобной операционной системы (ГНУ, расшифровывается как GNU's Not Unix, ГНУ — Не Юникс), в которой используется ядро Linux, написанное группой энтузиастов под руководством Линуса Торвальдса из Финляндии. Проект GNU (англ. The GNU Project) — проект по разработке свободного программного обеспечения, является результатом сотрудничества множества отдельных проектов. Изначальной целью проекта было «разработать достаточно свободного программного обеспечения, чтобы можно было обойтись без программного обеспечения, которое не является свободным».

[4] PC (англ. personal computer) «персональный компьютер», ПК.

В общем случае, ***транслятор*** — это программа, предназначенная для автоматического перевода описания алгоритма с одного языка программирования на другой, в классическом случае, на машинный язык.

В частности, в этом разделе будем рассматривать ***транслятор*** **—** как программу, которая переводит *исходную* программу *в эквивалентную ей объектную программу. Исходная программа* пишется на некотором исходном языке, объектная программа формируется на объектном языке. Так, исходным языком может быть язык высокого уровня (Фортран-99, Си), а объектным языком — ассемблер или некоторый машинный язык. Машинный язык иногда называют кодом машины, поэтому и объектная программа иногда называется объектным кодом.

Трансляция исходной программы *в объектную* происходит во время *компиляции*, а фактическое *выполнение объектной* программы происходит во время *выполнения готовой* программы.

*Транслятор я*вляется частью базового программного обеспечения ЭВМ, одним из средств автоматизации программирования.

Трансляция, в общем случае, — это общее обозначение процесса преобразования исходного кода в некоторый другой вид. По характеру функционирования этого процесса трансляторы делят на три типа (рис. 1.1): компиляторы, интерпретаторы (преобразующие исходный код в виде последовательности операторов языка программирования в исполняемый код) и дешифраторы (преобразующие исходный код на языке низкого уровня или исполняемых кодов в код на языке операторов языка более высокого уровня).

Рис. 1.1. Типы трансляторов

Интерпретатор **—** программа, осуществляющая пошаговое исполнение текста исходной программы. Одновременно и транслирует, и использует исходную программу.

Рис. 1.2. Общая схема процесса, выполняемого интерпретатором

Интерпретатор для некоторого языка *принимает исходную*

программу, написанную на этом языке как входную информацию *и выполняет её* (рис. 1.2).

Различие между *компилятором* и *интерпретатором* состоит в том, что *интерпретатор* ***не*** порождает объектную программу, которая должна превратиться в программу на машинном языке и затем выполниться (рис. 1.3).

Интерпретатор *непосредственно выполняет* программу сам (последовательно выбирая, анализируя и исполняя инструкцию за инструкцией).

Рис. 1.3. Детальная схема работы интерпретатора

Для того, чтобы осуществить выполнение инструкций исходной программы повторно, чистый интерпретатор *анализирует* её ***всякий раз,*** когда она должна быть *выполнена, с самого начала*,

При программировании интерпретатор обычно разделяют на 2 фазы:

- 1 фаза — анализируется вся исходная программа (почти также, как в компиляторе) и транслируется в некоторое внутреннее представление (которое невозможно сохранить в отдельном файле, в отличие от фазы получения объектного модуля в процессе компиляции);
- 2 фаза — внутреннее представление исходной программы интерпретируется (или выполняется).

Внутреннее представление разрабатывается для того, чтобы *уменьшить время выполнения инструкций*.

Существуют языки программирования с конструкциями, позволяющими создавать новые подпрограммы или модифицировать существующие динамически, во время выполнения программы.

Независимо от языка программирования интерпретация может быть (чаще всего таким образом и используется) предпочтительнее компиляции *на этапе отладки*.

Пример 1.1. Для трансляции регистро-независимого, одного из старейших языков программирования, ***LISP***[5] используется интерпретатор.

Пример 1.2. Для трансляции программ, исполняемых на виртуальной

[5] Лисп (***LISP***, от англ. *LISt Processing language* — «язык обработки списков») — семейство языков программирования, в которых код программы и данные представляются системами линейных списков символов. Лисп был создан Джоном Маккарти для работ по искусственному интеллекту и до сих пор остаётся одним из основных инструментальных средств в данной области, а также применяется как средство обычного промышленного программирования: от встроенных скриптов до веб-приложений массового использования. Лисп прошёл фундаментальную стандартизацию для использования в военном деле и промышленности; существуют его реализации для большинства современных платформ.

машине ***Java***[6], также используется интерпретатор, который интерпретирует текстовый код в низкоуровневый байт-код (такая интерпретация делает код независимым от архитектуры ЭВМ).

Компилятор — это программа, которая преобразуют исходный текст программы, написанной на языке программирования высокого уровня, в некоторое представление, пригодное в дальнейшем (или непосредственно) для выполнения ее на компьютере.

Компилятор порождает объектную программу, которая должна превратиться в программу на машинном языке и затем выполниться. При этом процесс порождения объектной программы представляет несколько последовательных фаз анализа: сначала выполняется анализ полного исходного текста в некоторое внутреннее представление, затем несколько фаз анализа более сложного типа очередного полученного внутреннего представления в другую структуру внутреннего представления вплоть до получения так называемого объектного (машинного) кода (рис. 1.4).

Важно подчеркнуть, что если процесс компиляции (фазы анализа) выполняется целиком для всей программы, и если этот процесс прошел неудачно, то выполнение программы невозможно, а результатом компиляции исходной программы в этом случае будет список найденных ошибок и предупреждений. Этим процесс компиляции отличается от процесса интерпретации, когда возможна трансляция фрагмента исходной программы до контрольной точки и выполнение этого фрагмента программы, не доходя до конца исходного текста программы.

Рис. 1.4. Принципиальная схема процесса компиляции

Разделение фаз компиляции и выполнения очень удобно для отладки программы по существу правильности выполнения, так как файл объектного кода можно неоднократно выполнять на различных данных.

Компиляторы раньше писались вручную, сейчас пишутся на языках высокого уровня (как, например, компилятор Си) или с использованием специальной инструментальной оболочки ("компиляторы компиляторов").

Декомпиля́тор — это программа, транслирующая исполняемый модуль (полученный на выходе компилятора) в эквивалентный исходный код на

[6] ***Java*** — сильно типизированный объектно-ориентированный язык программирования, один из самых популярных языков программирования. Назван в честь марки кофе *Java*, которая, в свою очередь, получила наименование одноимённого острова (Ява), поэтому на официальной эмблеме языка изображена чашка с горячим кофе. Существует и другая версия происхождения названия языка, связанная с аллюзией на кофе-машину, как пример бытового устройства, для программирования которого изначально язык создавался.

языке программирования высокого уровня. Иначе говоря, *декомпилятор* — программа, позволяющая по программе на языке низкого уровня (коды на уровне машинного языка, объектные коды) получить программу на высокоуровневом языке, в некотором смысле ей эквивалентную.

Эквивалентность рассматривается в смысле внешних проявлений работы исходной программы и декомпилированной. Таким образом, *декомпиля́ция* — это сложный процесс воссоздания исходного кода декомпилятором (рис. 1.5). Декомпиляция, в частности, используется при обратной разработке программ.

Рис. 1.5. Этапы декомпиляции

Не всегда процесс декомпиляции дает положительные результаты: маленькие программы вполне поддаются анализу, с большими объемами кодов возникают сложности.

Приблизительно восстановить программу в первозданном виде возможно, выполнив следующие действия:

— выяснить язык программирования, чей исполняемый код

представляет исходная для декомпиляции программа;

— предварительно провести дизассемблирование этой программы;

— провести тщательный анализ её библиотек;

— разделить исполняемый код на исходные составляющие.

Как правило, полученный код значительно отличается от написанного программистом: он выглядит запутанным, и разобраться в таком исходнике порой бывает проблематично. Особенно в том случае, когда еще при компиляции среды разработки применялся метод *обфускации* (запутывания) кода. В таком случае процедура декомпиляции становится затруднительной или практически невозможной.

Отметим, что удачность декомпиляции зависит от объема информации, представленной в декомпилируемом коде. Байт-код, используемый большинством виртуальных машин (таких, например, как *Java*) часто содержит обширные метаданные, делающие декомпиляцию вполне выполнимой, в то время как машинный код более скуден и сложен в декомпиляции. В частности, трудночитаемыми представляются вызовы подпрограмм или функций с косвенной адресацией вызовов (в терминах языков программирования высокого уровня — вызовы через указатели на функции/процедуры).

Вообще говоря, на уровне декомиляторов есть еще усложнители декомпиляции (шифраторы, обфускаторы).

Заметим, что и другие программные средства могут рассматриваться как языковые трансляторы, а именно:

- текстовые редакторы и текстовые процессоры (текстовые и/или специальные команды преобразуются в образ: *Word, TEX, HTML*);
- процессоры запросов (высокоуровневые языки запросов — *SQL*);
- программы для доказательства теорем (преобразуют спецификации теоремы в некоторый метаязык).

1.2. Формальное определение компилятора

Итак, ***компилятор*** — программа, преобразующая исходный текст, написанный на алгоритмическом языке, в программу, состоящую из машинных команд (или объектных кодов). Компилятор создает *законченный* вариант программы на машинном языке.

Одновременно с разработкой первых языков программирования в конце 50-х годов появились и первые компиляторы. Данное направление компьютерной науки, несмотря на 70-летнюю историю, нельзя назвать устоявшимся или устаревшим. Наоборот, с ходом времени, появлением новых задач и отраслей, для решения которых используются персональные компьютеры, суперкомпьютерные комплексы, появляется необходимость в разработке новых, более удобных языков

программирования или поддержка и распространение существующих языков на новых платформах.

Для всех этих языков, соответственно, и требуются компиляторы, причем свои разработки существуют для каждой платформы.

Исходный текст, созданный на языке высокого уровня разработчиком, должен быть преобразован в программу, написанную на специальном машинном языке. Этот код и называют исполняемой программой. Исполняемую программу можно устанавливать и запускать на любом персональном компьютере, не делая при этом никаких преобразований. Основные блоки компилятора представлены на рисунке 1.6.

Рис. 1.6. Основные блоки компилятора

Компиляторы по традиции являются одной из основных вещей в информатике, наряду с базами данных и операционными системами. Компилятор - это в каком-то смысле базис современной компьютерной науки.

Направление системного программного обеспечения, связанного с разработкой компиляторов, подразумевает большое количество технологических и теоретических аспектов, связанных с программированием.

Создатели компиляторов сталкиваются со множеством различных проблем. Это и научные проблемы, которые связаны с правильным отображением понятий в прикладной области, и технологические, и инженерные проблемы, связанные с реализацией отображения. При создании компилятора приходится выполнять множество разнородных подзадач. Это очень сложная отрасль, которой программисты посвящают всю свою жизнь.

В последнее время можно проследить четкую тенденцию, связанную с тем, что любая крупная компания в сфере информационных технологий выпускает собственный язык программирования, который затем продвигается в массы. Для каждого языка программирования требуется свой собственный компилятор. Как правило, их создают вместе с языками.

Однако, существует большое количество фирм и самостоятельных

программистов, которые хотят иметь собственные компиляторы для тех или иных языков, или же разрабатывают собственные языки программирования и соответственно компиляторы к ним. Компилятор — это своеобразная программа-переводчик, которая используется для взаимодействия между разработчиком и компьютером. Сегодня в сфере компьютерной техники этот программный продукт наиважнейший.

1.3. Динамическая компиляция

Динамическая компиляция (*Just-in-time compilation – JIT*), также известна как *dynamic translation*) — компиляция «на лету» — это технология увеличения производительности программных систем, использующих байт-код, путём трансляции байт-кода в машинный код непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения (сравнимая с компилируемыми языками) за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. Динамическая компиляция (JIT) базируется на двух более ранних идеях, касающихся среды исполнения: компиляции байт-кода и динамической компиляции.

Из-за необходимости интерпретации байт-код выполняется значительно медленнее машинного кода сравнимой функциональности, однако он более переносим (не зависит от операционной системы и модели процессора). Чтобы ускорить выполнение байт-кода, используется динамическая компиляция, когда виртуальная машина транслирует псевдокод в машинный код непосредственно перед его первым исполнением (и при повторных обращениях к коду исполняется уже скомпилированный вариант). Схема динамической компиляции приведена на рисунке 1.7.

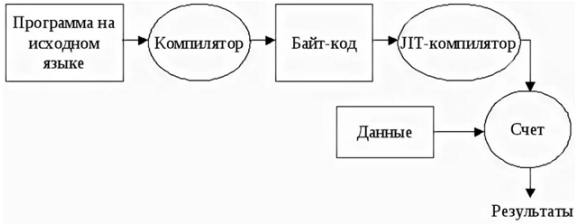


Рис. 1.7. Схема динамической компиляции (компиляции “на лету”)

Процесс компиляции *Java*-приложения отличается от процесса компиляции статически компилируемых языков программирования, подобных *Си* или *Си++*.

Статический компилятор преобразует исходный код *непосредственно* в машинные инструкции, которые могут быть выполнены на целевой платформе. Различные аппаратные платформы требуют применения

различных компиляторов.

Java-компилятор преобразует исходный Java-код в переносимые байт-коды JVM, которые являются для JVM[7] "инструкциями виртуальной машины".

Виртуальная машина Java исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java (***Javac***[8]). Javac написан на Java, может вызваться непосредственно из Java-программ. Виртуальная машина Java (JVM) может также использоваться для выполнения программ, написанных на других языках программирования.

Например, исходный код на языке Ada может быть скомпилирован в байт-код Java, который затем может выполниться с помощью виртуальной машины Java (JVM).

Виртуальная машина Java (JVM) является ключевым компонентом платформы Java. Так как виртуальные машины Java доступны для многих аппаратных и программных платформ, Java может рассматриваться и как связующее программное обеспечение, и как самостоятельная платформа. Использование одного байт-кода для многих платформ позволяет описать Java как «скомпилировано однажды, запускается везде» (*compile once, run anywhere*).

Виртуальные машины Java обычно содержат интерпретатор байт-кода, однако, для повышения производительности во многих машинах также применяется JIT-компиляция часто исполняемых фрагментов байт-кода в машинный код.

В отличие от статических компиляторов ***Javac*** выполняет очень маленькую оптимизацию; оптимизация же, проводимая статическими компиляторами, выполняется во время исполнения программы. Подробнее ***Java*** и ***Javac*** рассмотрим позднее.

Байт-код не является машинным кодом какого-либо компьютера и может *портироваться* (имеет свойство *переносимости*) на различные компьютерные архитектуры. Байт-код интерпретируется (исполняется) виртуальной машиной.

Примеры языков, компилирующихся в байт-код: ***Perl***[9]***, GNU CLISP***[10].

---

[7] ***Java Virtual Machine*** (сокращенно *Java VM, JVM*) — виртуальная машина ***Java*** — основная часть исполняющей системы ***Java***, так называемой ***Java Runtime Environment*** (*JRE*).

[8] ***Javac*** — оптимизирующий компилятор языка ***Java***, включенный в состав многих экосистема инструментов, охватывающая почти все, что может понадобиться при программировании на ***Java***. Компилятор принимает исходные коды, соответствующие одной спецификации комплекта экосистемы инструментов ***Java***, и возвращает байт-код, соответствующий спецификации *Java Virtual Machine Specification*.

9 ***Perl*** — высокоуровневый интерпретируемый динамический язык программирования общего назначения, созданный Ларри Уоллом, лингвистом по образованию. Название языка официально расшифровывается как *Practical Extraction and Report Language* («практический язык для извлечения данных и составления отчётов»), а в шутку — как *Pathologically Eclectic Rubbish Lister* («патологически эклектичный перечислитель мусора»). Первоначально название состояло из пяти символов и в таком виде в точности совпадало с английским словом *pearl* («жемчужина»). Но затем стало известно, что такой язык существует, и букву «а» убрали. Символом языка ***Perl*** является верблюд — не слишком красивое, но очень выносливое животное, способное выполнять тяжёлую работу.

Динамическая компиляция (*JIT*) используется в реализациях языков платформы *Microsoft.NET*. Реализации известного и популярного сегодня языка ***Python***[11] имеют ограниченные или неполные варианты динамической компиляции (*JIT*).

Вопросы для контроля

1. Дайте определение транслятора.
2. В чём разница в процессе выполнения статической компиляции и компиляции динамической?
3. Какие из известных вам продуктов программного обеспечения можно отнести к трансляторам?
4. В каких случаях удобнее использовать интерпретатор?

[10] ***CLISP*** — реализация языка программирования *Common Lisp*. Является свободным программным обеспечением и частью проекта ***GNU***. В состав ***CLISP*** входят *интерпретатор, компилятор байт-кода, отладчик, а также интерфейс сокетов, интерфейс для* стыковки *с другими языками* программирования, сильная поддержка интернационализации и объектные системы (***CLOS*** и ***MOP***). ***CLISP*** написан на языках программирования ***Cu*** и ***Common Lisp***.

[11] ***Python*** ([ˈpʌɪθ(ə)n]; в русском языке распространено название пито́н) — высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Синтаксис ядра ***Python*** минималистичен. В то же время стандартная библиотека включает большой объём полезных функций. ***Python*** поддерживает структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное программирование.

Глава 2. Этапы трансляции

Учебные цели:

- дать представление об общей схеме трансляции;
- дать представление об особенностях функционирования этапов компиляции;
- дать представление о внутреннем представлении входных и выходных структурах данных.

2.1. Анализ общей схемы работы транслятора.

Рассмотрим общую схему механизма трансляции, обратив основное внимание на крупные блоки процесса компиляции как наиболее сложного процесса трансляции.

Согласно материалу предыдущей главы, процесс трансляции можно представить тремя различными процессами: компиляции, интерпретации или декомпиляции. Детально процессы интерпретации и декомпиляции рассматривать не будем. Процесс интерпретации может рассматриваться как частный случай процесса компиляции, представляющий упрощенную схему. Процесс декомпозиции не входит в рамки настоящего рассмотрения.

Рассмотрим более подробно общую схему компиляции (рис. 2.1).

Рис. 2.1. Общая схема механизма трансляции.

Процесс компиляции состоит из двух частей: анализа (*analysis*) и синтеза (*synthesis*). Анализирующая часть компилятора разбивает

исходную программу на составляющие ее элементы (конструкции языка — *language constructions*) и создает промежуточное представление исходной программы. Синтезирующая часть из промежуточного представления создает новую, целевую, программу.

Основным блоком компилятора является блок *анализатора*, в котором и происходит последовательный анализ текстового кода программы, этот блок позднее мы рассмотрим гораздо подробнее.

При анализе текста возможно использование *генератора анализаторов* — инструментального средства, генерирующего лексические или семантические конструкции и связанные с ними действия, представляющие конструкции рассматриваемого языка и их первоначальный анализ.

Следующим обязательным блоком является *оптимизатор*.

Оптимизация — это процесс преобразования фрагмента кода в другой фрагмент, который функционально эквивалентен исходному, с целью улучшения одной или нескольких его характеристик, из которых наиболее важными являются скорость и размер кода. Оптимизация улучшает *реализацию* алгоритма компиляции, но сам алгоритм существенно улучшить нельзя. Однако, если оптимизация сделана удачно, программа может быть ускорена в несколько раз.

Фаз оптимизации может быть несколько.

Оптимизации обычно делят на

1. машинно-зависимые и машинно-независимые,
2. локальные и глобальные.

Машинно-независимую оптимизацию представляет, например, оптимизация на уровне исходного языка (при этом в результате получается программа, записанная на том же самом языке) либо оптимизация на уровне машинно-независимого промежуточного представления (при этом получается более эффективная с точки зрения реализации программы на промежуточном языке).

Машинно-зависимую оптимизация — оптимизация на уровне машинного языка. Так, часть машинно-зависимой оптимизации выполняется в фазе генерации кода.

Глобальная оптимизация отличается от *локальной* тем, что процесс глобальной оптимизации пытается принять во внимание структуру *всей* программы, а локальная оптимизация рассматривает только небольшие фрагменты программы.

Глобальная оптимизация основывается на глобальном потоковом анализе, который выполняется на графе программы и представляет, по существу, преобразование этого графа. При этом могут учитываться такие свойства программы, как

— межпроцедурный анализ,
— межмодульный анализ,
— анализ областей жизни переменных и так далее.

На этапе оптимизации можно использовать такие инструментальные средства как *усложнитель* и *интерпретатор промежуточных представлений* (рис. 2.1*)*.

Наконец, блок кодогенератора формирует из полученных на предыдущем шаге внутренних представлений заключительную часть программы – коды выполняемой программы.

2.2. Изучение особенностей построения интерпретаторов.

Учитывая информацию о структуре интерпретатора, изложенную в предыдущей главе, рассмотрим сравнительные особенности процесса интерпретации. Основным блоком интерпретатора, как и в случае процесса компиляции, является блок *анализатора*, но совмещенный с блоком оптимизатора и генератором кодов. То есть при выборе очередного фрагмента исходного текста программы происходит

— достаточно простой анализ конструкций этого фрагмента текстового кода программы,
— формирование некоторого внутреннего представления рассмотренного фрагмента программы и
— генерация кодов этого фрагмента с помощью кодогенератора.

Таким образом, интерпретатор позволяет получить исполнение (или вычисление) фрагмента программы, не доходя до конца программы.

Так как процесс интерпретации упрощен по сравнению с процессов компиляции, *генератор анализаторов* в этом случае не используется, *Оптимизация* же производится, но, большей частью, локальная. По окончании процесса интерпретации получаем результаты вычислений, однако сохранить результаты внутреннего представления при интерпретации для повторного запуска для выполнения вычислений невозможно.

2.3. Изучение внутреннего представления входных и выходных структур данных на этапах трансляции.

Практически структуру процесса компиляции программы с учетом представления входных и выходных структур данных в общем случае можно увидеть на рисунке 2.2.

Исходная текстовая программа может состоять из одного ли нескольких файлов и находиться на одном или нескольких внешних носителях (например, файлы с расширением ***c***) в оперативной памяти или на внешних дисках.

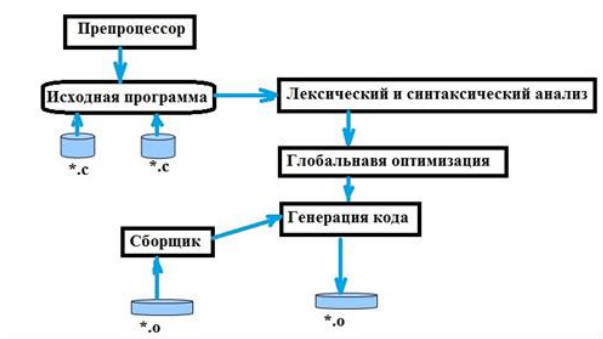


Рис. 2.2. Составные части процесса компиляции

Процесс компиляции разделяется на следующие составляющие части.

1. ***Препроцессор.*** Исходная программа обрабатывается путем подстановки имеющихся макросов и заголовочных файлов.
2. ***Лексический и синтаксический анализ.*** Программа преобразуется в цепочку лексем, а затем во внутреннее представление в виде дерева.
3. ***Глобальная оптимизация.*** Внутреннее представление программы неоднократно *преобразовывается* с целью *сокращения размера и времени исполнения* программы.
4. ***Генерация кода.*** Внутреннее представление преобразуется в блоки команд процессора, которые затем преобразуются в ассемблерный текст или объектный код (файлы с расширением ***o***).
5. ***Ассемблирование.*** Если генерируется ассемблерный текст, на выходе получаем объектный код.
6. ***Сборка*** **(*линковка*).** Сборщик (редактор связей, линкер) соединяет несколько объектных файлов, помещаемых в исполняемый файл в оперативной памяти или библиотеку.

Препроцессор — это программа, которая производит определенные действия (более или менее значительные) с исходным текстом программы перед тем, как передать его блокам компилятора, то есть производит предобработку исходного текста

Препроцессоры создают входной текст для компиляторов и могут выполнять следующие функции:

- обработку макроопределений,
- включение файлов,
- "рациональную" предобработку,
- расширение языка.

Препроцессор (или макропроцессор) выполняет действие, исполняя директиву препроцессора. Например, в Си / Си++ и им подобных языках директива препроцессора или оператор препроцессора — это одна строка исходного текста, начинающаяся с символа «***#***», за которым следуют название оператора (***define, pragma, include, if***) и операнды. При этом операторы препроцессора могут появляться в любом месте программы, а их действие распространяется на весь исходный файл.

Например, препроцессор добавляет ссылки на библиотеки в код по директиве ***#include***, убирает комментирования, заменяет макросы по директиве ***#define*** их значениями, выбирает нужные куски кода в соответствии с условиями ***#if, #ifdef*** и ***#ifndef***.

Так, в Си/Си++ вы уже знакомились в рамках дисциплины «Программирование» с примерами применения директив, например:

#include N

#define struct Steck

Лексическому и синтаксическому анализу, глобальной оптимизации, генерации кодов будет посвящены отдельные главы.

Если исполняемая программа должна быть представлена в виде команд в бинарном коде, а на этапе генерирования команд получили программу на языке низкого уровня (например, ассемблере), необходимо перевести ассемблерный код в машинный, делается это с помощью ***ассемблирования***.

Ассемблерный код — это доступное для понимания человеком представление машинного кода, где вместо групп нулей и единиц, представляющих машинную команду, записывается символьное обозначение кода команды и ее операндов, например:

A r1, r2,

что означает сложить содержимое регистра ***r1*** и регистра ***r2.*** В качестве машинной команды пусть эта запись обозначает:

01100011 11100001 10101000

Ассемблер преобразовывает ассемблерный код в машинный код, сохраняя его в *объектном файле*.

Сборщик или редактор связей (называют еще линкер) собирает все требуемые бинарные файлы в единый исполняемый файл или библиотеку статического или динамического типа.

Замечание. Одновременно с понятием *оптимизации* полезно рассмотреть понятие *рефакторинга*.

Рефакторинг — это процесс *улучшения внутренней структуры* текста программы, при котором, в отличие от оптимизации, не меняется её внешнее проявление, то есть *рефакторинг* — это контролируемый процесс улучшения кода, без написания новой функциональности (по

Фаулеру[12]). *Рефа́кторинг,* или перепроектирование кода, переработка кода, равносильное преобразование алгоритмов — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. Результат рефакторинга — это чистый код и простой дизайн.

В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и чёткости.

Цель рефакторинга — сделать код программы более легким для понимания; без этого рефакторинг нельзя считать успешным.

Рефакторинг следует отличать от *оптимизации производительности*. Как и рефакторинг, оптимизация обычно не изменяет поведение программы, а только ускоряет её работу. Но оптимизация часто *затрудняет* понимание кода, что противоположно рефакторингу.

С другой стороны, нужно различать термины *рефакторинг* и *реинжиниринг.* Реинжиниринг осуществляется для расширения функциональности программного обеспечения. Как правило, крупные рефакторинги предваряют реинжиниринг.

Вопросы для контроля

1. Какие функциональные блоки входят в состав компилятора?
2. Чем отличается глобальная оптимизация от локальной?
3. Приведите пример блока компиляции, где может применяться машинно-зависимая оптимизация.
4. Объясните, для чего применяется рефакторинг.
5. Объясните в чём функциональность препроцессора.

[12] Мартин Фаулер — британский программист. Автор ряда книг и статей по архитектуре ПО, объектно-ориентированному анализу и разработке, языку UML, рефакторингу, экстремальному программированию, предметно-ориентированным языкам программирования (Википедия).

Глава 3. Особенности построения и функционирования компиляторов

Учебные цели:

- дать представление о лексическом анализе компиляторов;
- дать представление об особенностях функционирования лексических и синтаксических блоков анализаторов
- дать представление о месте компиляторов в составе систем программирования.

В этой главе рассмотрим взаимосвязи между блоками процесса компиляции и основные информационные потоки, связанные с этими блоками, вид входящих и выходящих структур данных, взаимодействующих в процессе компиляции.

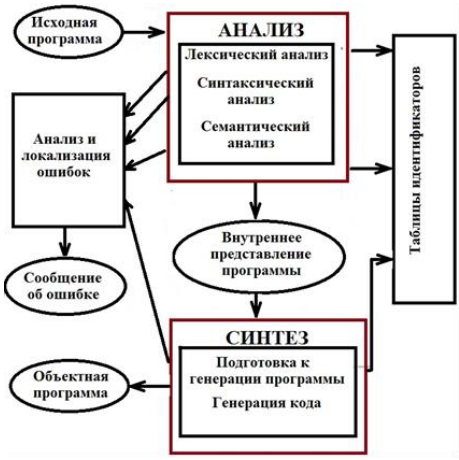


Рис. 3.1. Основные информационные связи и потоки компиляции.

На рисунке 3.1 представлена более детальная схема полного процесса компиляции, на которой отображены информационные потоки, взаимодействующие с различными блоками процесса компиляции и сопутствующими ресурсами (таблицами) и средствами (службами обработки ошибок, прерываний и тому подобное).

Прежде всего, более подробно рассмотрим блок анализа исходной программы (анализатор) и собственно этапы анализа процесса компиляции (см. рис. 3.1, блок АНАЛИЗ).

Как для процесса компиляции, так и для процесса интерпретации важное значение имеют, прежде всего,

— информационные таблицы: символьная таблица и интерфейс

доступа к ней (для фиксации имен и характеристик, дескрипторов переменных, файлов и прочих объектов);

— интерфейс операционной системы и обработки ошибок (таблица кодов сообщений операционной системы, в том числе, ошибок компиляции и ошибок выполнения программы).

Рис. 3.2. Блок анализа процесса компиляции (АНАЛИЗАТОР)

В блок анализа процесса компиляции входят такие этапы, или *проходы*, как их называют специалисты — разработчики трансляторов, как лексический анализатор, синтаксический анализатор, семантический анализатор, а иногда и контекстно-свободный анализатор.

3.1. Лексические анализаторы (сканеры)

В фазе *лексического анализа* (в литературе часто обозначают ЛА) входная программа, представляющая поток исходных *терминальных* символов, разбивается на ***лексемы*** — слова (символы промежуточного языка, *нетерминалы*) — в соответствии с определениями исходного языка.

Так, лексема, например, соответствует

— идентификатору,
— ключевому (служебному) слову,
— строковой, символьной или числовой константе, например, целому числу,
— знакам операций и одно-двух литерному разделителю, например, одному из элементов приведенного множества {*, +, (,), {, }, //, /*, */}.

Иногда лексемы в литературе называют атомами.

Теоретически *лексический анализатор* не является обязательной частью компилятора. Все функции лексического анализатора могут выполняться на этапе синтаксического разбора, так как полностью определяются синтаксисом входного языка. Однако существует значимые причины, почему в состав практически всех компиляторов включают лексический анализ, а именно, выделение лексического анализатора

— *сокращает* объем информации, обрабатываемой на этапе синтаксического разбора;
— позволяют *упростить* процесс решения некоторых задач, требующих использования сложных вычислительных методов на этапе синтаксического анализа, более простыми методами на этапе лексического анализа (например, решение задачи различения операции унарного минуса и бинарной операции вычитания, обозначаемых одним и тем же знаком «–»);
— *разделяет* два блока: сложный по конструкции блок синтаксического анализатора и блок более простой работы лексического анализатора непосредственно с текстом исходной программы (ввиду того, что структуру лексического анализатора можно варьировать в зависимости от архитектуры вычислительной системы, где выполняется компиляция, для перехода на другую вычислительную систему достаточно будет только перестроить лексический анализатор);
— *позволяет* в современных системах программирования осуществлять обработку текста исходной программы *параллельно* с его подготовкой пользователем (это дает системе программирования принципиально новые возможности, которые позволяют снизить трудоемкость разработки программ).

Функции, выполняемые лексическим анализатором, и состав типов лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от реализации компилятора. Конкретно, выбор функций, которые должен выполнять лексический анализатор, а какие оставлять для этапа синтаксического разбора, решают разработчики компилятора. В основном лексические анализаторы выполняют

— исключение из текста исходной программы комментариев и незначащих символов (пробелов, символов табуляции и перевода строки),
— выделение лексем следующих, указанных ранее, типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка, знаков операций и разделителей.

Лексический анализатор может работать в 2 основных режимах:

— либо как программа, вызываемая синтаксическим анализатором за очередной лексемой;
— либо как полный проход, результатом которого является полный

набор лексем, полученных при лексическом анализе всего исходного текста, причем, этот набор собран в отдельный файл.

В соответствии с указанными режимами по-разному организуется связь со вспомогательными ресурсами (таблицами, интерфейсами), а также различаются способы реализации выделения лексем.

В процессе выделения лексем (ЛА) лексический анализатор

— в первом случае может как самостоятельно строить таблицы имен и констант, так и выдавать значения лексемы при обращении к нему;
— во втором случае таблица имен строится в последующих фазах анализатора (например, в синтаксическом анализе).

Результатом работы лексического анализатора является полный последовательный перечень всех найденных в тексте исходной программы лексем. Этот перечень лексем можно представить в виде таблицы, называемой таблицей лексем. Каждой лексеме в таблице лексем соответствует

— некий уникальный условный код, зависящий от типа лексемы, и
— дополнительная служебная информация.

Кроме того, информация о некоторых типах лексем, найденных в исходной программе, должна помещаться в таблицу идентификаторов (или в одну из таблиц идентификаторов, если компилятор предусматривает различные таблицы идентификаторов для различных типов лексем).

Обратите внимание, что таблица лексем и таблица идентификаторов — это две принципиально разные таблицы, обрабатываемые лексическим анализатором.

Таблица лексем фактически содержит весь текст исходной программы, обработанный лексическим анализатором. В нее входят все возможные типы лексем, кроме того, любая лексема может встречаться в ней любое количество раз.

Таблица идентификаторов содержит только определенные типы лексем — *идентификаторы* и *константы*. В нее не попадают такие лексемы, как ключевые (служебные) слова входного языка, знаки операций и разделители. Кроме того, каждая лексема, входящая в таблицу идентификаторов (идентификатор или константа), может встречаться в таблице идентификаторов только *один* раз.

Отметим также, что лексемы в *таблице лексем* обязательно располагаются в том же порядке, что и в исходной программе (порядок лексем в ней не меняется), а в *таблице идентификаторов* лексемы располагаются в любом порядке так, чтобы обеспечить удобство поиска (методы организации таблиц идентификаторов будут рассмотрены позже).

В качестве примера рассмотрим некоторый фрагмент исходного кода на языке Си

for (i= 1; i<N; i++) { fg = fg * 0.5;}

и соответствующую ему таблицу лексем, представленную в таблице 3.1:

Таблица 3.1. Лексемы программы

Лексема	Тип лексемы	Значение
for	Ключевое слово	X1
(	Разделитель	R1
i	Идентификатор	i ~ 1
=	Знак присваивания	S1
1	Целочисленная константа	1
;	Разделитель	R2
i	Идентификатор	i ~ 1
<	Знак арифметической операции	A1
N	Идентификатор	N ~ 2
;	Разделитель	R2
i	Идентификатор	i ~ 1
+	Знак арифметической операции	A2
+	Знак арифметической операции	A2
)	Разделитель	R3
{	Разделитель	R4
fg	Идентификатор	fg ~ 3
=	Знак присваивания	S1
fg	Идентификатор	fg ~ 3
*	Знак арифметической операции	A4
0.5	Вещественная константа	0.5
;	Разделитель	R2
}	Разделитель	R5

В поле «Значение» в этой таблице записано некоторое подразумеваемое кодовое значение, которое после разбора лексемы лексическим анализатором будет помещено в итоговую таблицу лексем.

Таблица 3.2 представляет малую часть информации, которую содержит таблица идентификаторов, и приведена для понимания разницы содержимого таблицы лексем и таблицы идентификаторов.

Таблица 3.2. Фрагмент таблицы идентификаторов

Лексема	Тип лексемы	Обозначение для связи с таблицей идентификаторов	Адрес ячейки, где находится значение
i	Идентификатор	i ~ 1	i~ 1***
1	Целочисленная константа	1	C~1***
N	Идентификатор	N ~ 2	N~1***
fg	Идентификатор	fg ~ 3	fg~ ***1
0.5	Вещественная константа	0.5	F~1***

В обеих таблицах значения, которые записаны в примерах (в таблице лексем в столбце «Значение», а в таблице идентификаторов — в столбце «Адрес ячейки, где находится значение»), являются условными.

Конкретные коды выбираются разработчиками при реализации компилятора. Причем, обязательно установка связки таблицы лексем с таблицей идентификаторов (в примере это отражено индексом, следующим после идентификатора за знаком тильда («~»), а в реальном компиляторе определяется его реализацией).

На этапе лексического анализа (ЛА) исходной программы компилятор может обнаружить некоторые (простейшие) ошибки такие, как:

— *недопустимые символы* (так как исходная программа формируются по правилам конкретного языка программирования, а значит, имеет конечный алфавит допустимых символов, чаще всего, символов клавиатуры),
— *неправильная запись чисел* (например, в записи действительного числа присутствуют два подряд идущих символа “точка” или два подряд идущих знака перед числом и тому подобное),
— *неправильная запись идентификаторов* (идентификатор начинается с цифры или содержит внутри себя недопустимые символы, например, символ “слэш”) и другие ошибки подобного типа.

Мы выяснили, что лексический анализатор имеет дело с такими объектами, как различного рода константы и идентификаторы (в том числе и ключевые слова языка).

Язык констант и идентификаторов является регулярным, а это означает, что может быть описан с помощью регулярных грамматик. В таком случае, как мы знаем из первой части учебного пособия, распознавателями для регулярных языков являются конечные автоматы. Значит, основой для реализации лексических анализаторов следует использовать регулярные грамматики и конечные автоматы.

Итак, в основе реализации лексических анализаторов лежит применение теории автоматов и регулярных выражений.

Прежде всего, как известно конечный автомат для каждой входной цепочки языка дает ответ на вопрос о том, принадлежит ли цепочка языку, заданному автоматом. В общем случае для лексического анализатора задача несколько шире, чем проверка цепочки символов лексемы на соответствие входному языку, необходимо чтобы он выполнил следующие действия:

— определил границы лексем, которые в тексте исходной программы *не указаны явно*;
— выполнить действия для *сохранения информации* об обнаруженной лексеме (или выдать сообщение об ошибке, если лексема неверна).

Таким образом, на выходе лексического анализа получаем промежуточный код в виде списка лексем, который представляет дерево с одной ветвью.

Простейший представитель лексического анализатора — *сканер*. Сканер читает литеры (терминальные символы) первоначальной исходной программы и строит лексемы — нетерминальные символы.

Из всего сказанного понятно, что сканер выполняет простой лексический анализ в отличие от синтаксического анализа. Ввиду простоты реализации алгоритма лексического анализа, лексический анализатор все-таки отделяют от значительно более сложного синтаксического анализатора. Такому разделению функций лексического и синтаксического анализа есть еще несколько серьезных причин.

1. Значительная часть времени компиляции тратится на сканирование литер. Выделение операции сканирования позволяет сократить общее время. Один из способов сокращения времени - программирование части или всего сканера на автокоде (и это сделать легче, если сканер выделен). Такой подход хорош для дальнейшего широкого и массового использования компилятора.
2. Синтаксис символов можно описать в рамках очень простых грамматик. Если отделить сканирование от синтаксического разбора, можно разработать эффективную технику разбора, наилучшим образом учитывающую особенности грамматик, в том числе автоматические методы разбора.
3. Так как сканер выдает символы-лексемы вместо литер (терминальных символов — символов исходного текста), синтаксический анализ на каждом шаге получает больше информации о том, что нужно делать.
4. Развитие языков высокого уровня требует внимательного отношения как к лексическим, так и к синтаксическим свойствам языка. Разделение этих свойств позволяет исследовать их независимо друг от друга.
5. Часто для одного и того же языка существует несколько различных внешних проявлений. Например, в некоторых реализациях языка программирования АЛГОЛ служебные слова заключены в кавычки и пробелы не играют никакой роли (игнорируются). В других компиляторах служебные слова не могут использоваться как идентификаторы, и идентификаторы должны отделяться друг от друга, по крайней мере, одним пробелом.

Выделение сканера позволяет написать один синтаксический анализатор и несколько сканеров. (по одному на каждое представление исходной программы и/или устройство ввода/вывода). При этом каждый сканер переводит символы в одинаковую внутреннюю форму, используемую синтаксическим анализатором.

Сканер можно запрограммировать как отдельный проход, на котором выполняется полный лексический анализ исходной таблицы и который выдает синтаксическому анализатору таблицу, содержащую исходную программу в форме внутренних символов.

В другом варианте может быть создана программа сканера, назовем её SCAN, к которой обращается синтаксический анализатор всякий раз, как ему необходим новый символ. В ответ на каждый вызов программы сканера, SCAN распознает следующий символ программы и отсылает его синтаксическому анализатору. Этот последний вариант лучше, так как нет необходимости конструировать целиком всю внутреннюю программу и хранить её в памяти.

Из выше сказанного понятно, что распознавателем (сканером) является простейший детерминированный автомат, позволяющий определить принадлежность цепочки некоторому языку.

Напомним известный результат из теории автоматов, что наиболее близкими к распознавателям (и весьма наглядными) также являются диаграммы Вирта (рис. 3.3).

Используя вывод о том, что синтаксис большинства языков программирования можно задать в форме регулярной грамматики, приведем пример реализации простой части сканера — распознавателя целого десятичного числа, который можно представить следующим образом:

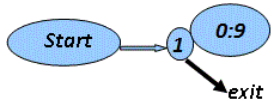


Рис. 3.3. Распознаватель целого десятичного числа

Лексический анализатор построен по принципу конечного автомата. Получив от сканера код, лексический анализатор переходит в новое состояние и/или выполняет какие-либо операции по построению дерева программы. Программа во внутреннем коде представляет дерево кодов. Каждый элемент дерева — узел и/или лист — задается некоторой структурой состояний и действий (операций).

В частности, состояниями лексического анализатора могут быть:

— ***void*** — после перехода от сканера ключевого слова ***void*** , ожидается идентификатор;

— ***main*** — ***ожидается*** пара скобок;

— ***float*** — ***состояние***, при переходе в которое ожидается список объявлений переменных, которые заносятся в (динамический) список идентификаторов с типом ***float***. При этом осуществляется проверка на уникальность имени. Переменные могут быть объявлены в любом месте до их использования. Не учитываются границы видимости и тому подобные характеристики.

Лексический анализатор представляет программу в виде ***дерева с одной веткой***, то есть без иерархий.

Ниже приводится пример работы лексического анализатора, анализирующего выражение ***a+b*c***, в результате работы которого получается следующее дерево разбора (линейное, то есть с одной веткой, рис. 3.4).

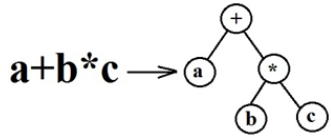


Рис. 3.4. Пример разбора выражения в дерево

Разбивать такое дерево в иерархическое дерево предоставляется *синтаксическому анализатору*.

3.2. Особенности построения и функционирования синтаксических и семантических блоков компиляторов

Синтаксический анализатор, как сегодня говорят, *парсинг*, — это программа или часть программы, выполняющая синтаксический анализ, то есть распознавание входной информации.

Задачей синтаксического анализатора является *полный синтаксический контроль* исходной программы и разбиение её на конструкции исходного языка. Однако, *синтаксический анализ* — более сложный разбор *структуры* программы, чем лексический анализ, и, обычно, применяется совместно с лексическим анализом. При парсинге (синтаксическом анализе) входные данные (полученные от лексического анализатора) преобразуются к виду (структуре данных), пригодному для дальнейшей обработки. Этот преобразованный вид, стандартно, представляет собой формальную модель входной информации для следующего этапа процесса обработки информации (на языке этого последующего этапа).

Сформулируем выше сказанное в терминах формальной теории. *Синтаксический анализ* — это процесс сопоставления линейной последовательности лексем языка его формальной грамматике. С точки зрения теории автоматов под структурой программы понимается *дерево*, соответствующее разбору в КС (контекстно-свободной) грамматике языка. Результатом синтаксического разбора обычно является дерево разбора — синтаксическое дерево. В отличие от результата лексического анализа — дерева с одной ветвью (линейной последовательностью лексем), результатом синтаксического анализа строится синтаксическое дерево — дерево со сложной структурой (дерево с иерархиями, со ссылками на объекты).

На рисунке 3.5 построено синтаксическое дерево вывода для предложения английского языка. Из рисунка видно, что изображенное синтаксическое дерево вывода имеет структуру с иерархиями.

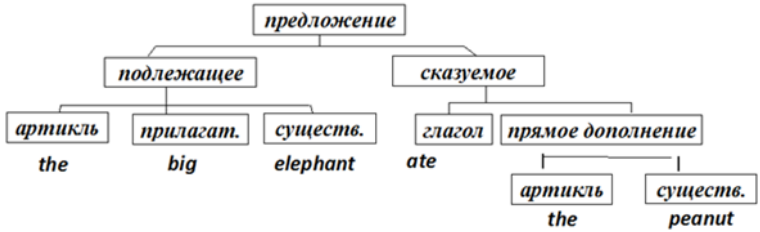


Рис. 3.5. Дерево синтаксического разбора предложения естественного языка

Как уже отмечалось, синтаксический анализ основывается на теории контекстно-свободных (КС) грамматик. Общая форма КС-грамматики не позволяет построить синтаксический анализатор достаточно простым (в частности, автоматически сгенерировать необходимую выходную структуру — синтаксическое дерево). Потому для языков программирования выделены и изучены несколько специальных подклассов КС-языков (LL, LR, LALR), которые проще разбирать и для которых разработаны соответствующие методы разбора. Для реализации автоматического синтаксического анализа структуры языка программирования выбирают соответствующий специальный подкласс.

В настоящее время используется либо LL(1) анализ (и его вариант — рекурсивный спуск), либо LR(1) анализ (и варианты LR(0), SLR(1), LALR(1) и другие).

LL(1) анализ — это синтаксический LL-анализатор (LL parser) — нисходящий синтаксический анализатор для некоторого подмножества контекстно-свободных грамматик, известных как LL-грамматики. При этом не все контекстно-свободные грамматики являются LL-грамматиками. Буквы L в выражении «LL-анализатор» означают, что

— входная строка анализируется слева направо (*left to right*), и при этом
— строится её левосторонний вывод (*leftmost derivation*).

Цифра 1 говорит, что для определения пути разбора нужна всего одна лексема.

LL(1) анализ прост в написании *вручную* без использования автоматических генераторов. Используется для разбора кода в ряде языков программирования (например, Pascal).

LR-анализатор (*LR parser*) — синтаксический восходящий анализатор для некоторого подмножества контекстно-свободных грамматик LR, который читает входной поток слева направо (*Left*) и производит наиболее правое (*Right*) правило контекстно-свободной грамматики. Используется также термин LR(k)-анализатор, где ***k*** выражает количество непрочитанных

символов предпросмотра во входном потоке, на основании которых принимаются решения при анализе. Обычно ***k = 1*** (то есть LR(1)) и часто единица опускается (пишут просто LR).

LR(0), SLR(1), LALR(1) — также восходящие алгоритмы синтаксического разбора. SLR(1) представляет собой расширение алгоритма LR(0).

Более подробно рассмотрим методы нисходящего и восходящего разбора в главе 5.

Отметим только, что рекурсивный спуск LL(1) чаще используется при *ручном* программировании синтаксического анализатора, LR(1) – при использовании *систем автоматизации* построения синтаксических анализаторов.

Рекурсивный спуск очень быстр в исполнении и имеет характерное сообщение об ошибке вида «ожидался такой-то символ».

Простейший способ реагирования на некорректную входную цепочку лексем, переданную лексическим анализатором, — завершить синтаксический анализ и вывести сообщение об ошибке. Однако часто оказывается полезным найти за одну попытку синтаксического анализа как можно больше ошибок. Именно так ведут себя трансляторы большинства распространённых языков программирования.

Таким образом, обработчику ошибок синтаксического анализатора требуется решать следующие задачи:

— ясно и точно сообщать о наличии ошибок;
— обеспечивать быстрое восстановление после ошибки, чтобы продолжать поиск других ошибок;
— не замедлять существенно обработку корректной входной цепочки.

В процессе синтаксического анализа также, как в лексическом анализе, обнаруживаются ошибки, (но значительно более сложные), связанные со структурой программы.

Приведем наиболее известные стратегии восстановления процесса синтаксического анализа после ошибок.

а) Восстановление в режиме паники.

Обнаружив ошибку, синтаксический анализатор пропускает входные лексемы по одной, пока не будет найдена одна из специально определенного множества, так называемых, синхронизирующих лексем. Такими лексемами предпочтительно определить разделители, например, «;», «,», «)» или «}». Набор синхронизирующих лексем определяют разработчики анализируемого языка. Эта стратегия восстановления наиболее проста в реализации, но в ней есть существенный недостаток: при такой стратегии восстановления может оказаться, что значительное количество символов будет пропущено без проверки на наличие дополнительных ошибок.

б) Восстановление на уровне фразы.

В некоторых случаях при обнаружении ошибки синтаксический анализатор может выполнить локальную коррекцию входного потока так, чтобы это позволило ему продолжать работу. Например, перед точкой с запятой, отделяющей различные операторы в языке программирования, синтаксический анализатор может закрыть все ещё не закрытые круглые скобки по своему усмотрению (однако, эта коррекция может не совпадать с логикой программиста). Это более сложный в проектировании и реализации способ, однако, в некоторых ситуациях, он может работать значительно лучше восстановления в режиме паники. Очевидно, рассматриваемая стратегия бессильна, если настоящая ошибка произошла до точки обнаружения ошибки синтаксическим анализатором.

в) Команды регистрации ошибок.

Знание наиболее распространённых ошибок позволяет расширить грамматику языка командами регистрации ошибок, порождающими ошибочные конструкции. При срабатывании таких команд регистрируется ошибка (выдается сообщение об ошибке), но синтаксический анализатор продолжает работать в обычном режиме.

Итак, результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицу символов (рис. 3.6).

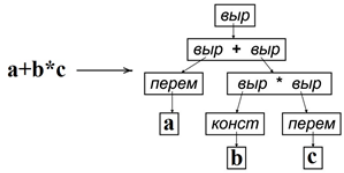


Рис. 3.6. Синтаксическое дерево вывода арифметического выражения

Таблица символов, как и таблица идентификаторов, строится в процессе разных фаз компилятора: лексического, синтаксического, семантического, контекстно-свободного, а используется также во всех фазах, включая генерацию кодов.

Примерный вид таблицы символов приведен ниже (таблица 3.3). Информацию о таблице символов подробнее разберем в главе 4.

Таблица 3.3. Фрагмент таблицы символов

Номер элемента	Идентификатор	Дополнительная информация (тип, размер и т. п.)
1	***i***	идентификатор, тип – “целое”
2	***1***	константа, целое число
3	***N***	идентификатор, тип – “целое”
4	***fg***	идентификатор, тип – “вещественное число”
5	***0.5***	константа, “вещественное число”

Упомянутые выше достижения теории формальных языков в разработке методов лексических и синтаксических анализов насчитывают уже несколько десятков лет, и в этой области не ожидается каких-либо новых эффективных разработок, с учетом того что имеющиеся теоретические результаты достаточно хорошо решают практические задачи.

Иначе обстоит дело с изучением других двух разделов компиляции: семантическим анализом и оптимизацией. Работы по формальному описанию семантики языков программирования интенсивно появлялись в 70-80-х годах XX века. Однако, ни одна из построенных теорий не вошла в широкое употребление при построении компиляторов. Сегодня, как и прежде, семантика языка сначала описывается на естественном языке, а затем реализуется в компиляторе большим набором вручную закодированных условных операторов.

А вот актуальность теории *оптимизирующих преобразований* в последние годы растёт — это объясняется появлением всё более высокоуровневых языков и разнообразных аппаратных платформ. Оптимизация (о которой более подробно будет говориться в главе 13) также требует привлечения развитого математического аппарата (в частности, для решения вопроса о корректности преобразований). Многие из достижений активно внедряются в промышленные компиляторы. Всё больше это касается преобразований, связанных с распараллеливанием.

Семантический анализ

Рассмотрим особенности процесса *семантического* анализа.

На этапе построения семантического дерева продолжается построение таблицы символов: в нее заносятся имена, присутствовавшие в исходном тексте, к ним добавляются такие атрибуты как тип, отведённая память, область видимости.

К семантическим ошибкам относятся

- неправильные типы аргументов функций и операторов;
- несуществующие переменные, метки, функции;
- несуществующие поля объектов, функций;
- неправильно сформированные строки;
- неправильное формирование списка аргументов для функций, принимающих переменное количество аргументов.

Многие современные компиляторы позволяют также выявить сомнительные с точки зрения смысла места в исходной программе. Такими подозрительными на наличие семантической (смысловой) ошибки местами являются

- недостижимые операторы,
- неиспользуемые переменные,
- неопределенные результаты функций
- и тому подобное.

Компилятор указывает такие места в виде предупреждений, которые разработчик может принимать или не принимать во внимание.

Для достижения этой цели компилятор должен иметь представление о том, как программа будет выполняться, и во время компиляции отследить пути выполнения отдельных фрагментов исходной программы - там, где это возможно. Возможности компиляторов по проверке осмысленности предложений входного языка существенно ограничены. Именно поэтому большинство из них в лучшем случае ограничиваются только рекомендациями по тем местам исходного текста программ, которые вызывают сомнения с точки зрения семантики.

Компиляторы обнаруживают только незначительный процент от общего числа смысловых (семантических) ошибок, а, следовательно, подавляющее число такого рода ошибок всегда, к большому сожалению, остается на совести автора программы.

Семантический анализ — это абстрактный структурный анализ. Рассмотрим следующий пример:

cos(sin((int*)("Argument"))) **—** это синтаксически правильная запись, но бессмысленная.

Этот пример показывает, что правильно разобранная программа может содержать множество ошибок контекстного характера, которые компилятором не отслеживаются ни на одном из предыдущих этапах.

Решение проблемы семантического анализа является исключительно инженерной технологией, так как невозможно формировать практическое большинство операций, относящихся к этой фазе.

Приведем ниже некоторые проблемы, которые с трудом решаются на уровне семантического анализа.

1. Области видимости и списки идентификаторов не отслеживаются при объявлении переменных внутри вложенных блоков.

   Для большинства языков (например, Си) допускается объявление переменных внутри вложенных блоков.

   При переопределении имени в таком случае доступ производится по имени, определенному в *самом вложенном блоке*. Но, применение специального квалификатора позволяет иметь доступ к глобальному имени, скрытому за объявлением в локальном блоке имени.
2. Доступ к идентификаторам внутри таблиц обычно осуществляется с помощью метода хеширования. В случае обращения к объектам, структурам, массивам дополнительно проверяется семантическая информация, связанная с правильностью обращения к полям класса, методам класса, правильности индексации массива.
3. Генератору кода и глобальному анализатору с этапа семантического анализа должна подаваться только корректная информация, так как

дополнительные проверки на правильность внутренних структур данных увеличивают в несколько раз время обработки программы и размер; на порядок утяжеляют отладку проходов, распространяя глюки семантического анализатора далее по компилятору.

4. Преобразование типов.

Довольно тривиальные ошибки могут не отлавливаться программистом, например, потеря точности при арифметических сдвигах, или преобразование типов, при которых могут теряться значащие разряды.

if (i=2) вместо ***if (i==2)***

****p=i // flag_bit*** вместо ****p=i / flag_bit***

Семантический анализ приписывает формируемым элементам внутреннего представления атрибуты типа, области видимости и тому подобное и анализирует совместимость этих атрибутов в рамках арифметических выражений и базовых блоков.

Разобранное представление программы со сформированными дополнительными атрибутами передается дальше на этапы оптимизации и генерации кодов.

Контекстно-свободный анализ

На этапе *контекстного анализа* выявляются *зависимости* между частями программы, которые не могут быть описаны контекстно-свободным синтаксисом. Это, в основном, ошибки типа «описание – использование», в частности, анализ *типов* объектов, анализ *областей видимости*, соответствие параметров (в описаниях и вызовах), метки и тому подобное.

В процессе к*онтекстного анализа также пополняется* ***таблица символов,*** которую нужно рассматривать как таблицу имен, пополняемую информацией об описаниях (свойств) объектов.

При контекстном анализе используются атрибутные[13] грамматики. Результатом работы этой фазы является ***атрибутное дерево*** программы. Информация об объектах может быть, как рассредоточена в самом дереве, так и сосредоточена в отдельных таблицах объектов. В процессе контекстного анализа также могут быть обнаружены ошибки, связанные с неправильным использованием объектов.

Затем программа может быть переведена во внутреннее представление. Это делается для целей оптимизации и/или удобства генерации кода. Еще одной целью преобразования программы во внутреннее представление является желание иметь *переносимый компилятор*. Тогда только последняя фаза (генерация кода) является машинно-зависимой. В качестве

[13] *Атрибутная грамматика* – это КС-грамматика (контекстно-свободная грамматика), с узлами дерева вывода которой связаны атрибуты (семантические правила). Атрибут- это свойство объекта; под атрибутом в грамматиках понимаются значение или семантический смысл (значимость) объекта. КС-правилам сопоставляются правила вычисления атрибутов. Правило вычисления значений атрибутов, соответствующее данному КС-правилу, применяется для всех вхождений этого правила в дерево вывода

внутреннего представления может использоваться префиксная или постфиксная запись, ориентированный граф, тройки, четверки и другие.

Далее следуют этапы оптимизации и генерации кодов, которые будем исследовать в дальнейшем.

Большинство грамматик допускают несколько выводов для одной и той же цепочки из языка.

Те или иные фазы транслятора могут отсутствовать или объединяться. В случае однопроходного транслятора (интерпретатора) нет явной фазы генерации промежуточного представления, остальные фазы объединены в одну, причем нет и явно построенного синтаксического дерева.

3.3. Компиляторы в составе систем программирования.

Системой программирования называют комплекс программных средств, предназначенных для кодирования, тестирования и отладки программного обеспечения. Другими словами, это набор специализированных программных продуктов, которые являются инструментальными средствами разработчика.

Система программирования, как правило, включает следующие программные компоненты:

- редактор текста;
- транслятор с соответствующего языка;
- компоновщик (редактор связей, линкер);
- отладчик;
- библиотеки подпрограмм.

Заметим, что любая система программирования может работать только в соответствующей операционной системе (ОС), под которую она и создана, однако при этом она может позволять разрабатывать программное обеспечение и под другие ОС.

Основным модулем системы программирования всегда является компилятор.

Именно характеристики компилятора, прежде всего, влияют на эффективность результирующих программ, порождаемых системой программирования.

Кроме основного компилятора, большинство систем программирования могут содержать в своем составе целый ряд других компиляторов. Так, большинство систем содержат компилятор с языка *Assembler* и компилятор с входного языка описания ресурсов. Но они редко непосредственно взаимодействуют с пользователем.

Компиляторы составляют существенную часть программного обеспечения ЭВМ. Это связано с тем, что языки высокого уровня стали основным средством разработки программ. Только очень незначительная часть программного обеспечения, требующая особой эффективности,

программируется с помощью ассемблеров. В настоящее время распространено довольно много языков программирования. Наряду с традиционными языками, такими, как Фортран, широкое распространение получили так называемые «универсальные» языки (Паскаль, Си, Модула-2, Ада и другие), а также некоторые специализированные (например, язык обработки списочных структур ЛИСП). Кроме того, большое распространение получили языки, связанные с узкими предметными областями, такие, как входные языки пакетов прикладных программ.

Для некоторых языков имеется довольно *много реализаций*. Например, реализаций Паскаля, Модулы-2 или Си для ЭВМ типа IBM PC на рынке десятки.

С другой стороны, постоянно растущая потребность в новых компиляторах связана с бурным *развитием архитектур* ЭВМ. Это развитие идет по различным направлениям. Совершенствуются старые архитектуры как в концептуальном отношении, так и по отдельным, конкретным линиям. Это можно проиллюстрировать на примере микропроцессора *Intel*. Последовательные версии микропроцессоров отличаются не только техническими характеристиками, но и, что более важно, новыми возможностями и, значит, изменением (расширением) системы команд. Естественно, это требует новых компиляторов (или модификации старых).

В рамках традиционных последовательных машин возникает большое число различных направлений архитектур. Примерами могут служить архитектуры CISC, RISC[14]. Такие ведущие фирмы, как *Intel, Motorola, Sun,* начинают переходить на выпуск машин с RISC-архитектурами. Естественно, для каждой новой системы команд требуется полный набор новых компиляторов с распространенных языков.

Наконец, бурно развиваются различные *параллельные архитектуры*. Первыми из них появились векторные, многопроцессорные, с широким командным словом (вариантом которых являются суперскалярные ЭВМ). На рынке уже имеются множество типов ЭВМ с параллельной архитектурой, начиная с классических примеров: супер-ЭВМ (из зарубежных *Cray, CDC,* из отечественных – *Эльбрус,* из более современных – вычислительные комплексы серии СКИФ), мощные мультипроцессорные рабочие станции или ноутбуки, в составе которых

---

[14] **CISC** (англ. complex instruction set computing или complex instruction set computer) — тип процессорной архитектуры, которая характеризуется следующим набором свойств:

- нефиксированное значение длины команды;
- арифметические действия кодируются в одной команде;
- небольшое число регистров, каждый из которых выполняет строго определённую функцию.

Методика построения системы команд CISC противостоит методике, применяемой в другом распространённом типе процессорных архитектур — RISC, где используется набор упрощённых инструкций.

RISC (англ. reduced instruction set computer[1][2] — компьютер с набором коротких (простых, быстрых) команд) — архитектура процессора, в котором быстродействие увеличивается за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения — меньшим. Первые RISC-процессоры даже не имели инструкций умножения и деления. Это также облегчает повышение тактовой частоты и делает более эффективной суперскалярность (распараллеливание инструкций между несколькими исполнительными блоками).

несколько процессоров. Естественно, для каждого из комплексов создаются новые компиляторы для многих языков программирования. Здесь необходимо также отметить, что новые архитектуры требуют разработки совершенно новых подходов к созданию компиляторов, так что наряду с собственно разработкой компиляторов ведется и большая научная работа по созданию новых методов трансляции.

Вопросы для контроля

1. Назовите этапы, составляющие блок анализа процесса компиляции. Охарактеризуйте входные и выходные потоки каждого из этих этапов.
2. Дайте определение лексемы. Какие элементы языка программирования соответствуют лексеме?
3. Из каких соображений выделяют лексический анализатор в отдельный проход?
4. Какие ошибки выявляет семантический анализатор?
5. Какие методы синтаксического разбора используются для анализа последовательности лексем?

Глава 4. Моделирование преобразований внутреннего представления лексем.

Учебные цели:

- дать представление о внутреннем представлении и преобразовании входных потоков;
- дать представление об идентификации лексических единиц языков программирования.

Прежде чем рассматривать преобразования потоков лексем в процессе компиляции, кратко напомним составные части компилятора и их функциональность.

Итак, в состав компилятора входят следующие программы.

- *Драйвер* (монитор, диспетчер) — управляющая программа компилятора, последовательно вызывающая части компилятора.
- *Препроцессор* — программа, выполняющая моделирование данных (предобработку) для подготовки ввода в исходный код программы (например, переформатирование кода или вставка макрорасширения, библиотеки с помощью инструкций ***#define, #include***).
- *Анализатор*, выполняющий
  - лексический анализ (поток символов исходного кода разбивается на лексемы конструкций языка);
  - синтаксический анализ (разбор структуры программы и преобразование последовательности лексем в дерево зависимостей с иерархиями);
  - семантический анализ (более сложный разбор структуры программы, например, проверка соответствия количества параметров и их типов в описании и вызове функций и тому подобное);
  - контекстный анализ (выявление зависимостей между частями программы, например, анализ типов, областей видимости, соответствия параметров и т.п.)
- *Оптимизатор* — преобразует программу с целью улучшения её характеристик по одному или нескольким параметрам.
- *Кодогенератор* — порождает на основе промежуточного представления ассемблерный или объектный код.

Отметим, что на протяжении всего процесса компиляции

- создаются, дополняются, взаимодействуют и используются многие вспомогательные ресурсы, необходимые для выполнения компиляции, например, таблица лексем, таблица идентификаторов, таблица символов, интерфейсы, в частности, интерфейс с пользователем и с операционной системой, обработчик сообщений об ошибках, системные и пользовательские библиотеки;

— анализируются, преобразуются потоки, поступающие на вход каждого из перечисленных выше блоков компилятора, в форму, удобную для последующей обработки непосредственно следующим блоком.

На рисунке 4.1 представлены три возможных подхода проектирования компилятора с учетом структуры входных и выходных потоков данных для каждого блока компилятора.

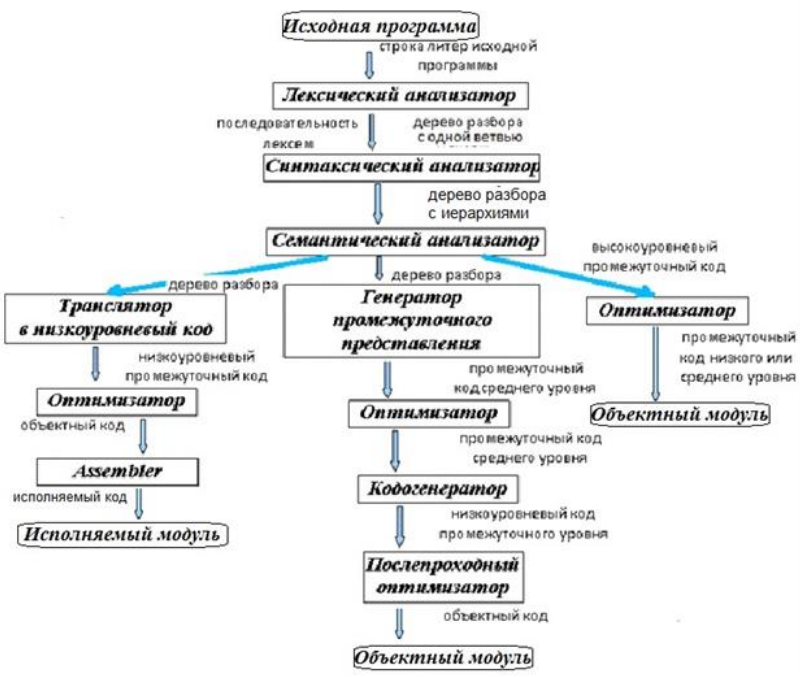


Рис. 4.1. Три подхода при проектировании компилятора

Не всегда область разработок заказчика (например, используемый язык программирования) требует задействовать для разработки компилятора все теоретически рассматриваемые блоки (средняя группа блоков на рис. 4.1). В некоторых случаях после работы анализатора достаточно задействовать блок анализатора и получить объектный модуль (правая ветвь на рис. 4.1) или применить транслятор низкоуровневого кода (когда проектируется проблемно-ориентированный компилятор с разработкой промежуточного языка) с последующей оптимизацией и ассемблированием для получения в результате исполняемого кода (левая ветвь рис. 4.1).

При этом для общности достаточно подробно рассмотреть функциональность основного подхода (средней линии рис. 4.1), в котором представлены все блоки и, соответственно все структуры потоков данных.

4.1. Изучение видов внутреннего представления и преобразования выходных потоков (грамматики, графы, синтаксические деревья) с помощью операций над строками.

Способы внутреннего представления программ.

Прежде чем обсуждать вопросы внутреннего представления и преобразования выходных потоков компилятора на разных этапах компиляции, рассмотрим структуру и назначение одной из важных таблиц, участвующих во всех этапах компиляции.

Таблица символов — это важная структура данных, создаваемая на этапе лексического анализа, дополняемая на этапе синтаксического анализа и поддерживаемая компилятором для отслеживания семантики переменной, то есть она хранит информацию об области видимости и привязке информации об именах, информацию об экземплярах различных объектов, таких как имена переменных и функций, классы, объекты и так далее. Затем эта таблица используется фазами синтеза компилятора для генерации кода для достижения эффективности времени компиляции.

В таблице символов хранится информация о следующих объектах:

- именах и константах переменных,
- именах процедур и функций,
- литеральных константах и строках,
- временных объектах,
- метках на исходных языках.

Перечислим основные действия с этой таблицей, которые происходят на последовательных этапах компиляции.

1. На этапе *лексического анализа* создаются новые записи в таблице символов, например, записи о лексеме.
2. На этапе *синтаксического анализа* в таблицу символов добавляется информация о следующих характеристиках:
   - типе атрибута,
   - области действия,
   - измерении,
   - линии ссылки,
   - использовании и так далее.
3. На этапе семантического анализа используется доступная информация из таблицы символов для проверки семантики, то есть для проверки того, что выражения и присваивания являются семантически правильными (проверка типов), и соответственно обновляет таблицу символов.
4. В случае использования *промежуточного кода* корректирует таблицу символов для сохранения информации о создаваемой

вспомогательной временной переменной: о том, сколько и чего выделяется (тип, время выполнения).

5. На этапе *оптимизации кода* используется информация, представленная в таблице символов, для машинно-зависимой оптимизации.
6. На этапе *генерации целевого кода* для генерирования кода используется адресная информация идентификатора, присутствующего в таблице символов.

Рассмотрим более подробно составляющие элементы схемы на рисунке 4.1 (основной подход, средняя ветвь).

Исходная программа рассматривается лексическим анализатором как единая непрерывная длинная последовательность символов (литер, терминальных символов, которые можно набрать на клавиатуре). На рисунке 4.1 входной поток для лексического анализатора обозначен как «поток литер исходной программы».

В главе 2 был подробно описан процесс лексического анализа, рассмотрены два режима его работы. В нашем случае, не нарушая общности, будем считать, что лексический анализатор работает в таком режиме, что на выходе получается *полный* набор лексем.

Там же было выяснено, что, анализируя исходный текст, который представляет с точки зрения лексического анализатора не что иное, как длинную строку символов, лексический анализатор преобразует эту строку символов в последовательность лексем (нетерминальных символов) и организует соответствующие таблицы (таблицу символов, таблицу идентификаторов), содержащие необходимую для дальнейшей работы информацию об этих лексемах. Понятно, что для программирования анализа входного потока символов потребуются функции работы с символами и строками.

Таким образом, выходной поток лексического анализатора представляет собой последовательность таких объектов, как различного рода константы и идентификаторы (в том числе и ключевые слова языка), сопоставленные лексемам, с информацией о типах этих лексем, которая содержится в специальных таблицах. Лексемам, в свою очередь, сопоставлены уникальные коды, зависящие от типа лексемы.

То есть, в фазе лексического анализа входная программа разбивается на лексемы в соответствии с определениями языка.

Язык констант и идентификаторов является регулярным, а это означает, что может быть описан с помощью регулярных грамматик. В таком случае, как мы знаем из первой части учебного пособия, распознавателями для регулярных языков являются детерминированные конечные автоматы. Значит, основой для реализации лексических анализаторов следует использовать регулярные грамматики и конечные автоматы.

Итак, в основе реализации лексических анализаторов лежит применение теории автоматов и регулярных выражений.

Результатом работы лексического анализатора является полный последовательный перечень всех найденных в тексте исходной программы лексем. Этот перечень лексем можно представить в виде таблицы, называемой таблицей лексем, с одной стороны, и в виде дерева с одной ветвью, с другой. Лексемы в таблице лексем могут повторяться столько раз, сколько их присутствует в выходном потоке лексем, причем в том же порядке, как они встречаются в выходной последовательности.

Приведем пример работы лексического анализатора (рис. 4.2). Верхняя строка в примере — исходный текст (входной поток для лексического анализатора), нижняя строка – имитация дерева с одной ветвью.

Пример 4.1. *Дерево вывода с одной ветвью*

for (i=1; i<N; i++) {fg=fg*0.5;}

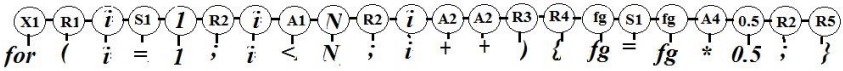


Рис. 4.2. Дерево с одной ветвью (без иерархий)

На вход синтаксического анализатора подается выходной поток, полученный в результате работы лексического анализатора.

Синтаксический анализ — это процесс сопоставления линейной последовательности лексем языка (входного потока синтаксического анализатора) его формальной грамматике.

В теории автоматов было показано, что структура языков программирования соответствует КС (контекстно-свободной) формальной грамматике, а результатом синтаксического разбора обычно является дерево разбора — синтаксическое дерево — дерево со сложной структурой (дерево с иерархиями, со ссылками на объекты).

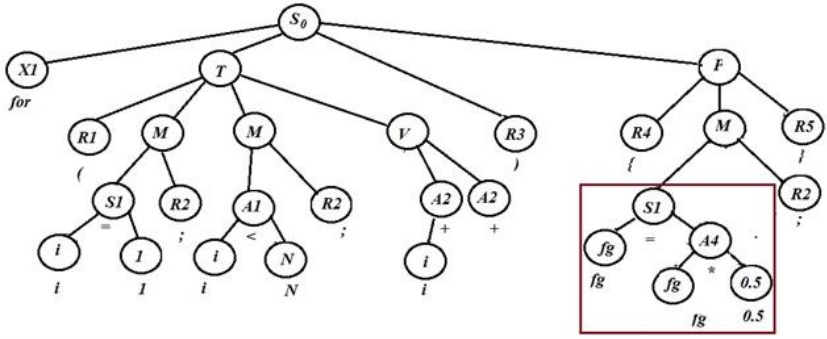


Рис. 4.3. Стилизованное синтаксическое дерево с иерархиями

Синтаксический анализатор производит разбор структуры программы с учетом информации из таблиц, построенных на этапе лексического анализатора, в частности, со ссылками на таблицу символов, и строит

синтаксическое дерево сложной структуры (с иерархиями). Полученная информация (синтаксическое дерево) представляет выходной поток синтаксического анализатора. Стилизованный граф в виде дерева с иерархиями (или зависимостями) для примера предыдущего этапа, изображенного на рисунке 4.2 представлен на рисунке 4.3.

Рассмотрим простой блок этой схемы (блок выделен рамкой), он соответствует разбору оператора ***fg=fg+0.5***.

Далее на этапе семантического анализа происходит дальнейший более сложный разбор структуры программы, а именно, проверка соблюдения во входной программе семантических соглашений входного языка, то есть сопоставлении входных цепочек программы с требованиями *семантики* входного языка программирования. Каждый язык программирования имеет четко заданные и специфицированные семантические соглашения, которые не могут быть проверены на этапе синтаксического разбора. Именно их в первую очередь проверяет семантический анализатор. Подробно этап семантического анализа будет разобран в главе 5.

При этом выходной поток этапа семантического анализа преобразуется следующим образом.

Внутреннее представление программы дополняется операторами и действиями, *неявно* предусмотренными семантикой входного языка, связано с преобразованием типов операндов в выражениях и при передаче параметров в процедуры и функции. Вызовы функций преобразования типов, например, будут встроены в текст результирующей программы для удовлетворения семантических соглашении о преобразованиях типов во входном языке, хотя в тексте программы в явном виде они не присутствуют.

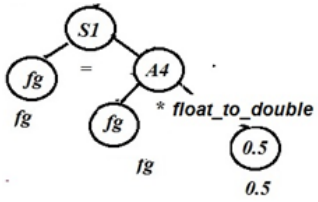


Рис. 4.4. Дерево вывода фрагмента после этапа семантического анализа

Так, для выделенного блока, например, может быть добавлена информация о преобразовании типов, например, если переменная ***fg*** была определена как ***double*** (рис. 4.4).

Важную роль в преобразовании внутренних потоков имеют вспомогательные таблицы, в частности, таблица символов.

Но формы представления, используемые на этапах синтаксического, семантического анализа, оказываются неудобными в работе при генерации и оптимизации объектного кода. Поэтому перед оптимизацией и непосредственно

перед генерацией объектного кода внутреннее представление программы может преобразовываться в одну из соответствующих форм записи:

- связочные списочные структуры, представляющие синтаксические деревья;
- многоадресный код с явно именуемым результатом (тетрады);
- многоадресный код с неявно именуемым результатом (триады);
- обратная (постфиксная) польская запись операций;
- ассемблерный код или машинные команды.

Все внутренние представления программы обычно содержат в себе две принципиально различные вещи — операторы и операнды. Различия между формами внутреннего представления заключаются лишь в том, как операторы и операнды соединяются между собой. Также операторы и операнды должны отличаться друг от друга, если они встречаются в любом порядке. За различение операндов и операторов, как уже было сказано выше, отвечает разработчик компилятора, который руководствуется семантикой входного языка.

В каждом конкретном компиляторе может использоваться одна из этих форм, выбранная разработчиками. Но чаще всего компилятор не ограничивается использованием только одной формы для внутреннего представления программы. На различных фазах компиляции могут использоваться различные формы, которые по мере выполнения проходов компилятора преобразуются одна в другую.

В этой главе из предложенного списка выше рассмотрим основные принципы работы с такой формой, как синтаксические деревья, подробно рассмотрим синтаксические деревья в главе 9. Остальные формы, связанные с оптимизацией и генерацией кодов подробно рассматривать также не будем, так как они применяются при глобальной оптимизации и генерации кодов, и этим формам будет посвящена глава 13.

Синтаксические деревья или связочные списочные структуры — это структура, представляющая собой результат работы синтаксического анализатора. Она отражает синтаксис конструкций входного языка и явно содержит в себе полную взаимосвязь операций. Отметим также, что синтаксические деревья – это *машинно-независимая* форма внутреннего представления программы.

Недостаток синтаксических деревьев заключается в том, что они представляют собой сложные связные структуры, а потому не могут быть *тривиальным* образом преобразованы в линейную последовательность команд результирующей программы. Тем не менее, они удобны при работе с внутренним представлением программы на тех этапах, когда нет необходимости непосредственно обращаться к *командам* результирующей программы.

Синтаксические деревья могут быть преобразованы в другие формы внутреннего представления программы, представляющие собой линейные списки, с учетом семантики входного языка. Эти преобразования

выполняются на основе принципов *синтаксически управляемой* компиляции (СУ-компиляции[15], СУ-перевода).

Возможна модель компилятора, в которой синтаксический анализ исходной программы и генерация кода результирующей программы объединены в одну фазу. Модель такого типа можно представить в виде компилятора, у которого операции *генерации кода* совмещены с операциями выполнения *синтаксического разбора*. Для описания компиляторов такого рода часто используется термин синтаксически управляемая компиляция (СУ-компиляция).

Идея проектирования синтаксически управляемого перевода (СУ-перевода) основана на том, что синтаксис и семантика языка взаимосвязаны. Это значит, что смысл предложения языка зависит от синтаксической структуры этого предложения.

Теория синтаксически управляемого перевода была предложена американским лингвистом Ноамом Хомским. Она справедлива как для формальных языков, так и для естественных языков. Например, смысл предложения русского языка зависит от входящих в него частей речи (подлежащего, сказуемого, дополнений и других частей речи) и от взаимосвязей между ними.

Однако естественные языки допускают *неоднозначности* в грамматиках — поэтому существуют различные двусмысленные фразы, значение которых человек обычно понимает из того *контекста*, в котором эти фразы встречаются (хотя и человек не всегда может тот смысл, который был во фразу вложен).

В языках программирования *неоднозначности* в грамматиках *исключены*, поэтому любое предложение языка имеет четко определенную структуру и *однозначный смысл*, напрямую связанный с этой структурой.

Входной язык компилятора имеет бесконечное множество допустимых предложений, поэтому невозможно задать смысл каждого предложения. Но все входные предложения строятся на основе конечного множества правил грамматики, которые всегда можно найти. Так как этих правил *конечное число*, то для каждого правила можно определить его семантику (значение).

Абсолютно то же самое можно утверждать и для *выходного* языка компилятора. Выходной язык содержит бесконечное множество допустимых предложений, но все они строятся на основе *конечного множества* известных правил, каждое из которых имеет определенную семантику (смысл). Если по отношению к исходной программе компилятор выступает в роли распознавателя, то для результирующей программы он является генератором предложений выходного языка.

[15] СУ-компиляция (СУ-перевод — синтаксически управляемый перевод) - метод построения кода результирующей программы для синтаксической конструкции входного языка.

Задача заключается в том, чтобы найти *порядок* правил выходного языка, по которым необходимо выполнить генерацию.

Обобщая выше сказанное, можно сделать вывод, что идея СУ-перевода заключается в том, что каждому правилу входного языка компилятора сопоставляется одно или несколько (или ни одного) правил выходного языка в соответствии с семантикой входных и выходных правил. То есть при сопоставлении надо выбирать правила выходного языка, которые несут тот же смысл, что и правила входного языка.

Таким образом, СУ-перевод — это основной метод порождения кода результирующей программы на основании результатов синтаксического анализа. Для удобства понимания сути метода можно считать, что результат синтаксического анализа представлен в виде дерева синтаксического анализа, хотя в реальных компиляторах это не всегда так.

Объясним более конкретно принцип СУ-перевода. Он заключается в следующем:

— с каждой вершиной дерева синтаксического разбора ***N*** связывается цепочка некоторого промежуточного кода ***C(N)***,

— код для вершины ***N*** строится путем сцепления (конкатенации) в фиксированном порядке последовательности кода ***C(N)*** и последовательностей кодов, связанных со всеми вершинами, являющимися прямыми потомками ***N***,

— в свою очередь, для построения последовательностей кода прямых потомков вершины ***N*** потребуется найти последовательности кода для их потомков (потомков второго уровня вершины ***N***) и так далее.

Таким образом, процесс перевода происходит снизу вверх в строго установленном порядке, определяемом структурой дерева.

Для того, чтобы построить СУ-перевод по заданному дереву синтаксического разбора, необходимо найти последовательность кода для корня дерева. Поэтому для каждой вершины дерева порождаемую цепочку кода надо выбирать таким образом, чтобы код, приписываемый корню дерева, оказался искомым кодом для всего оператора, представленного этим деревом.

В общем случае необходимо иметь единообразную интерпретацию кода ***C(N)***, которая бы встречалась во всех ситуациях, где присутствует вершина ***N***.

В принципе, эта задача может оказаться нетривиальной, так как требует оценки смысла (семантики) каждой вершины дерева. При применении СУ-перевода задача оценки смысловой нагрузки для каждой вершины дерева решается разработчиком компилятора.

Однако, схему СУ-компиляции можно реализовать не для всякого входного языка программирования. Если принцип СУ-перевода применим ко всем входным КС-языкам, то *применить СУ-компиляцию оказывается не всегда возможным*.

В процессе СУ-перевода или СУ-компиляции не только вырабатываются цепочки текста выходного языка, но и совершаются некоторые дополнительные действия, выполняемые самим компилятором. В общем случае схемы СУ-перевода могут предусматривать выполнение следующих действий:

- помещение в выходной поток данных машинных кодов или команд ассемблера, представляющих собой результат работы (выход) компилятора;
- выдача пользователю сообщений об обнаруженных ошибках и предупреждениях (которые должны помещаться в выходной поток, отличный от потока, используемого для команд результирующей программы);
- порождение и выполнение команд, указывающих, что некоторые действия должны быть произведены самим компилятором (например, операции, выполняемые над данными, размещенными в таблице идентификаторов).

4.2. Идентификация лексических единиц языков программирования.

Идентификация переменных, типов, процедур, функций и других лексических единиц языков программирования — это установление однозначного соответствия между данными объектами и их именами в тексте исходной программы. Идентификация лексических единиц языка чаще всего выполняется на этапе семантического анализа.

Как правило, в большинстве языков программирования существует правило, которое требуют, чтобы в исходной программе имена лексических единиц не совпадали ни с

— именами других лексических единиц,
— ключевыми словами синтаксических конструкций языка.

Однако, чаще всего, этого требования бывает недостаточно для того, чтобы установить однозначное соотношение между лексическими единицами и их именами, потому что существуют дополнительные смысловые ограничения, накладываемые языком на употребление эти имен.

Например, локальные переменные в большинстве языков программирования имеют область видимости, и эта область видимости ограничивает употребление имени переменной рамками того блока исходной программы, где эта переменная описана.

Это означает, что,

— с одной стороны, такая переменная не может быть использована вне пределов своей области видимости;
— с другой стороны, имя переменной может быть *не уникальным*, поскольку в двух различных областях видимости допускается существование двух различных переменных с одинаковыми именами.

Полный перечень таких ограничений зависит от семантики конкретного языка программирования. Все они четко заданы в описании языка и не могут допускать неоднозначности в толковании. Однако, они также не могут быть *полностью определены* на этапе лексического разбора, а поэтому требуют от компилятора дополнительных действий на этапах синтаксического разбора и семантического анализа.

Целью включения этих дополнительных действий в процесс анализа обеспечить каждой лексической единице языка *уникальное имя* в пределах *всей исходной программы* и потом использовать это имя при синтезе результирующей программы.

Можно привести примерный перечень действий компиляторов (требований), которые должны быть реализованы для идентификации переменных, констант, функций, процедур и других лексических единиц языка:

- имена локальных переменных дополняются именами тех блоков (функций, процедур), в которых эти переменные описаны (например, блок поименован ***1***, а переменная ***t*** , описаная в блоке, переменная будет храниться, как ***t_1***);
- имена внутренних переменных и функций модулей исходной программы дополняются именем самих модулей, причем это касается только внутренних имен и не должно происходить, если переменная или функция доступна извне модуля (аналогично предыдущему пункту: если функция имеет имя ***f1***, а переменная имеет локальное имя в функции ***t***, переменная будет храниться как ***t_f1***);
- имена процедур и функций, принадлежащих объектам (классам), в объектно-ориентированных языках программирования дополняются наименованием типа объекта (класса), которому они принадлежат (аналогично предыдущим пунктам);
- имена процедур и функций модифицируются в зависимости от типов их формальных аргументов.

Конечно, это далеко не полный перечень возможных действий компилятора, каждая реализация компилятора может предполагать свои набор действий. То, какие из них будут использоваться и как они будут реализованы на практике, зависит от языка исходной программы и разработчиков компилятора.

Как правило, уникальные имена, которые компилятор присваивает лексическим единицам языка, используются компилятором только *во внутреннем представлении* исходной программы, и человек, создавший исходную программу, не сталкивается с ними.

Но они могут потребоваться пользователю в некоторых случаях, например,

- при отладке программы,

— при порождении текста результирующей программы
  - на языке ассемблера или
  - при использовании библиотеки, созданной версией компилятора для одного языка программирования на другом языке (или даже просто в другой версии компилятора).

Тогда пользователь должен знать, по каким правилам компилятор порождает уникальные имена для лексических единиц исходной программы.

Во многих современных компиляторах (и обрабатываемых ими входных языках) предусмотрены специальные настройки и ключевые слова, которые позволяют отключить процесс порождения компилятором уникальных имен для лексических единиц языка. Эти слова учтены в специальных синтаксических конструкциях языка (как прилило, это конструкции, содержащие слона ***export*** пли ***external*****).**

Если пользователь использует эти средства, то компилятор не применяет механизм порождения уникальных имен для указанных лексических единиц. В этом случае разработчик программы сам отвечает за уникальность имени данной лексической единицы в пределах всей исходной программы или даже в пределах всего проекта.

Если требование уникальности не будет выполняться, могут возникнуть

— синтаксические или семантические ошибки на стадии компиляции,
— либо же другие ошибки на более поздних этапах разработки программного обеспечения.

Поскольку наиболее широко используемыми лексическими единицами в различных языках программирования являются, как правило, имена процедур и функций, то этот вопрос, прежде всего, касается именно этих объектов.

Вопросы для контроля

1. Какая информация хранится в таблице символов?

2. Какую структуру имеют входной и выходной потоки этапа синтаксического анализа? Чем отличаются деревья вывода?

3. Опишите алгоритм построения синтаксического дерева.

4. Какие теоретические исследования применяются при проектировании лексического анализатора, синтаксического анализатора?

5. Какие требования предъявляются к проектированию компиляторов, чтобы обеспечить уникальность лексем?

Глава 5. Семантический анализ и подготовка к генерации кода программы

Учебные цели:

- дать представление о назначении семантического анализа;
- дать представление о методах нисходящего и восходящего разборов, идентификации лексических единиц языков программирования.

5.1. Назначение семантического анализа.

Практически вся языки программирования, строго говоря, не являются КС-языками (контекстно-свободными языками). Поэтому, как это показано в первой части учебного пособия («Формальные грамматики») полный разбор цепочек символов входного языка (текста программы) компилятор не может выполнить в рамках КС-языков с помощью КС-грамматик и МП-автоматов (автоматов с магазинной памятью). Полный распознаватель для большинства языков программирования может быть построен в рамках КЗ-языков (контекстно-зависимых языков), поскольку все реальные языки программирования шире КС-грамматик, а именно, контекстно-зависимы.

Однако известно, что для выполнения разбора входной цепочки длины ***n*** такой распознаватель имеет экспоненциальную зависимость вычислительных ресурсов, требуемых для распознавания. Компилятор, построенный на основе такого распознавателя, будет неэффективным с точки зрения либо скорости работы, либо объема необходимой памяти.

Поэтому такие компиляторы практически не используются, а все реально существующие компиляторы на этапе разбора входных цепочек проверяют только синтаксические конструкции входного языка, не учитывая его семантику.

5.2. Этапы семантического анализа.

С целью повышения эффективности компиляторов разбор цепочек входного языка выполняется в два этапа:

- первый — синтаксический разбор на основе распознавателя одного из известных классов КС-языков;
- второй — семантический анализ входной цепочки.

Для проверки семантической правильности входной программы необходимо иметь всю информацию о найденных лексических единицах языка. Эта информация помещается в таблицу лексем на основе конструкций, найденных синтаксическим распознавателем.

Примерами таких конструкциями являются блоки описания констант и идентификаторов (если они предусмотрены семантикой языка) пли операторы, где тот или иной идентификатор встречается впервые (если описание происходит по факту первого использования). Поэтому полный семантический анализ входной программы может быть произведен только после полного завершения её синтаксического разбора.

Таким образом, входными данными для семантического анализа служат:

— таблица идентификаторов;

— результаты разбора синтаксических конструкций входного языка.

Результаты выполнения синтаксического разбора могут быть представлены в одной из форм внутреннего представления программы в компиляторе. Как правило, на этапе семантического анализа используются различные варианты *деревьев синтаксического разбора*, поскольку семантический анализатор интересует прежде всего *структура входной программы*.

Семантический анализ обычно выполняется на двух этапах компиляции:

— на этапе синтаксического разбора и

— в начале этапа подготовки к генерации кода.

В первом случае всякий раз по завершении распознавания определенной синтаксической конструкции входного языка выполняется её семантическая проверка на основе имеющихся в таблице идентификаторов данных (такими конструкциями, как правило, являются процедуры, функции и блоки операторов входного языка).

Во втором случае, после завершения всей фазы синтаксического разбора, выполняется полный семантический анализ программы на основании данных в таблице идентификаторов (сюда попадает, например, поиск неописанных идентификаторов).

Иногда семантический анализ выделяют в отдельный этап (фазу) компиляции.

В каждом компиляторе обычно присутствуют оба варианта семантического анализатора.

Семантический анализатор выполняет следующие основные действия:

— проверка соблюдения во входной программе семантических соглашений входного языка;

— дополнение внутреннего представления программы в компиляторе операторами и действиями, неявно предусмотренными семантикой входного языка;

— проверка элементарных семантических (смысловых) норм языков программирования, напрямую не связанных с входным языком.

Проверка соблюдения во входной программе семантических соглашений входного языка заключается в сопоставлении входных

цепочек программы с требованиями семантики входного языка программирования. Каждый язык программирования имеет четко заданные и специфицированные семантические соглашения, которые не могут быть проверены на этапе синтаксического разбора. Именно их в первую очередь проверяет семантический анализатор.

Примерами таких соглашении являются следующие требования:

— каждая *метка*, на которую есть ссылка, должна один раз присутствовать в программе;
— каждый *идентификатор* должен быть описан один раз, и ни один идентификатор не может быть описан более одного раза (с учетом блочной структуры описаний);
— все *операнды* в выражениях и операциях должны иметь типы, допустимые для данного выражения или операций;
— *типы переменных* в выражениях должны быть согласованы между собой;
— при вызове процедур и функций число и *типы фактических параметров* должны быть согласованы с числом и типами формальных параметров.

Например, если оператор языка Си имеет вид

a = b + c;

то, с точки зрения *синтаксического разбора,* как Вы видите, это будет абсолютно правильный оператор.

Однако, нельзя положительно ответить, является ли этот оператор правильным с точки зрения входного языка (Си), пока не провести проверку *семантических требований* для всех входящих в него лексических элементов. Такими элементами здесь являются идентификаторы ***a, b*** и ***c***.

Если неизвестно, что собой представляют идентификаторы ***a, b*** и ***c***, нельзя с уверенностью утверждать не только тот факт, что приведенный выше оператор является правильным, но и, вообще, имеет ли он смысл. Фактически необходимо знать *описание* идентификаторов, входящих в состав оператора.

1. Если хотя бы один из идентификаторов не описан, компилятор должен выдать сообщение об *явной ошибке*.

2. Прежде всего, эти идентификаторы должны обозначать объекты сопоставимых типов:

— если это числовые переменные и константы, то «плюс» означает, что должен выполняться оператор сложения,
— если же это строковые переменные или константы, то «плюс» — это оператор конкатенации строк.

3. Кроме того, обратите внимание, что идентификатор, стоящий слева от символа «=», то есть, идентификатор ***a***, например, ни в коем случае *не может быть константой*, так как в этом случае будет нарушена семантика оператора присваивания.

4. Невозможна такая ситуация, когда одни из идентификаторов являются числовыми идентификаторами, а другие — строчными (или символьными), или, к примеру, идентификаторами массивов или структур, — такое сочетание аргументов для оператора сложения недопустимо.

5. Следует также отметить, что от семантических соглашений зависит не только *правильность* оператора, но и его *смысл*.

Простейший пример: операции алгебраического сложения и конкатенации строк имеют различный смысл, хотя и обозначаются в приведённом выше примере

a = b + c;

одним и тем же знаком "+". Это означает, что от результата семантического анализатора этого оператора зависит код его результата.

Таким образом, если хотя бы одно из семантических требований входного языка не выполняется, то компилятор обязан выдать сообщение об ошибке и процесс компиляции на этом, как правило, прекратить.

6. Рассмотренные выше пять пунктов демонстрируют явные нарушения семантических требований входного языка. Однако, существуют и неявные нарушения. Например, дополнение внутреннего представления программы операторами и действиями, *неявно предусмотренными* семантикой входного языка, может быть связано с преобразованием типов операндов в выражениях, а также преобразованием типов операндов при передаче параметров в процедуры и функции.

Если вернуться к рассмотренному выше элементарному оператору языка Си: ***a = b + c;***

то можно отметить, что здесь выполняются две операции:

— одна операция сложения (или конкатенации, в зависимости от типов операндов) и

— одна операция присвоения результата.

Соответствующим образом будет порожден и код результирующей программы.

Однако все просто лишь в случае если все идентификаторы имеют одинаковые типы. Если же допустить, что где-то перед рассмотренным оператором имеется описание его операндов в следующем виде:

```
float a;
int b;
double c;
```

Из этого описания следует, что

a - вещественная переменная языка Си,

b - целочисленная переменная,

c - вещественная переменная с двойной точностью.

Так как в языке Си нельзя непосредственно выполнять операции над операндами *различных типов*, смысл рассмотренного оператора с точки зрения входной программы существенным образом меняется.

Необходимо, чтобы существовали правила преобразования типов, принятые для данного языка, и были реализованы механизмы неявного преобразования этих типов.

Один путь — реализовывать эти преобразования будет разработчик программы. Тогда преобразования типов *в явном виде* должны будут присутствовать в тексте входной программы (но в рассмотренном примере это не так, мы не видим применения каких-либо функций для приведения операндов разного типа к одному).

Второй путь — делегировать выполнение неявных преобразований типов некоему коду, порождаемому компилятором (при котором преобразования типов в явном виде в тексте программы не присутствуют, но неявно предусмотрены семантическими соглашениями языка).

Для этого в составе библиотек функций, доступных компилятору, должны присутствовать *функции преобразования типов*. Вызовы этих функции как раз и будут встраиваться в текст результирующей программы для удовлетворения семантических соглашении о преобразованиях типов во входном языке, хотя в тексте программы в явном виде они не будут присутствовать. Чтобы этот механизм функционировал, функции преобразования типов должны быть встроены и во внутреннее представление программы в компиляторе. За реализацию описанного механизма отвечает именно семантический анализатор.

Таким образом, для рассматриваемого примера с учетом предложенных типов данных нужны будут не две, а четыре операции:

— преобразование целочисленной переменной ***b*** в формат вещественных чисел с двойной точностью;
— сложение двух вещественных чисел с двойной точностью;
— преобразование результата в вещественное число с одинарной точностью;
— присвоение результата переменной ***c***.

Количество операций увеличилось вдвое, причем добавились нетривиальные функции преобразования типов. Преобразование типов — эго только один вариант операций, неявно добавляемых компилятором в код программы на основе семантических соглашении.

В качестве еще одного рода операций могут служить операции вычисления адреса, когда происходит *обращение* к элементам *сложных структур данных*.

Существуют и другие варианты такого рода операций.

Таким образом, действия, выполняемые семантическим анализатором, существенным образом влияют на порождаемый компилятором код результирующей программы.

Проверка элементарных смысловых норм языков программирования, напрямую не связанных с входным языком, — это сервисная функция, которую предоставляют большинство современных компиляторов. Эта функция обеспечивает проверку компилятором некоторых соглашений, применимых к большинству современных языков программирования, выполнение которых связано со смыслом как всей входной программы в целом, так и отдельных её фрагментов.

5.3. Методы нисходящего и восходящего разборов: метод рекурсивного спуска и метод предшествования.

Методы синтаксического разбора.

Методов синтаксического разбора существует довольно много. Общая классификация рассматриваемых вариантов построения синтаксического распознавателя представлена на рисунке 5.1.

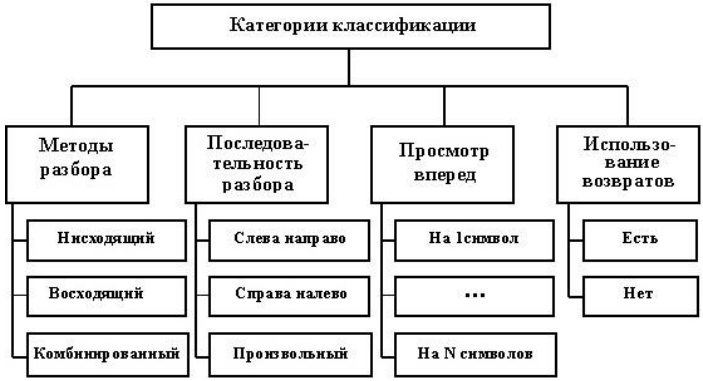


Рис. 5.1 Классификация методов организации синтаксического разбора

Рассмотрим некоторые принципиально важные варианты. Их можно классифицировать на три класса: два основных — нисходящий, восходящий и третий — комбинированный.

Отдельно можно отметить

— ассемблеры, в которых разбор осуществляется построчно, часто без необходимости использования грамматик, и

— препроцессоры, которые просто производят замену строк с применением некоторых простых правил.

Каждый метод синтаксического разбора характеризуется:

- — последовательностью разбора;
- — использованием просмотра вперед;
- — использованием возвратов.

1. *Нисходящий разбор*. Суть состоит в том, что текст некоей программы, который с самого начала представлен в виде очень большой строки, постепенно разбивается на лексемы; к лексемам применяется синтаксический анализ для построения внутреннего представления программы.
2. *Восходящий разбор*. Суть этого типа разбора состоит в том, что программа представляется в виде последовательности лексем, которые далее «снизу-вверх» склеиваются в более сложные предложения языка; после успешной склейки производится построение внутреннего представления программы.
3. *Комбинированный разбор.*

В реальных компиляторах редко используются первые два метода в чистом виде. Обычно используются комбинированные методы, объединяющие те или иные фазы в два – три прохода анализа с учетом *локальной оптимизации*, или измененные методы разбора с целью повышения скорости разбора. При этом надо отметить, что увеличение фрагментов кода локальной оптимизации повышает удельную массу всего кода программы. И так как время синтаксического разбора в современных технологиях разработки компиляторов занимает небольшой процент общего времени компиляции, нет необходимости заниматься оптимизацией разбора на этом этапе сверх меры.

Прежде всего рассмотрим схематическое представление этих видов разбора.

1. *Нисходящий разбор* заключается в построении дерева разбора, начиная от корневой вершины. Разбор заключается в последовательной подстановке соответствующих правил между начальным нетерминалом и символами входной цепочки.

Начинается разбор с правил, выводимых из начального нетерминала. Подстановка основывается на том факторе, что корневая вершина является узлом, состоящим из листьев, являющихся цепочкой терминалов и нетерминалов одного из альтернативных правил, порождаемых начальным нетерминалом. Далее подставляемое правило в общем случае выбирается произвольно. Вместо новых нетерминальных вершин осуществляется подстановка выводимых из них правил.

Процесс протекает до тех пор, пока не будут установлены все связи дерева, соединяющие корневую вершину и символы входной цепочки, или пока не будут перебраны все возможные комбинации правил. В последнем случае входная цепочка отвергается. Построение дерева разбора подтверждает принадлежность входной цепочки данному языку.

При этом, в общем случае, для одной и той же входной цепочки может быть построено несколько деревьев разбора. Это говорит о том, что грамматика данного языка является недетерминированной.

Для примера рассмотрим простую грамматику ***G***: ***G= ({S}, {a, +, *}, P, S),*** где правила ***P*** определяется следующим образом:

1. $S \rightarrow a$
2. $S \rightarrow S + S$
3. $S \rightarrow S * S$

Недетерминированность грамматики позволяет порождать одну и ту же терминальную цепочку с использованием различных выводов. Например, выражение «***a+a*a+a***» можно получить способами:

1. $S \rightarrow S+S \rightarrow a+S \rightarrow a+S*S \rightarrow a+a*S \rightarrow a+a*S+S \rightarrow a+a*a+S \rightarrow a+a*a+a$
2. $S \rightarrow S+S \rightarrow S+a \rightarrow S*S+a \rightarrow S*a+a \rightarrow S+S*a+a \rightarrow S+a*a+a \rightarrow a+a*a+a$ (5.1)
3. $S \rightarrow S*S \rightarrow S+S*S \rightarrow S+S*S+S \rightarrow a+S*S+S \rightarrow a+a*S+S \rightarrow a+a*S+a \rightarrow a+a*a+a$

Этот демонстрационный пример показывает, что число вариантов одной и той же произвольной цепочки вывода может быть весьма велико, поэтому нет смысла говорить о практическом применении такого разбора для данной грамматики. Но этот пример позволяет показать, каким образом могут порождаться различные деревья при нисходящем разборе.

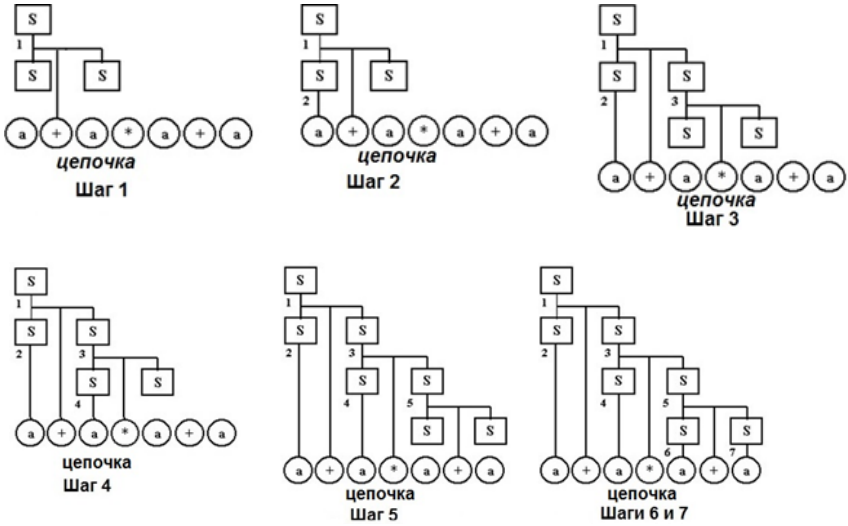


Рис. 5.2. Нисходящий разбор цепочки *слева направо*

Пошаговое построение различных деревьев показано на рисунках 5.2, 5.3, 5.4. Отметим, что процесс построения дерева совпадает с последовательностью шагов вывода входной цепочки.

Рисунок 5.2 представляет схему нисходящего разбора слева направо.

Рисунок 5.3 представляет схему нисходящего разбора справа налево

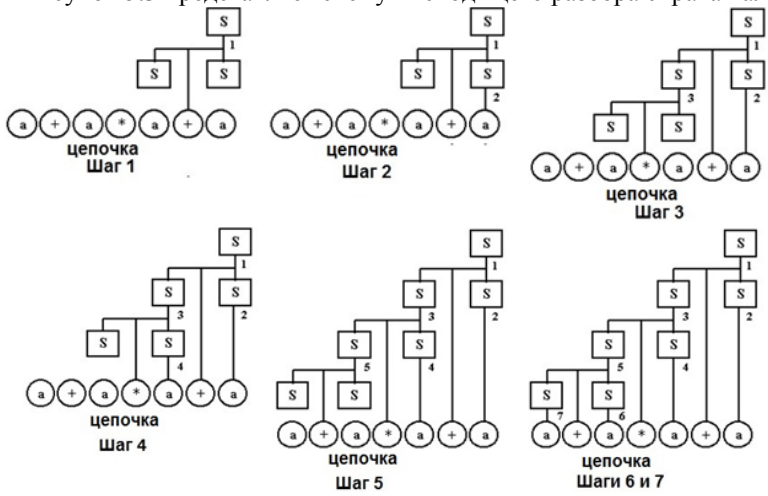


Рис. 5.3. Нисходящий разбор цепочки *справа налево*

И, наконец, приведем схему нисходящего *комбинированного* разбора, в котором исходной точкой разбора взят терминальный символ умножения.

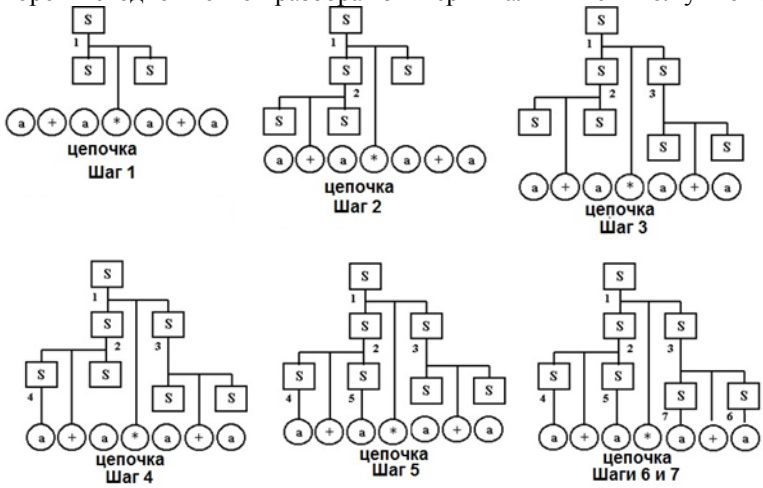


Рис. 5.4. Нисходящий разбор цепочки комбинированным методом, начиная с символа умножения

2. *Восходящий разбор.* Синтаксическое дерево начинает строиться от терминальных листьев подстановкой правил, применяемых к входной цепочке, в общем случае, как и при нисходящем разборе, в произвольном порядке. На очередном шаге новые узлы полученных поддеревьев используются как листья для вновь применяемых правилах. Процесс

построения дерева разбора завершается, когда все символы входной цепочки будут являться листьями дерева, корнем которого окажется начальный нетерминал. Если, в результате полного перебора всех возможных правил, невозможно построить требуемое дерево разбора, то рассматриваемая входная цепочка считается не принадлежащей данному языку.

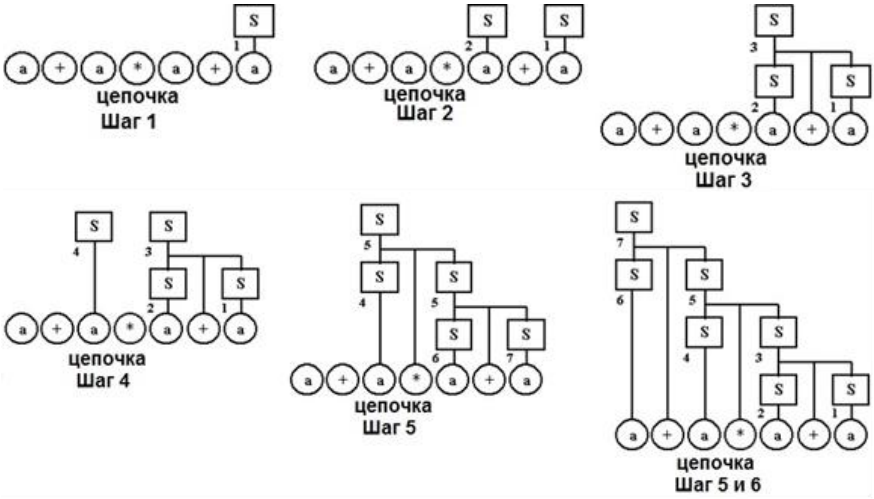


Рис. 5.5. Восходящий разбор справа налево

При восходящем разборе также возможно получение любого возможного вывода цепочки из начального нетерминала, но реализуется этот вывод, в отличие от нисходящего разбора, с точностью до "наоборот".

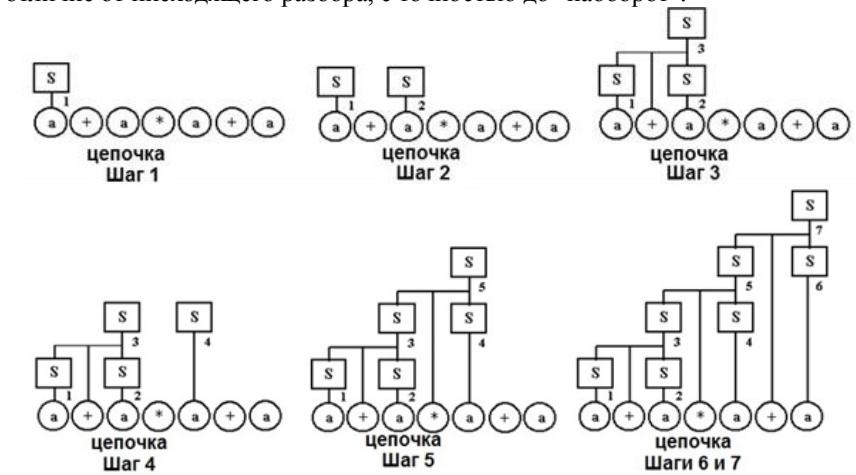


Рис. 5.6. Восходящий разбор слева направо

На рисунке 5.5, 5.6, 5.7 приведены примеры построения деревьев разбора для грамматики ***G*** и процессов порождения цепочек, представленных выражениями (5.1), где шаги порождения дерева соответствуют *движению по представленным цепочкам вывода* (5.1), но теперь справа налево.

Комбинированный разбор реализуют тогда, когда процесс распознавания разбивается на два этапа. На одном этапе осуществляется нисходящий, а на другом — восходящий разбор. Этапов может быть и больше, а порядок их применения — произвольным. Вообще говоря, комбинированным можно считать разбор в любом трансляторе, если фазу лексического анализа принять за первый этап, а синтаксического — за второй.

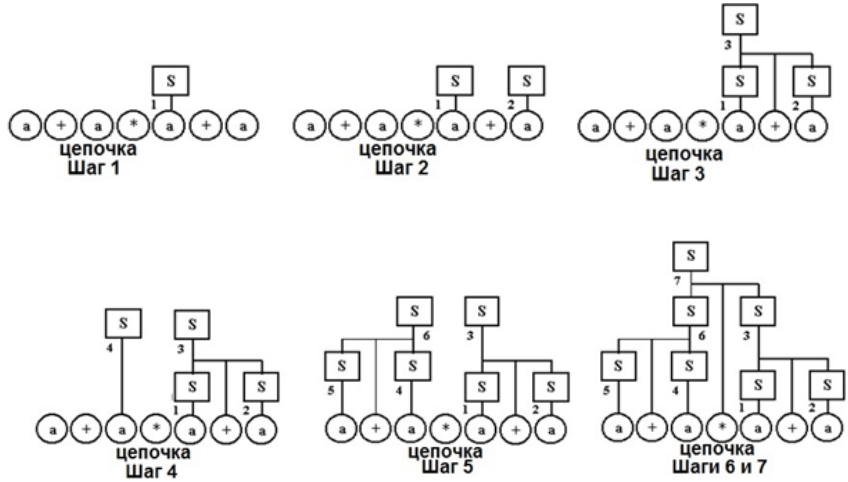


Рис. 5.7. Восходящий комбинированный произвольный разбор

Но лексический анализатор нельзя считать истинным распознавателем, так как осуществляется формирование только одного слоя ветвей в дереве разбора, расположенного над символами входной цепочки (рис. 5.8).

Рассмотрев основные простейшие схемы синтаксических разборов, отметим существенные особенности, которые определяют сложность разбора реальных языков программирования.

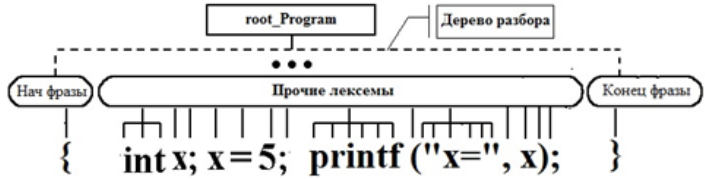


Рис. 5.8. Пример комбинированного разбора, когда в роли распознавателя самых нижних ветвей дерева выступает лексический анализатор

Прямой лексический анализатор, в этом случае, будет являться восходящим распознавателем, который далее может сочетаться с нисходящим. Непрямой же лексический анализ можно рассматривать как нисходящий разбор некоторой *подцепочки* (при проверке версий). Поэтому комбинированный разбор будет осуществляться при его совместном использовании с восходящим распознавателем. В синтаксическом же анализаторе комбинация различных видов разборов обычно не используются, так как в этом нет особого смысла.

Повышение эффективности разбора осуществляется разработкой грамматик, специально поддерживающих согласованные между собой *метод* и *последовательность* разбора.

Чем сложнее язык, тем сложнее грамматика, в которой могут встречаться альтернативные правила, начинающиеся с одинаковых цепочек символов. Возникающая *неоднозначность* может быть разрешена путем предварительного просмотра правила на **n** символов вперед до той границы, начиная с которой данное правило можно будет отличить от других. В грамматиках могут встречаться альтернативные правила, начинающиеся с одинаковых цепочек символов. Возникающая неоднозначность может быть разрешена путем *предварительного просмотра правила* на ***n*** символов вперед до той границы, начиная с которой данное правило можно будет отличить от других. Просмотр вперед - это один из возможных вариантов упорядочивания подстановок, обеспечивающий решение проблемы недетерминированности. С этой же целью используются возвраты для альтернативных правил, начинающихся с одинаковых подцепочек.

Несмотря на разнообразие факторов, влияющих на особенности организации разрабатываемого языка программирования, при разработке анализатора всегда необходимо искать разумный компромисс, то есть всегда надо учитывать, что:

— чем сложнее синтаксис языка программирования, тем сложнее его грамматика;
— чем сложнее грамматика, тем больше сложность и универсальность методов разбора;
— универсальные методы разбора ведут к более медленному его выполнению.

Поэтому, на всем протяжении процесса разработки компилятора языка программирования, вплоть до его реализации, постоянно необходимо обеспечивать баланс между сложностью замысла и простотой реализации. Кроме того, усложнение синтаксиса не всегда обеспечивает более удобное восприятие конструкций языка пользователем. Можно иметь простой синтаксис и удобный для использования язык, поддерживаемый очень быстрым транслятором.

Учитывая все выше сказанное, займемся синтаксическими методами разбора, используемыми в современных разработках.

В настоящее время в нисходящем разборе используется либо *LL(1)* анализ (рекурсивный спуск) и его вариант — *LL(k)* анализ, а в восходящем разборе *LR(1)* анализ и варианты *LR(0), SLR(1), LALR(1)* и другие (глава 3).

Рекурсивный спуск чаще используется при ручном программировании синтаксического анализатора, *LR(1)* — при использовании систем автоматизации построения синтаксических анализаторов.

Результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицу символов (глава 3, рис. 3.6).

Алгоритмы разбора, которые будут описаны ниже (и нисходящий, и восходящий) являются алгоритмами *слева направо* ввиду того, что они обрабатывают сначала самые левые символы обрабатываемой цепочки и продвигаются по цепочке только тогда, когда это необходимо. Можно, как мы видели, подобным способом определить разбор справа налево, но он *менее естественен* (инструкции в программе выполняются слева направо, да и мы читаем слева направо).

Рассмотрим теперь более подробно методы синтаксического анализа, используемые при ручном и автоматическом проектировании синтаксического анализатора компилятора.

I. Метод нисходящего разбора.

Напомним, что при нисходящем разборе дерево строится от корня (начального символа) вниз к концевым узлам. *LL(1)*-грамматика

— это, во-первых, *КС*-грамматика, то есть левые части правил начинаются с единственного нетерминала (грамматика типа 2 по Хомскому),

— *КС*-грамматика с такими правилами, в которых одинаковые левые части не пересекаются (то есть правила с одинаковыми правыми частями в левых частях правил имеют разные нетерминалы),

— *КС(1)*-грамматика, где цифра 1 говорит, что для определения пути разбора нужен просмотр вперед всего одной лексемы. В качестве примера можно предложить *LL(1)*-грамматику, содержащую следующие правила:

1.	***SAbB***	***4.***	***A→B***	***7.***	***C→a***
2.	***S→d***	***5.***	***B→cSd***	***8.***	***C→ed***
3.	***A→Cab***	***6.***	***B→ε***		

То, что это именно *LL(1)*-грамматика доказывается в теоретических источниках (книге Льюиса[16] и других), где приводится описание доказательства на одиннадцати страницах, с использованием демонстрационных примеров, разреженных матриц) и получением дополнительной информации для построения автомата с магазинной памятью.

[16] Ф.Льюис, Д.Розенкранц, Р.Стирнз. Теоретические основы проектирования компиляторов. М., Мир, 1979г.

Частично в ряде случаев для нисходящих анализаторов используются и автоматические среды для построения лексических и синтаксических анализаторов. В этом случае используется метод *LL(k)* — левый вывод грамматики. Здесь ***k*** означает, что нисходящий разбор зависит не от одного рассмотренного символа, а от слова длиной ***k*** символов.

Из теории формальных грамматик известно, что КС-грамматики можно описать *недетерминированным* конечным автоматом (НКА) с магазинной памятью. Но не все КС-грамматики пригодны для построения нисходящего детерминированного МП-автомата, так как многие из них могут порождать одну и ту же терминальную цепочку различными левыми выводами (получим неоднозначность вывода цепочки). А так как есть неоднозначность вывода, следовательно, невозможно использование для *детерминированного* разбора.

Однако, определены и изучены такие классы грамматик, которые поддерживают нисходящий детерминированный разбор. Наиболее простыми из них являются ***S***-грамматики.

КС-грамматика называется ***S***-грамматикой (раздельной, или простой) тогда и только тогда, когда выполняются два условия:

1. правая часть каждого правила начинается терминалом t_i.
2. если два правила имеют совпадающие левые части, то правые части этих правил должны начинаться различными терминальными символами.

На основе выше сказанного, учитывая эквивалентность построенной таблицы переходов и исходной грамматики, а также особенности организации современных языков программирования, можно реализовать логику работы нисходящего распознавателя *LL(1)*-грамматики на основе таблицы переходов нисходящего автомата с магазинной памятью, построенного по правилам этой грамматики. Метод разработки программы на основе синтаксических правил *LL(1)*-грамматики называется *рекурсивным спуском*.

В общем случае *метод рекурсивного спуска* или *нисходящий разбор* — это один из методов определения принадлежности входной строки к некоторому формальному языку, описанному *LL(k)* контекстно-свободной грамматикой. Это класс алгоритмов грамматического анализа, где правила формальной грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности символов (просматривая вперед символов).

Рассмотрим формальные условия применения алгоритма на примере.

Пусть в рассматриваемой формальной грамматике $G = (N, \Sigma, S, P)$:

N — это конечное множество нетерминальных символов;

Σ — конечное множество терминальных символов,

S — стартовый символ,

P — правила грамматики.

Тогда метод рекурсивного спуска применим только, если каждое правило этой грамматики имеет следующий вид:

— или $A \to \alpha$, где α, $\alpha \subset (\Sigma \cup N)*$ — и это единственное правило вывода для этого нетерминала

— или $A \to a_1\alpha_1 | a_2\alpha_2 | \ldots | a_n\alpha_n |$ для всех $i=1, 2, \ldots, n$; $a_i \neq a_j$, $i \neq j$;

Идея метода (общие положения).

Для каждого нетерминального символа K_i строится *функция*, которая для любого входного слова x удовлетворяет следующим критериям:

— считывает из еще необработанного входного потока максимальное начало A, являющегося началом некоторого слова, выводимого из K_i;

— определяет, является ли A выводимым из K_i или просто невыводимым началом выводимого из K_i слова.

В случае, если такое начало считать не удается (и корректность функции для нетерминала K доказана), то входные данные не соответствуют языку, и следует остановить разбор.

Разбор заключается в вызове описанных выше функций. Если для считанного нетерминала есть составное правило, то при его разборе будут вызваны другие функции для разбора входящих в него терминалов. В результате получим дерево вызовов, начиная с самой «верхней» функции, эквивалентное дереву разбора.

Наиболее простой и «человечный» вариант создания анализатора, использующего метод рекурсивного спуска, — это непосредственное (ручное) программирование по каждому правилу вывода для нетерминалов грамматики.

Известны следующие наиболее популярные алгоритмы нисходящего разбора (для самостоятельного изучения):

— рекурсивный нисходящий парсер (*Recursive descent parser*)

— *LL*-парсер (*LL parser*)

— парсер старьёвщика (*Packrat*)

II. Восходящий разбор (сдвиг свертки).

При разборе снизу вверх промежуточные выводы перемещаются по дереву по направлению к корню, поэтому такой разбор называется *свёрткой*. В восходящем разборе используются *LR*-грамматики (**L** — означает просмотр цепочки слева направо, а **R** — относится к конечным выводам.

Так как свертку применяем каждый раз только к последним символам цепочки, получим правые выводы цепочки.

Часто в восходящем разборе используют *грамматики предшествования*.

Разбор конструкций при восходящем методе — для получения синтаксического дерева — осуществляется методом *операторного предшествования*, основанном на задании матрицы предшествования, которая содержит символы начала «<», конца «>» и середины «=» фразы

(первичной основы). Эти символы («<», «>» и «=») — это не символы меньше, больше или равно. Это именно символы, определяющие начало, конец или середину некоторой последовательности символов.

Важно отметить, что значение отношения в методе *операторного предшествования* зависит от предыдущего и текущего (одного) рассмотренного символа, поэтому и единица в скобках в названии метода *LR(1).*

На вход синтаксического анализатора поступает дерево с одной ветвью. Положим, что представление этого дерева, — это последовательность лексем, представляющая фразы со вспомогательными символами «<», «>» и «=».

Алгоритм синтаксического разбора в этом случае состоит в следующем.

1. Сначала отыскивается самая левая первичная фраза (то есть отыскивается самый левый символ «<», отмечающий начало фразы), а затем действует по следующему рекурсивному алгоритму.
2. Символы входной строки переписываются в стек, пока между верхним нетерминальным символом стека и очередным нетерминальным символом строки не будет отношения «>» (значок конца основы).
3. Затем уже стек просматривается от вершины стека к его началу до тех пор, пока между двумя очередными нетерминальными символами стека не будет отношения «<» (значок начала основы).
4. Вызывается программа по обработке найденной основы (первичной).
5. Найденная основа заменяется в стеке на соответствующий нетерминальный символ в соответствии с правилом грамматики исходного языка.
6. После замены первичной основы на нетерминальный символ просмотр программы начинается с начала с целью выявить новую основу. (снова найти символ начала фразы («<») — первичной основы).

В результате выстраивается синтаксическое дерево вывода — дерево с иерархиями.

Замечание. Отметим связь рассматриваемого метода с изученным материалом теории формальных грамматик, так как этот метод строится на выводах этой теории. Грамматика является грамматикой *предшествования*, если она *приведенная, обратимая*, и между двумя терминальными или нетерминальными символами выполняется не более одного отношения предшествования.

Напомним, что грамматика называется обратимой, если в ней нет двух правил с одинаковыми правыми частями; грамматика называется приведенной, если в ней нет **ε**-правил, бесполезных символов и циклов.

Грамматики предшествования. Для грамматик предшествования можно построить восходящий распознаватель на основе алгоритма *перенос/свертка.* Данный метод будет сводиться к основе, которая соответствует правой части грамматики.

При этом между символами устанавливается некоторое отношение называемое *отношение предшествования*.

Рассмотрим два вида грамматик: *простое предшествование* и *операторное предшествование*.

Грамматикой *простого предшествования* называется грамматика без **ε**-правил для которой выполняются следующие условия:

— для каждой упорядоченной пары терминальных и нетерминальных может быть установлено одно из трех отношений предшествования.
  - **<.** — предшествует;
  - **.>** — следует;
  - **=.** — составляет основу;

— различные правила грамматики не могут содержать одинаковую часть.

Для построения отношения предшествования используется специальный алгоритм, который требует нахождения для каждого правила грамматики множеств крайних левых и крайних правых символов:

L(I) – для крайних левых,

R(I) – для крайних правых

Шаг 1. Для каждого нетерминального символа ***A*** ищем все правила, содержащие ***A*** в левой части, получаем множество ***L(A)***.

Если множество ***L(A)*** содержит нетерминальные символы: ***A1, A2, … , An***, то множество ***L(A)*** необходимо дополнить символами, входящими в множество: ***L(A1), L(A2), … , L(An)***.

Шаг 2 повторяется пока множество не перестает выполняться.

A<.B – перенос,

A.>B – свертка,

A=.B – перенос.

Для любых подряд идущих символов правой части грамматики справедливо следующее:

— на пересечения строки **A** и столбца **B** ставим знак «составление основы» (**=.**),

— на пересечении строки **A** и столбца **B(B)** ставим знак «предшествует» (**<.**),

— на пересечении строк **R(A)** и столбцов **B, L(B)** ставим знак «следует» (**.>**),

— на пересечении строк *начала* и столбцов **L(I)** ставим знак «предшествует» (**<.**),

— на пересечении строк **R(I)** и столбцов, соответствующих «конец строки» ставим знак «следует» (**.>**).

Пример 5.1.

I→TR | T

R→+T | -T | +TR | -TR

T→a | b

Строим крайние левые и правые:

L(I) = { T}	***R(I) = { R, T}***
L(R) = { +, -}	***R(R) = {T, R}***
L(T) = { a, b}	***R(T) = {a, b}***

Дополняем к ***T*** входящие ***a, b:***

L(I) = { T, a, b}	***R(I) = { R, T, a, b}***
L(R) = { +, -}	***R(R) = {T, R, a, b}***
L(T) = { a, b}	***R(T) = {a, b}***

Вопросы для контроля

1. Зачем, при синтаксическом разборе нужны автоматы с магазинной памятью?
2. Методы программной реализации нисходящих автоматов с магазинной памятью.
3. В чем специфика распознавателя, построенного с применением рекурсивного спуска?
4. Какие данные являются входными для этапа семантического анализа?
5. Какие действия выполняет семантический анализатор?
6. При невыполнении каких семантических требований семантический анализатор должен выдать сообщение об ошибке?

Глава 6. Преобразование дерева разбора в дерево операций

Учебные цели:

— дать представление об особенностях преобразования дерева разбора в дерево операций;
— дать представление о реализации деревьев с помощью списков и массивов.

Результатом работы распознавателя *КС*-грамматики входного языка является последовательность правил грамматики, применённых для построения входной цепочки. По найденной последовательности, зная тип распознавателя, можно построить цепочку вывода или дерево вывода. В этом случае дерево вывода выступает в качестве дерева синтаксического разбора и представляет собой результат работы синтаксического анализатора в компиляторе.

Однако, ни цепочка вывода, ни дерево синтаксического разбора не являются целью работы компилятора. Дерево вывода содержит массу избыточной информации, которая для дальнейшей работы компилятора не требуется. Эта информация включает в себя все нетерминальные символы, содержащиеся в узлах дерева, — после того как дерево построено, они не несут никакой смысловой нагрузки и не представляют интереса для дальнейшей работы фаз компилятора.

Для полного представления о типе и структуре найденной и разобранной синтаксической конструкции входного языка, в принципе, достаточно знать последовательность номеров правил грамматики, применённых для построения этой синтаксической конструкции. Однако, форма представления этой добавочной информации может быть различной, как в зависимости от реализации самого компилятора, так и от фазы компиляции. Эта форма называется *внутренним представлением* программы.

В синтаксическом дереве внутренние узлы (вершины) соответствуют *операциям*, а листья представляют собой *операнды*. Как правило, листья синтаксического дерена связаны с записями в таблицах, в частности с записями в таблице идентификаторов. Структура синтаксического дерева отражает синтаксис языка программирования, на котором написана исходная программа.

Синтаксические деревья могут быть построены компилятором для любой части входной программы. Но не всегда синтаксическому дереву должен соответствовать фрагмент кода результирующей программы: например, возможно построение синтаксических деревьев для декларативной части языка. В этом случае операции, имеющиеся в дереве вывода, не требуют порождения объектного кода, но несут информацию о действиях, которые должен выполнить сам компилятор над соответствующими элементами.

В том же случае, когда синтаксическому дереву соответствует некоторая последовательность операций, влекущая порождение фрагмента объектного кода, говорят о *дереве операций*.

Дерево операций можно непосредственно построить из дерева вывода, порождённого синтаксическим анализатором. Для этого достаточно исключить из дерева вывода цепочки нетерминальных символов, а также узлы, не несущие семантической нагрузки при генерации кода. Примером таких узлов могут служить различные скобки, которые определяют, или меняют, порядок выполнения операций и операторов, но после построения дерева никакой смысловой нагрузки уже не несут, так как им не соответствует никакой объектным код.

То, какой узел в дереве является операцией, а какой — операндом, никоим образом невозможно определить из грамматики, описывающей синтаксис входного языка. Аналогично, ниоткуда не следует то, каким операциям должен соответствовать объектный код в результирующей программе, а каким — нет. Все это определяется только исходя из семантики — «смысла» — языка входной программы. Поэтому только разработчик компилятора может четко определить, как при построении дерева операций должны различаться операнды и сами операции, а также то, какие операции являются семантически незначащими для порождения объектного кода.

6.1. Моделирование дерева разбора с помощью списков

Известно, что структура деревьев хорошо описывается с помощью списков. Структура списков в условиях динамического распределения памяти позволяет легко удалять, добавлять, заменять элементы списка, то есть легко преобразовывать связи между элементами. Одним из вариантов организации списочной структуры является дерево (организацию списочной структуры подробнее рассмотрим в главе 11). В частности, сначала результат лексического анализа (дерево с одной ветвью, рис. 6.1) преобразуется в более сложное дерево (с иерархиями) — синтаксическое дерево (рис. 6.2), которое удобно затем преобразовывать в дерево операций. Описание одного из алгоритмов преобразования синтаксического дерева в дерево операций представлено ниже.

С помощью дерева операций в дальнейшем эффективно формируются специальные формы команд на этапе генерации.

Следуя ниже приведенному алгоритму, рекомендуется на практическом занятии разработать программу моделирования дерева разбора исходной цепочки, полученной после синтаксического анализа.

Алгоритм преобразования дерева синтаксического разбора в дерево операции можно представить следующим образом.

Шаг 1. Если в дереве больше не содержится узлов, помеченных нетерминальными символами, то выполнение алгоритма завершено; иначе — перейти к шагу 2.

Шаг 2. Выбрать крайний левый узел дерева, помеченный нетерминальным символом грамматики и сделать его текущим. Перейти к шагу 3.

Шаг 3. Если текущий узел имеет только один нижележащий узел, то текущий узел необходимо удалить из дерева, а связанный с ним узел присоединить к узлу вышележащего уровня (исключить из дерена цепочку) и вернуться к шагу 1; иначе — перейти к шагу 4.

Шаг 4. Если текущий узел имеет нижележащий узел (лист дерева), помеченный терминальным символом, который не несет семантической нагрузки, тогда этот лист нужно удалить из дерева и вернуться к шагу 3; иначе — перейти к шагу 5.

Шаг 5. Если текущий узел имеет один нижележащий узел (лист дерева), помеченный терминальным символом, обозначающим знак операции, а остальные узлы помечены как операнды, то лист, помеченный знаком операции, надо удалить из дерева, текущий узел пометить этим знаком операции и перейти к шагу 1; иначе — перейти к шагу 6.

Шаг 6. Если среди нижележащих узлов для текущего узла есть узлы, помеченные нетерминальными символами грамматики, то необходимо выбрать крайний левый среди этих узлов, сделать его текущим узлом и перейти к шагу 3; иначе — выполнение алгоритма завершено.

Этот алгоритм всегда работает с узлом дерева, который считается текущим и стремится исключить из дерева, все узлы, помеченные нетерминальными символами. То, какие из символов считать семантически незначащими, а какие считать, знаками операций, решает разработчик компилятора. Если семантика языка задана корректно, то в результате работы алгоритма из дерева будут исключены все нетерминальные символы.

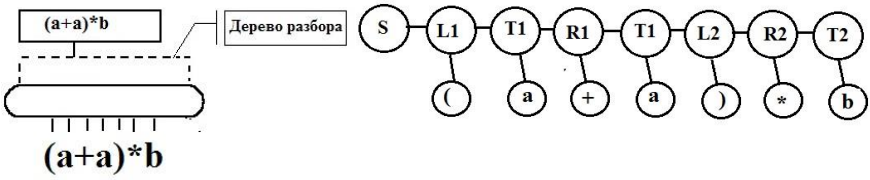


Рис. 6.1. Построение дерева вывода после лексического анализа

Рассмотрим разбор выражения ***(a+a)*b.***

Сначала проведем лексический разбор и получим дерево вывода с одной ветвью (рис. 6.1).

Затем построим дерево вывода с иерархиями (после синтаксического анализа, см. рисунок 6.2), с добавленными нетерминалами.

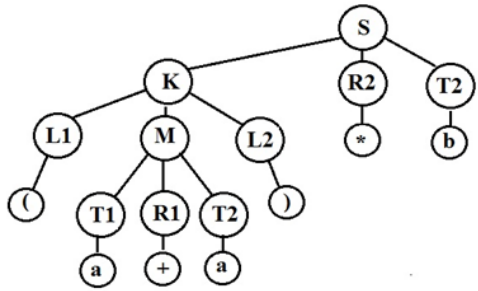


Рис. 6.2. Синтаксическое дерево вывода (с иерархиями)

Замечание. Обратите внимание, что операции плюс «+» и умножение «*» в формуле ***(a + a) * b*** — операции бинарные, поэтому имеют по 2 операнда; их правильный синтаксис и, соответственно, порядок расположения на диаграмме Вирта слева направо: «операнд» «операция» «операнд».

Теперь покажем, как строится дерево операций согласно приведенному выше алгоритму. Текущий узел будем обозначать двойной границей.

В соответствии с шагом 1 алгоритма начинается построение с корня дерева «***S***»: берём его в качестве текущего узла (рис. 6.3). Переходим к шагу 2: выбираем крайний левый узел дерева, помеченный нетерминальным символом грамматики «***K***» и делаем его текущим (рис. 6.4.). Узел «***K***» имеет несколько нижележащих узлов, поэтому переходим к шагу 4; этот узел в качестве нижележащих узлов имеет только нетерминальные узлы, поэтому переходим к шагу 5. На шаге 5 нет нижележащих узлов, помеченных терминальным символом, поэтому переходим к шагу 6. Среди нижележащих узлов для текущего узла «***K***» выбираем крайний левый «***L1***», делаем его текущим (рис. 6.5) и переходим к шагу 3.

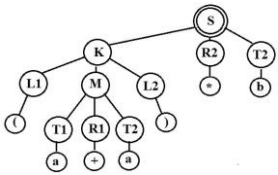


Рис. 6.3. Шаг 1

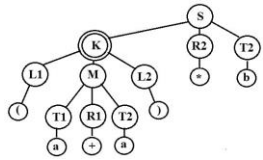


Рис. 6.4. Шаг 2

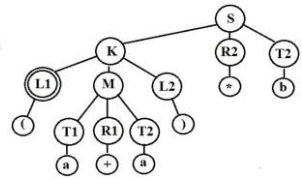


Рис. 6.5. Шаг 6

В соответствии с шагом 3 текущий узел «***L1***» имеет только один нижележащий узел: удаляем его из дерева, а связанный с ним узел «(» присоединяем к узлу вышележащего уровня «***K***» (рис. 6.6), при этом текущим становится вышележащий узел «***K***»; возвращаемся к шагу 1, а затем к шагу 2.

В соответствии с шагом 2 в поддереве, корень которого «***K***», ищем крайний левый узел дерева, помеченный нетерминальным символом грамматики и делаем его текущим (рис. 6.7), затем переходим к шагу 3.

У текущего узла «***M***» несколько нижележащих узлов, помеченных нетерминальными символами, поэтому переходим к шагу 4. Условиям шага 4 текущий узел «***M***» не удовлетворяет, так как имеет только узлы, помеченные нетерминальными символами ***«T1», «R1», «T2»***, поэтому переходим к шагу 5. Условиям шага 5 текущий узел «***M***» также не удовлетворяет, так как не имеет узлов, помеченных терминальными символами, переходим к шагу 6.

Среди нижележащих узлов для текущего узла «***M***» есть узлы, помеченные нетерминальными символами грамматики ***«T1», «R1», «T2»***, выбираем крайний левый среди этих узлов ***«T1»*** и делаем его текущим узлом, а затем переходим к шагу 3.

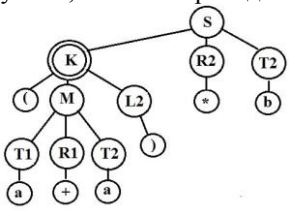


Рис. 6.6. Шаг 3

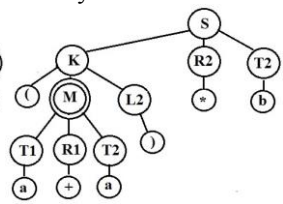


Рис. 6.7. Шаг 2

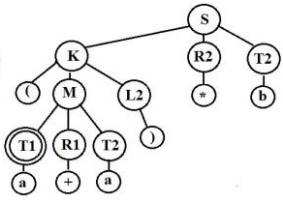


Рис. 6.8. Шаг 6

На шаге 3 текущий узел ***«T1»*** имеет только один нижележащий узел ***«a»***, поэтому текущий узел ***«T1»*** удаляем из дерева, а связанный с ним узел ***«a»*** присоединяем к узлу вышележащего уровня «***M***» (рис. 6.9), который становится текущим, затем возвращаемся к шагу 1.

Шаг 1 не выполняется, так как построение дерева операций еще не завершено, переходим к шагу 2. Выбираем у поддерева с корнем «***M***» крайний левый узел дерева, помеченный нетерминальным символом грамматики ***«R1»*** и делаем его текущим (рис. 6.10), затем переходим к шагу 3.

На шаге 3 текущий узел ***«R1»*** имеет только один нижележащий узел ***«+»***, то текущий узел ***«R1»*** удаляем из дерева, а связанный с ним узел ***«+»*** присоединяем к узлу вышележащего уровня ***«M»*** (рис. 6.11), который становится текущим, затем возвращаемся к шагу 1.

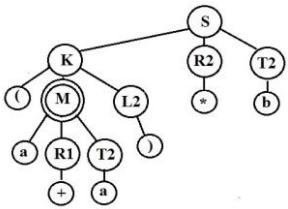


Рис. 6.9. Шаг 6

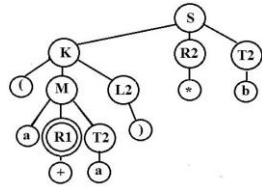


Рис. 6.10. Шаг 2

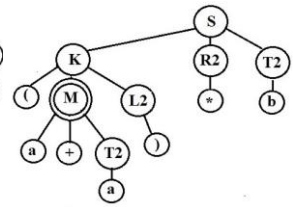


Рис. 6.11. Шаг 3

Шаг 1 не выполняется, так как построение дерева операций еще не завершено, переходим к шагу 2. Выбираем у поддерева с корнем «M» крайний левый узел дерева, помеченный нетерминальным символом грамматики ***«T2»*** и делаем его текущим (рис. 6.12), затем переходим к шагу 3.

На шаге 3 текущий узел ***«T2»*** имеет только один нижележащий узел ***«a»***, поэтому текущий узел ***«T2»*** удаляем из дерева, а связанный с ним узел ***«a»*** присоединяем к узлу вышележащего уровня ***«M»*** (рис. 6.13), который становится текущим, затем возвращаемся к шагу 1.

Шаг 1 не выполняется, так как построение дерева операций еще не завершено, переходим к шагу 2. На шаге 2 среди узлов ***«M»*** и ***«L2»*** крайним левым узлом дерева, помеченным нетерминальным символом грамматики, является узел ***«M»***, он остается текущим и переходим к шагу 3.

Текущий узел ***«M»*** имеет только несколько нижележащих узлов, поэтому условие шага 3 не выполняется, переходим к шагу 5.

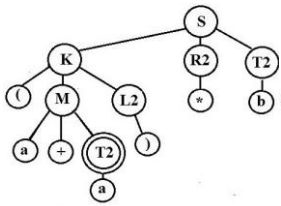


Рис. 6.12. Шаг 2

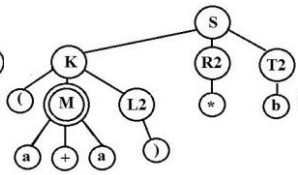


Рис. 6.13. Шаг 3

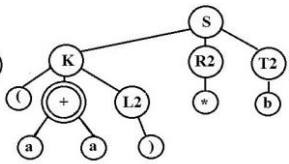


Рис. 6.14. Шаг 5

Ввиду того, что выполняется условие пункта 5, а именно, текущий узел ***«M»*** среди нижележащих узлов имеет один нижележащий узел, помеченный терминальным символом, обозначающим знак операции ***«+»,*** а остальные узлы помечены как операнды ***«a»*** и ***«a»***, поэтому лист, помеченный знаком операции ***«+»***, надо удалить из дерева, а текущий узел пометить этим знаком операции ***«+»*** (терминальным символом)(рис. 6.14); в этой ветви нет узлов, помеченных нетерминальными символами, делаем текущим вышележащий узел ***«K»*** (рис. 6.15) и переходим к шагу 1.

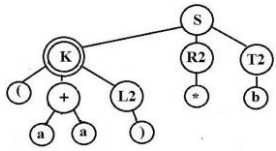


Рис. 6.15. Шаг 5

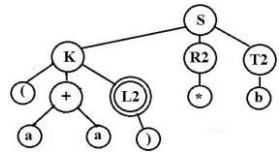


Рис. 6.16. Шаг 2

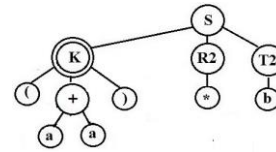


Рис. 6.17. Шаг 3

Шаг 1 не выполняется, так как построение дерева операций еще не завершено, переходим к шагу 2.

На шаге 2 крайним нижележащим левым узлом дерева, помеченным нетерминальным символом грамматики, является узел ***«L2»***, он становится текущим (рис. 6.16) и переходим к шагу 3.

Текущий узел ***«L2»*** на шаге 3 имеет только один нижележащий узел ***«)»***, то текущий узел необходимо удалить из дерева, а связанный с ним узел ***«)»*** присоединить к узлу вышележащего уровня ***«K»*** и вернуться к шагу 1 (рис. 6.17).

Шаг 1 не выполняется, так как построение дерева операций еще не завершено, переходим к шагу 2. Текущий узел ***«K»*** является крайним левым узлом дерева, помеченный нетерминальным символом грамматики, поэтому переходим к шагу 3. Условие шага 3 не выполняется, потому что текущий узел ***«K»*** имеет три нижележащих узла, поэтому переходим к шагу 4.

На шаге 4 текущий узел ***«K»*** имеет слева нижележащий узел, помеченный терминальным символом ***«(»***, который не несет семантической нагрузки; удаляем этот лист ***«(»*** (рис. 6.18.) и возвращаемся к шагу 3. Условие шага 3 не выполняется, потому что текущий узел ***«K»*** имеет два нижележащих узла, поэтому переходим к шагу 4.

На шаге 4 текущий узел ***«K»*** имеет справа нижележащий узел, помеченный терминальным символом ***«)»***, который не несет семантической нагрузки; удаляем этот лист ***«)»*** (рис. 6.19) и возвращаемся к шагу 3.

На шаге 3 текущий узел ***«K»*** теперь имеет только один нижележащий узел, удаляем его из дерева, а связанный с ним узел ***«+»*** присоединяем к узлу вышележащего уровня ***«S»*** (рис. 6.20); возвращаемся к шагу 1.

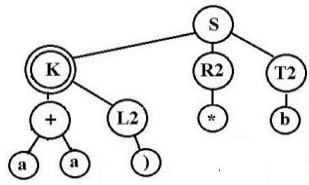


Рис. 6.18. Шаг 4

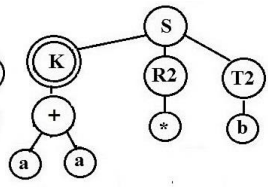


Рис. 6.19. Шаг 4

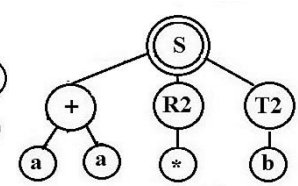


Рис. 6.20. Шаг 3

Шаг 1 не выполняется, так как построение дерева операций еще не завершено, переходим к шагу 2. Выбираем крайний левый узел дерева, помеченный нетерминальным символом грамматики ***«R2»*** (рис. 6.21), и сделаем его текущим, затем переходим к шагу 3.

На шаге 3 текущий узел ***«R2»*** имеет только один нижележащий узел ***«+»***, то текущий узел ***«R2»*** удалим из дерева, а связанный с ним узел ***«+»*** присоединим к узлу вышележащего уровня ***«S»*** (рис. 6.22) и возвратимся к шагу 1.

Шаг 1 не выполняется, так как построение дерева операций еще не завершено, переходим к шагу 2. Выбираем крайний левый узел дерева, помеченный нетерминальным символом грамматики ***«T2»***, и сделаем его текущим (рис. 6.23), затем переходим к шагу 3.

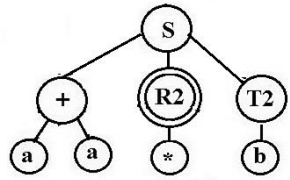


Рис. 6.21. Шаг 3

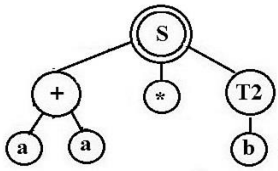


Рис. 6.22. Шаг 2

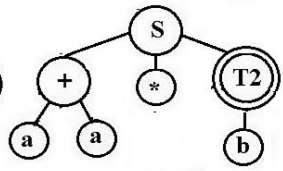


Рис. 6.23. Шаг 3

Соответственно шагу 3 текущий узел ***«T2»*** имеет только один нижележащий узел ***«b»***, поэтому текущий узел ***«T2»*** удаляем из дерева, а связанный с ним узел ***«b»*** присоединяем к узлу вышележащего уровня ***«S»*** (рис. 6.24) и возвращаемся к шагу 1.

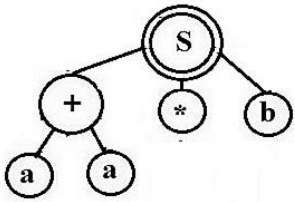


Рис. 6.24. Шаг 3

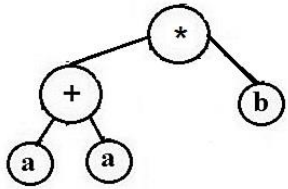


Рис. 6.25. Шаг 3

Шаг 1 не выполняется, так как построение дерева операций еще не завершено, переходим к шагу 2. Крайним левым узлом дерева, помеченным нетерминальным символом грамматики является текущий взел ***«S»*** (остальные разобраны), далее переходим к шагу 3. Условие шага 3 не выполняется, потому что текущий узел ***«S»*** имеет три нижележащих узла, поэтому переходим к шагу 4. Условие шага 4 также не выполняется, потому что все нижележащие узлы имеют семантический смысл, переходим к шагу 5.

Текущий узел ***«S»*** имеет один нижележащий узел, помеченный терминальным символом ***«*»***, обозначающим знак операции ***«*»***, а остальные узлы помечены как операнды: это узлы ***«+»*** и ***«b»*** (вспомните замечание перед объяснением работы разбираемого алгоритма), поэтому лист, помеченный знаком операции ***«*»***, удаляем из дерева, а текущий узел ***«S»*** пометим знаком операции ***«*»*** (рис. 6.25), затем переходим к шагу 1.

Условие шага 1 выполняется, так как в дереве больше не содержится узлов, помеченных нетерминальными символами, следовательно, выполнение алгоритма завершено, и дерево операций построено.

6.2. Работа со статическими массивами

В пункте 6.1 выше рассматривалось представление синтаксического дерева с помощью списочной структуры. Такое представление встречается чаще всего. Но иногда в простых компиляторах или других приложениях, близких по структуре, используется представление дерева в виде статического массива.

Синтаксические деревья общего вида, как было показано, достаточно важны, и в процессе компиляции могут быть весьма сложными. Но, не умаляя общности, сосредоточимся здесь на более простом типе деревьев — на бинарных деревьях.

Такие бинарные деревья (*binary trees*) имеют унифицированную структуру, допускающую разнообразные алгоритмы преобразования и эффективный доступ к элементам. Изучение бинарных деревьев дает возможность решать наиболее общие задачи, связанные с деревьями, поскольку любое дерево общего вида можно представить эквивалентным ему бинарным деревом. Более подробно о классификации бинарных деревьев и их свойствах будет рассмотрено в главе 9.

Бинарные деревья достаточно просто могут быть представлены в виде списков или массивов.

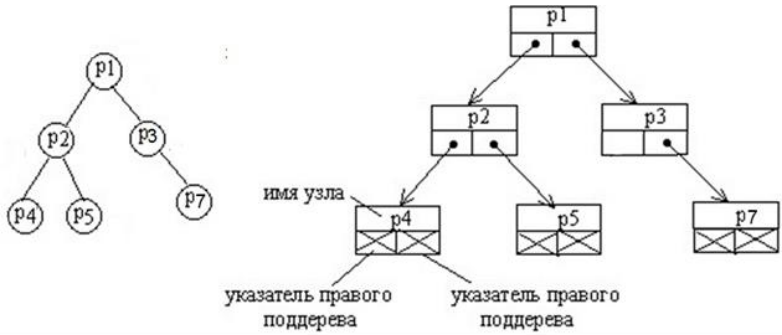


Рис.6.26. Представление бинарного дерева в виде списковой структуры

Списочная структура бинарных деревьев строится на элементах, представляющих узлы дерева (рис. 6.26). Каждый узел (элемент) имеет информационное поле (поле данных) и два поля указателей для связи с двумя потомками (левым и правым). Листья дерева имеют пустые указатели потомков. Указатель на узел, являющийся корнем дерева, хранится без изменений, пока существует дерево.

Такой способ представления бинарного дерева является разновидностью мультисписка, образованного комбинацией множества линейных списков. Каждый линейный список объединяет узлы, входящие в путь от корня дерева к одному из листьев.

Освежив в памяти детали организации структуры дерева в виде списка, обратимся теперь к представлению дерева в виде статического массива.

Прежде всего, отметим, что статический массив имеет изначально заданное и неизменяемое количество элементов

В виде массива, проще всего, представляется полное бинарное дерево, так как оно всегда имеет строго определенное число вершин на каждом уровне. В таком случае вершины можно пронумеровать слева направо последовательно по уровням и использовать эти номера в качестве индексов в одномерном массиве.

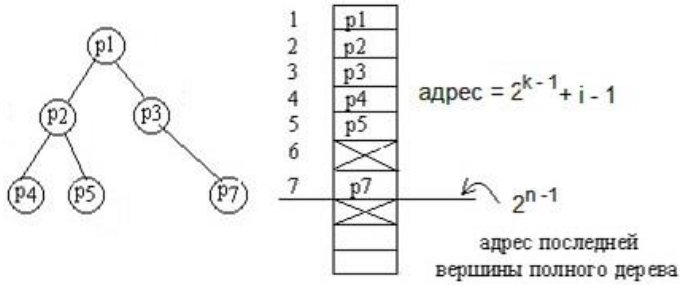


Рис. 6.27. Представление бинарного дерева в виде массива

Если число уровней дерева в процессе преобразований существенно не изменяется, то такой способ представления полного бинарного дерева оказывается значительно более экономичным, чем любая списковая структура (ввиду того, что к элементам массива доступ значительно быстрее, чем к элементам списка).

Однако, далеко не все бинарные деревья являются полными. Для неполных бинарных деревьев применяют следующий способ представления:

— бинарное дерево дополняется до полного дерева,
— вершины последовательно нумеруются.

В массив заносятся только те вершины, которые были в исходном неполном дереве.

При таком представлении элемент массива выделяется независимо от того, будет он содержать узел исходного дерева или нет. Поэтому, необходимо каким-либо способом отметить неиспользуемые элементы массива. Это можно сделать, например, занесением специального значения в соответствующие элементы массива. В результате структура дерева переносится в одномерный массив.

Адрес любой вершины в массиве вычисляется как

$$\text{адрес} = 2^{k-1}+i-1,$$

где ***k*** — номер уровня вершины, ***i*** — номер слева направо на уровне ***k*** в полном бинарном дереве.

Адрес корня, таким образом, равен единице.

Для любой вершины можно вычислить адреса в массиве левого (адрес_***L***) и правого (адрес_***R***) соответственно потомков:

$$\text{адрес\_}L = 2^{k-1}+(i-1)-1=2^{k-1}+i$$
$$\text{адрес\_}R = 2^{k-1}+(i-1)= 2^{k-1}+(i-1)$$

Главный недостаток рассмотренного способа представления бинарного дерева — это то, что структура представления данных статическая. Размер (или количество элементов) массива выбирается, исходя из максимально возможного количества уровней бинарного дерева. Поэтому, если у дерева ветви имеют разную длину, эффективность работы с таким деревом серьезно снижается. При этом, чем менее полным является дерево, тем менее рационально используется память (больше пустых элементов в массиве).

Рассмотрим пример построения бинарного дерева в виде массива. Объявим массив ***a[i]***, состоящий из 15 элементов (от 1 до 15) в качестве исходной базы построения.

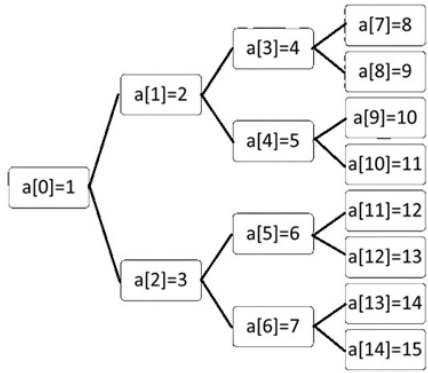


Рис.6.28. Бинарное дерево в виде массива

В результате дерево должно иметь следующий вид, представленный на рисунке 6.28, где ***a[i]*** —это последовательные элементы массива. У этого дерева ветви одинаковой длины, и любой узел имеет по два потомка (кроме листьев этого дерева), следовательно, это полное дерево. А значит, в массиве будут заполнены все последовательные элементы, и работа с таким представлением дерева статическим массивом будет эффективна.

Приведем код программы, построения нелинейного бинарного дерева, представленного массивом на языке программирования Си.

```
#include <fstream.h>
#include <stdio.h>
#include <conio.h>
#include <windows.h>

char bufRus[256];
char* Rus(const char* text)
  {CharToOem(text, bufRus);
   return bufRus;
  }

int main()
 {
  int i,k;
  int a[]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
  cout << Rus("Дерево:") << endl;
  cout << Rus("Корень дерева - a[0] = ") << a[0] << endl;
  k=0;
  for (i=1; i<=14; i++)
    { if (i%2==0) k=k+1;
      if (i%2!=0)
          cout << Rus(" левый сын узла a[") << i-k-1<<"]="<<a[i]<<endl;
      if (i%2==0)
          cout<< Rus(" правый сын узла a[")<<i-k-1<<"]=" <<a[i]<<endl;
    }
  return 0;
 }
```

Результат выполнения программы.

```
Дерево:
Корень дерева — a[0] = 1
 левый сын узла a[0] = 2
 правый сын узла a[0] = 3
 левый сын узла a[1] = 4
 правый сын узла a[1] = 5
 левый сын узла a[2] = 6
 правый сын узла a[2] = 7
 левый сын узла a[3] = 8
```

правый сын узла a[3] = 9
левый сын узла a[5] = 10
правый сын узла a[0] = 11
левый сын узла a[5] = 12
правый сын узла a[5] = 13
левый сын узла a[6] = 14
правый сын узла a[6] = 15
Press any key to continue

Вопросы для контроля

1. Чем отличается структура синтаксического дерева от структуры дерева операций?

2. Почему для полного дерева удобно использовать его представление в виде массива?

3. Для чего используется преобразование синтаксического дерева в дерево операций?

4. Чем характеризуются бинарные деревья?

5. Как отображаются узлы бинарного дерева в элементы статического массива

Глава 7. Основные технологии доступа к памяти

Учебные цели:

- дать представление о принципах организации и распределения памяти;
- дать представление о видах переменных и областей памяти;
- дать представление о глобальной и локальной памяти.

7.1. Принципы организации и распределения памяти.

Организацию и распределение памяти можно рассматривать с двух позиций: с точки зрения процесса компиляции и с точки зрения функционирования операционной системы.

Сначала рассмотрим организацию и распределение памяти при компиляции.

Дадим определения основным понятиям с этой точки зрения.

Распределение памяти — это процесс, который ставит в соответствие лексическим единицам исходной программы (лексемам) информацию, необходимую для работы с этой лексической единицей на протяжении всего времени её существования, а именно, адрес, размер и атрибуты области памяти.

Область памяти — это блок ячеек оперативной памяти, который выделяется для данных, логически объединённых в соответствии с правилами семантики исходного языка.

Распределение памяти работает с *лексическими единицами языка* (лексемами, см главу 3, пункт 3.1), то есть с переменными, константами, функциями и тому подобными элементами, представляющими лексемы, — а также с *информацией* об этих единицах, которая формируется на этапах лексического и синтаксического анализа. Для процесса распределения памяти в компиляторе исходными данными служат таблица идентификаторов, построенная лексическим анализатором, и декларативная (описательная) часть программы (так называемая “область описаний”), полученная в результате синтаксического анализа. Однако, не во всех языках программирования декларативная часть программы присутствует явно, например, в языке ***PHP*** нет явного описания переменных. Некоторые языки предусматривают дополнительные семантические правила для описания констант и переменных; кроме того, перед распределением памяти надо выполнить идентификацию лексических единиц языка. Поэтому реальное распределение памяти выполняется уже после семантического анализа текста исходной программы.

Процесс распределения памяти в современных компиляторах, как правило, работает с относительными, а не абсолютными адресами ячеек памяти. Распределение памяти выполняется перед генерацией кода результирующей программы, потому что его результаты должны быть использованы в процессе генерации кода.

Во всех языках программирования существует понятие так называемых «базовых типов данных» (иначе их называют основными или «скалярными» типами). Для лексической единицы базового типа размер области памяти, ей предназначаемой, в языке программирования считается заранее известным. Объём памяти для лексемы определяется семантикой языка и архитектурой вычислительной системы, на которой должна будет выполняться созданная компилятором результирующая программа. Размер памяти, выделяемой под лексические единицы базовых типов, не зависит от версии компилятора, а зависит от архитектуры вычислительной системы. Такой подход обеспечивает совместимость и переносимость исходных программ. Поэтому ведущие разработчики компиляторов строго придерживаются этого правила.

Для более сложных структур данных используются правила распределения памяти, которые определяются семантикой (то есть, смыслом) этих структур. Эти правила достаточно просты и, в принципе, одинаковы для всех языков программирования.

Правила распределения памяти под основные виды структур данных следующие:

— *для массивов* объём памяти определяется произведением числа элементов в массиве на размер памяти для одного элемента;
— *для строк* объём памяти определяется произведением числа символов в строке на размер памяти для одного символа (однако, в Си/Си++ подобных языках и во многих других языках программирования строки содержат еще и дополнительную служебную информацию фиксированного объема, например, в Си/Си++ в конце строки добавляется символ конца строки «\0»);
— *для структур* (записей с именованными полями) объём памяти определяется суммой размеров памяти по всем полям структуры с учётом выравнивания границ;
— *для объединений* (союзов, общих областей, записей с вариантами) объём памяти определяется размером максимального поля в объединении;
— *для реализации объектов* (классов) объём памяти определяется размером памяти для структуры с такими же именованными полями плюс память под служебную информацию объектно-ориентированного языка (как правило, фиксированного объема).

Объем памяти для более сложных структур данных входного языка, отводимой под эти структуры данных, вычисляется рекурсивно.

Например, если определен *массив структур*, то при вычислении объема памяти, отводимого под весь этот массив сначала будет вызвана процедура для вычисления объема памяти, необходимой для одного элемента массива (являющегося структурой), то есть процедура

вычисления объёма памяти для структуры, а затем этот объём будет умножен на количество элементов массива.

Такой подход определения объема занимаемой памяти очень удобен, если декларативная часть языка представлена в виде дерева типов. Тогда для вычисления объема памяти, занимаемой типом из каждой вершины дерева, нужно вычислить объем памяти для всех потомков этой вершины, а потом применить формулу, связанную непосредственно с самой вершиной (этот механизм подобен механизму СУ-перевода (см. главу 4) — синтаксически управляемого перевода, — применяемому при генерации кода). Такого типа древовидные конструкции строит синтаксический разбор для декларативной части языка.

Далеко не все лексемы языка требуют для себя выделения памяти. Под какие элементы языка нужно выделять ячейки памяти, а под какие не нужно, определяется исключительно реализацией компилятора и архитектурой используемой вычислительной системы. Например, целочисленные константы можно разместить в статической памяти, а можно и непосредственно в тексте результирующей программы (это позволяют практически все современные вычислительные системы), то же самое относится и к константам с плавающей точкой, но их размещение в тексте программы допустимо не всегда. Кроме того, в целях экономии памяти, занимаемой результирующей программой, под различные элементы языка компилятор может выделять одни и те же ячейки памяти. Например, в одной и той же области памяти могут быть размещены одинаковые строковые константы или две различные локальные переменные, которые никогда не используются одновременно.

Изучая процесс выделения объемов памяти, занимаемой различными лексическими единицами и структурами данных языка, следует отметить еще один факт, связанный с выравниванием отводимых для различных лексических единиц границ областей памяти.

Архитектура многих современных вычислительных систем предусматривает тот факт, что обработка данных выполняется более эффективно, если адрес, по которому выбираются данные, кратен определенному числу байт (как правило, это 2, 4, 8 или 16 байтам).

Современные компиляторы учитывают эти особенности вычислительных систем, на которые ориентирована результирующая программа. При распределении данных они могут размещать области памяти под лексические единицы наиболее оптимальным образом. Поскольку не всегда размер памяти, отводимой под лексическую единицу, кратен указанному числу байт, то в общем объеме памяти, отводимой под результирующую программу, могут появляться неиспользуемые области.

Кроме всего сказанного, память можно разделить на локальную и глобальную память, динамическую и статическую память (рассмотрим их в этой главе, но несколько позже.

Теперь обратим внимание на более общие схемы организации и распределение памяти с точки зрения функционирования операционной системы.

Рассмотрим сначала наиболее простые схемы.

Простое непрерывное распределение и распределение с перекрытием (оверлейные структуры).

Простое непрерывное распределение — это самая простая схема, согласно которой вся память может быть условно разделена на три части:

- область, занимаемая операционной системой;
- область, в которой размещается исполняемая задача;
- незанятая ничем (свободная) область памяти.

В этой схеме не предполагается, что операционная система не поддерживает мультипрограммирование (одновременное выполнение нескольких задач), поэтому не возникает проблемы распределения памяти между несколькими задачами. Программные модули, необходимые для *всех* программ, располагаются в области самой операционной системы, а вся оставшаяся память может быть предоставлена задаче. При этом получается, что в этом случае область памяти, предоставляемая задаче, оказывается непрерывной, что облегчает работу системы программирования.

Операционная система строится таким образом, что постоянно в оперативной памяти располагается только самая нужная ее часть — резидентное ядро, — остальные модули операционной системы могут загружаться в оперативную память по необходимости, и после своего выполнения вновь освобождать память.

При такой схеме возможны два вида потерь вычислительных ресурсов:

- потеря процессорного времени, потому что процессор простаивает, пока задача ожидает завершения операции ввода/вывода, и
- потеря самой оперативной памяти, потому что далеко не каждая программа использует всю память, а режим работы в этом случае однопрограммный.

Подобная простая схема распределения памяти реализована в ***DOS***.

Метод *распределения памяти с перекрытием,* то есть с использованием оверлейных структур (***overlay*** — перекрытие, расположение поверх чего-то), предполагает, что вся исполняемая программа может быть разбита на относительно независимые части (сегменты). Каждая программа, представляющая оверлейную структуру, имеет одну главную часть (***main***) и несколько сегментов (***segment***); причем в памяти компьютера одновременно могут находиться одна главная ее часть и один или несколько не

перекрывающихся сегментов (в зависимости от объема оперативной памяти, предоставленного исполняемой программе).

Пока в оперативной памяти располагаются выполняющиеся сегменты, остальные находятся во внешней памяти. После того, как текущий (выполняющийся) сегмент завершит свое выполнение, возможны два варианта:

- либо он сам (если данный сегмент не нужно сохранять в памяти в его текущем состоянии) обращается к операционной системе с указанием, какой сегмент дожжен быть загружен в память следующим;
- либо он возвращает управление главному сегменту задачи (в модуль ***main***), и далее ***main*** обращается к операционной системе с указанием, какой сегмент сохранить, а какой сегмент загрузить в оперативную память, и вновь отдает управление одному из сегментов, располагающихся в памяти (сегмент становится активным.

В далеких 80-х годах программисты сами должны были включать в тексты своих программ соответствующие обращения к операционной системе и тщательно планировать, какие сегменты могут находиться в оперативной памяти одновременно, чтобы их адресные пространства не пересекались. В настоящее время система программирования автоматически подставляет эти вызовы в код программы с помощью специальной структуры, называемой, *оверлеем*.

Распределение со статическими и динамическими разделами.

Теперь рассмотрим *мультизадачную операционную систему*, то есть поддерживающую одновременную работу нескольких задач пользователя/пользователей. В этом случае операционная система должна обладать механизмом разделения центральной памяти между совместно выполняющимися процессами. Для этого оперативная память разбивается на разделы с выделением каждому процессу (выполняющейся задаче) своего раздела. Размер и расположение разделов могут быть либо заранее заданы (разделы фиксированного размера), либо назначаться динамически в процессе выполнения заданий (разделы переменного размера).

Задания пользователя		
Задание	**Длина** (в шестнадцатеричном виде)	**Объем памяти** (в байтах)
1	**A000**	**7000**
2	**14000**	**14000**
3	**A8000**	**8000**
4	**4000**	**4000**
5	**E000**	**10000**
6	**B000**	**8500**
7	**C000**	**9000**

Рис. 7.1. Исходные данные для пояснения способов распределения памяти

Рассмотрим для наглядности пример функционирования такой схемы (информация о заданиях пользователя представлена на рисунке 7.1.

Предполагается, что уровень мультипрограммирования (то есть количество одновременно выполняющихся заданий) ограничен только числом самих заданий.

Пусть полный объем доступной оперативной памяти предполагается равным 56000 байт; сама операционная система занимает первые 10000 байтов. Пусть память, не занятая операционной системой, состоит из четырёх разделов. Раздел первый начинается с адреса 10000 сразу за операционной системой и имеет длину 18000 байт, разделы второй и третий имеют объем по 10000 байтов каждый, раздел четвёртый имеет объем 8000 байтов.

I. Простая схема *распределения с разделением фиксированного размера* (когда каждое входящее задание загружается в наименьший подходящий по объему раздел (рис. 7.2).

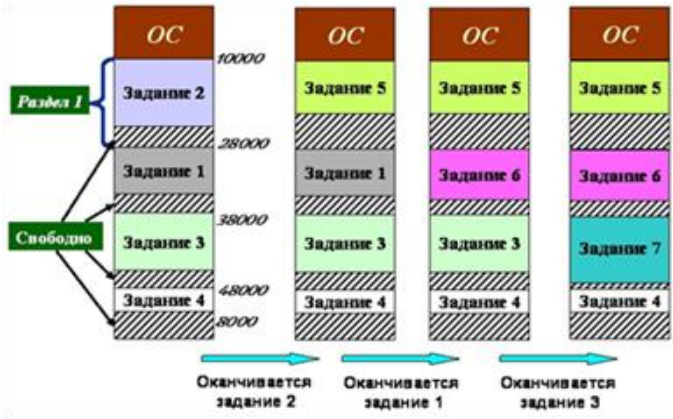


Рис. 7.2 Распределение памяти с разделами фиксированного размера

Если размер раздела превосходит размер задания, то оставшаяся внутри раздела память не используется. Система, имея вначале пустыми все четыре раздела, сначала загрузит задание №1 в раздел №2. Затем задание №2 будет загружено в единственный, достаточно большой для него раздел №1. Задания №3 и №4 загружаются в разделы №3 и ;4. После этого все разделы оказываются занятыми, поэтому больше заданий загрузить нельзя.

Однажды загруженное в раздел задание остается в разделе до конца своего выполнения. После того, как задание завершится, память занимаемого им раздела освобождается и вновь становится доступной для использования.

По окончании задания №2 в раздел №1 загрузится задание №5.

Необходимо отметить, что при этой схеме расположение и объём разделов жёстко фиксировано изначально (не изменяется в зависимости от размера, занимающего раздел задания).

При такой схеме начальный выбор величины раздела в схеме с фиксированными разделами очень важен. Число больших разделов должно быть достаточным, чтобы длинные задания могли выполнятся без слишком большой задержки. Однако если больших разделов слишком много, а выполняются короткие задания, значительный объём памяти в разделах не используется, а другие задания в этот момент не загружаются в оперативную память — и это неэффективно.

Использование *разделов фиксированного размера* наиболее эффективно, когда размеры большинства заданий находятся в пределах конкретных объемов разделов, а распределение заданий в разделы меняется часто. Это позволяет эффективно использовать доступную память посредством выделения набора разделов ожидаемому множеству заданий.

При *небольшом объеме памяти* и, следовательно, небольшом количестве разделов увеличить количество параллельно выполняемых приложений можно за счет свопинга (***swapping***). При свопинге задача может быть выгружена на внешний диск, а на ее место загружается либо более привилегированная, либо проста готовая к выполнению другая задача, находившаяся на диске в приостановленном состоянии. При свопинге из основной памяти во внешнюю (и обратно) перемещается вся программа (со всем свои контентом, то есть информацией о состоянии), а не ее отдельная часть.

Существует проблема при организации мультипрограммного режима работы вычислительной системы в защите от ошибок как самой операционной системы от ошибок и преднамеренного вмешательства задач в ее работу, так и задач друг от друга, но современные операционные системы решают ее вполне эффективно.

У рассмотренной схемы распределения памяти есть и недостаток — *большая* фрагментация оперативной памяти. Сократить потери в её использовании можно двумя способами:

- выделять раздел ровно такого объема, который нужен под текущую задачу;
- размещать задачу не в одной непрерывной области, а в нескольких областях (реализовано в нескольких способах организации виртуальной памяти).

Поэтому появились другие схемы распределения памяти.

Разделы с подвижными границами.

Для каждого загружаемого задания создается новый раздел с *размерами, соответствующими заданию*. После окончания задания отведенная ему память освобождается и может быть использована при распределении других разделов (рис. 7.3).

Вначале вся оперативная память (кроме той, что отведена для операционной системы) не распределена, так как заранее нет определенных разделов. Раздел для задания №1 создается при его загрузке. Предположим, что этот раздел находится непосредственно за разделом операционной системы. Затем заданию №2 отводится раздел, следующий сразу за заданием №1 и так далее. Объем свободной памяти, оставшейся за заданием №5, для загрузки очередного задания №6 оказывается уже недостаточным.

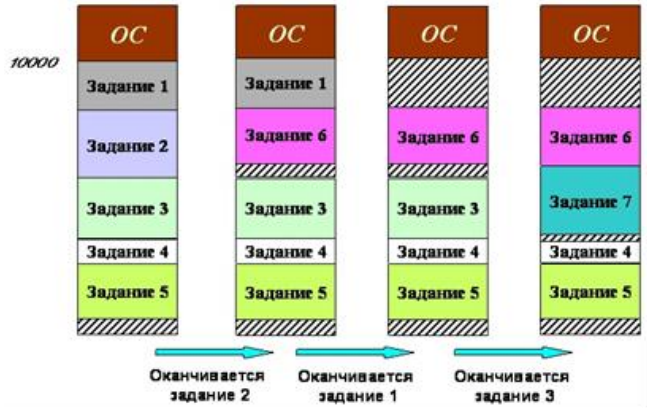


Рис. 7.3.

Распределение памяти с подвижными границами

Когда завершается задание №2, его раздел освобождается, и новый раздел (в границах задания №2 или меньше) отводится заданию №6. Как видно из рисунка, этот новый раздел занимает часть памяти, которая отводилась заданию №2. Остаток от прежнего раздела задания №2 остается свободным. Теперь образовались две несмежные свободные области памяти (после задания №6 и после задания №5); однако ни одна из них не велика настолько, чтобы вместить еще одно задание, и так далее.

При использовании *разделов переменного размера* нет надобности выбирать размер заранее. Однако операционной системе, которая отслеживает какие области памяти уже распределены, а какие свободны, приходится проделывать большую работу. Обычно это выполняется системой при помощи поддерживаемого ею связанного списка свободной области. Этот список просматривается при выделении нового раздела, который размещается либо в первой (первое подходящее размещение), либо в наименьшей (наиболее подходящее размещение) подходящей для него свободной области. Когда раздел освобождается, отведенная ему память объединяется со всеми смежными свободными областями и заносится в список.

Вне зависимости от используемого способа создания разделов необходимо, чтобы операционная система и аппаратные средства обеспечивали *защиту памяти*. При выполнении заданий в одном разделе недопустимо, чтобы оно изменяло ячейки памяти другого раздела или операционной системы.

Единой проблемой для всех общецелевых способов динамического распределения является *фрагментация памяти*. Фрагментация имеет место, когда доступная свободная память разбита на несколько несмежных блоков, каждый из которых слишком мал для использования. Чтобы, например, разместить задание №1, в сумме двух несмежных кусков памяти имеется более, чем достаточно свободной памяти; однако из-за того, что нет ни одного свободного блока достаточно большого размера, оно не может быть загружено.

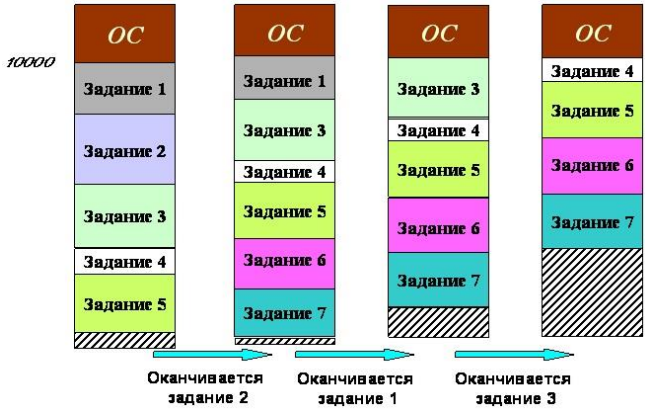


Рис. 7.4. Распределение памяти с перемещающимися разделами

Одно из возможных решений этой проблемы *использование перемещаемых разделов* (рис. 7.4). После окончания каждого задания оставшиеся разделы передвигаются как можно дальше к одному концу памяти. В результате этого вся доступная свободная память собирается в один блок, который больше подходит для распределения новых разделов.

Из рисунка 7.4 видно, что этот способ может привести к более эффективному использованию памяти по сравнению с тем, что достигается с помощью неперемещаемых разделов. Однако копирование заданий из одного места памяти в другое может потребовать значительного количества времени и, самое главное, при перемещении заданий в оперативной памяти невозможно выполнять собственно вычислительные процессы. Этот недостаток часто перевешивает преимущества усовершенствованного использования памяти.

7.2. Виды переменных и областей памяти.

Для хранения различных данных в языках программирования используют переменные. *Переменной* называется область памяти, имеющая имя; это имя иначе называют идентификатором.

Когда программист объявляет переменную, давая ей имя, одновременно с тем же именем он называет и область памяти для хранения значений этой переменной. Когда этой переменной будут присваиваться значения (причем, неоднократно), они будут записываться в эту область памяти.

Хорошим стилем является осмысленное именование переменных. Правила именования переменных, или идентификаторов:

- разрешается использовать строчные и прописные буквы, цифры и символ подчёркивания, который в Си считается буквой;
- первым символом обязательно должна быть буква;
- в имени переменной не должно быть пробелов;
- прописные и строчные буквы в именах переменных различаются, то есть переменные ***a*** и ***A*** — разные переменные;
- имя переменной *не может* совпадать с зарезервированными ключевыми словами;
- в современных версиях компиляторов длина имени практически не ограничена.

Ниже приведём далеко не полный список зарезервированных ключевых слов: ***double int long char float unsigned signed union short enum struct auto break void typedef extern register else if switch case default for do return continue sizeof volatile while***.

В языках Си/Си++ все переменные должны быть объявлены. Это означает, что,

- во-первых, в начале каждой программы или функции необходимо привести список всех используемых переменных,
- а во-вторых, указать тип каждой из них.

Обратите внимание, что при объявлении переменной компилятор отводит ей место в памяти в зависимости от её типа (рис. 7.5).

Вспомним *типы переменных*, используемых в языках программирования Си/СИ++: «базовые простые типы» ***char, int, float, bool***, а также сложные типы.

На примере простых типов ***char***, ***int*** и их вариантов рассмотрим особенности выделения памяти для различных типов.

Тип ***char*** — самый экономный тип, под переменную типа ***char*** отводится один байт памяти (8 бит). Тип ***char*** может быть знаковым и беззнаковым, обозначается, соответственно, как ***signed char*** (знаковый тип) и ***unsigned char*** (беззнаковый тип). Знаковый тип хранит значения в диапазоне от -128 до +127, беззнаковый — от 0 до 255.

Тип	Диапазон (десятичный)	Диапазон (шестнадцатеричный)	Размер
unsigned char	0 ... 255	0x00 ... 0xFF	8 bit
signed char или просто **char**	-128 ... 127	-0x80 ... 0x7F	8 bit
unsigned short int или просто **unsigned int** или **unsigned short**	0 ... 65535	0x0000 ... 0xFFFF	16 bit
signed short int или **signed int** или просто **short** или **int**	-32768 ... 32767	0x8000 ... 0x7FFF	16 bit
unsigned long int или просто **unsigned long**	0 ... 4294967295	0x00000000 ... 0xFFFFFFFF	32 bit
signed long или просто **long**	-2147483648 ... 2147483647	0x80000000 ... 0x7FFFFFFF	32 bit
unsigned long long	0 ... 18446744073709551615	0x0000000000000000 ... 0xFFFFFFFFFFFFFFFF	64 bit
signed long long или просто **long long**	-9223372036854775808 ... 9223372036854775807	0x8000000000000000 ... 0x7FFFFFFFFFFFFFFF	64 bit

Рис. 7.5. Размер памяти переменных в зависимости от типа

Ключевые слова (модификаторы) ***signed*** и ***unsigned*** указывают, как интерпретируется нулевой бит объявляемой переменной, то есть, если указано ключевое слово ***unsigned***, то нулевой бит интерпретируется как старший бит числа, в противном случае нулевой бит интерпретируется как бит для знака.

Рассмотрим подробно виды переменных типа ***int***.

Тип ***int*** обозначает целочисленную величину; под переменную типа ***int*** выделяется два или четыре байта (в зависимости от 32 или 64 битной разрядности ОС соответственно). Тип ***int*** также может быть ***short*** (короткий) или ***long*** (длинный). Для примера, не нарушая общности, будем рассматривать 32 битную ОС (в случае 64 битной ОС размеры переменных будут в 2 раза больше).

Ключевое слово (модификатор) ***short*** ставится после ключевых слов ***signed*** или ***unsigned***. Таким образом, различают следующие типы: ***signed short int, unsigned short int, signed long int, unsigned long int***.

Переменная типа ***signed short int*** (знаковая короткая целая) может принимать значения от -32768 до +32767, ***unsigned short int*** (беззнаковая короткая целая) - от 0 до 65535. Под каждую из них отводится ровно по

два байта памяти (16 бит). При объявлении переменной типа ***signed short int*** ключевые слова ***signed*** и ***short*** могут быть пропущены, и такой тип переменной может быть объявлен просто ***int***. Допускается и объявление этого типа одним ключевым словом ***short.***

Переменная ***unsigned short int*** может быть объявлена как ***unsigned int*** или ***unsigned short.***

Под каждую величину ***signed long int*** или ***unsigned long int*** отводится 4 байта памяти (32 бита). Значения переменных этого типа могут находиться в интервалах от -2147483648 до 2147483647 и от 0 до 4294967295 соответственно.

Существуют также переменные типа ***long long int***, для которых отводится 8 байт памяти (64 бита). Они также могут быть знаковыми и беззнаковыми. Для знакового типа диапазон значений лежит в пределах от -9223372036854775808 до 9223372036854775807, для беззнакового - от 0 до 18446744073709551615. Знаковый тип может быть объявлен и просто двумя ключевыми словами ***long long.***

Пример 7.1.

```
char x;          //Для переменной x будет отведен 1 байт памяти
int a, b, c;     //Для переменных a, b, c будет отведено по 2 байта
                 //(причем адреса для переменных будут выровнены по
                 //четному адресу)
unsigned long long y;  //Для переменной y будет отведено 8 байт
                       //памяти и адрес выровнен по четному адресу
```

Таким образом, будут объявлены переменные ***x, a, b, c, y***. В переменную ***x*** можно будет записывать значения от -128 до 127. В переменные ***a, b, c*** — от -32768 до +32767. В переменную **y** - от 0 до 18446744073709551615.

При объявлении переменную можно проинициализировать, то есть присвоить ей начальное значение. При объявлении переменной ***x*** оператором ***int x = 100;*** переменной ***x*** будет отведена память в 2 байта, выровненная по четному адресу и по этому адресу сразу же будет записано её значение — число 100 (которое в дальнейшем может быть изменено другим присваиванием).

Теперь рассмотрим такой объект языка программирования как *константа.* Константа объявляется добавлением ключевого слова **const** к спецификатору типа; она может иметь любой «базовый тип», но в отличие от переменной немодифицируема. То есть при объявлении константы ей отводится объём памяти, соответствующий объявленному типу, и по этому адресу заносится значение, указанное при объявлении, в дальнейшем это значение не меняется на протяжении выполнения всей программы. Таким образом, константы представляют собой данные, используемые только для чтения. Если после слова ***const*** отсутствует спецификатор типа, то константы рассматриваются как величины со знаком, и им присваивается

тип ***int*** или ***long int*** в соответствии со значением константы: если константа меньше 32768, то ей присваивается тип ***int***, в противном случае ***long int***.

Пример 7.2.

```
const long int k = 25;     //Для константы k будет отведено 4 байта
                           //памяти, адрес для константы выровнен по четному
                           //адресу и  по этому адресу записано целое число 25
const m = -50; // подразумевается const int m=-50
                           // Для константы m будет отведено 2 байта памяти,
                           //адрес для константы выровнен по четному адресу и
                           // по этому адресу записано целое число -50
const n = 100000; // подразумевается const long int n=10000
                           // Для константы n будет отведено 4 байта памяти,
                           //ввиду того, что значение 100000 имеет тип long int
                           //адрес для константы выровнен по четному адресу и
                           // по этому адресу записано целое число 100000
```

Модель памяти в языке Си — система хранения объектов в языке Си.

Способ хранения объекта в языке Си определяет его *время жизни* — часть времени выполнения программы, во время которого объект существует или для него зарезервировано место. Объект имеет постоянный адрес и сохраняет своё последнее значение. Запрещается обращаться к объекту, который перестал существовать, при этом, если при работе с объектом использовался указатель, его значение остаётся неопределённым.

Существует три способа хранения объектов: *автоматический (**auto**), статический (**static**) и динамический (**malloc, calloc, realloc, free**).*

Свойство	Автоматический	Статический	Динамический
Объявление	Объект без связывания и без ***static***	Имеет внутреннее или внешнее *связывание* или объявлен с квалификатором ***static***	Выделен с помощью ***malloc***
Время существования	Блок, в котором объявлен объект	Всё время выполнения программы	От вызова ***malloc*** до вызова ***free***
Инициализация	Отсутствует в случае отсутствия явной инициализации	Происходит один раз до запуска программы	Частично в случае ***calloc***
Размер	Фиксированный, неизменяемый	***Фиксированный, неизменяемый***	Любой, изменяемый
Типичное размещение	Стек или регистры процессора	***Отдельный сегмент памяти***	Куча

Рис. 7.6. Способы хранения объектов языка программирования и их характеристики

Статический объект можно инициализировать либо явно, либо использовать умалчиваемую инициализацию.

При использовании функции ***calloc*** все объекты имеют нулевое значение кроме чисел с плавающей запятой и указателей.

Си-строки, которыми инициализируются указатели ***char***, имеют статический тип хранения и не должны изменяться.

Динамическая память.

Ни один объект не может находиться в динамической памяти без явного указания программиста. Для работы с динамической памятью существуют функции ***malloc***, ***calloc***, ***realloc*** и ***free***. Поскольку функции, выделяющие память, принимают размер переменной типа ***size_t*** (зарезервированная переменная конкретного типа), максимальный объём выделяемой памяти ограничен ***SIZE_T_MAX*** (максимальное значение для конкретного типа, зарезервированное в компиляторе).

Функции ***malloc*** и ***calloc*** выделяют память, которая после использования должна быть освобождена с помощью вызова функции ***free***. После освобождения значение указателя остаётся *неопределённым*. Если функция ***realloc*** создает новый динамический блок, возвращает указатель на этот новый или изменённый блок памяти, если же запрос не может быть удовлетворён, ни адрес, ни размер блока памяти не изменяется.

Пример 7.3.

```
 #include <stdlib.h>
void foo (void **ptr, size_t size)
   {*ptr = realloc (*ptr, size+128); /* утечка памяти, если realloc вернёт
                                                    NULL */
     if (!*ptr)    {   ...   }
   }
```

При работе с динамической памятью возможны утечки памяти и ошибки двойного освобождения блока.

Пример 7.4.

```
#include <stdlib.h>
#include <string.h>
static int x;        /* 0 по умолчанию, существует всё время выполнения */
static int y=45;     /* 45, существует всё время выполнения */
```

```
int cnt(void)
 {static int i=0;      /*статический тип, инициализируется нулём только
                       при запуске программы, а не каждый вызов функции */
  int j=-1;            /*автоматический тип, инициализируется каждый
                       раз при вызове функции -1*/
  i++;/*  увеличивается на 1 в статической области памяти
                       каждый запуск функции*/
  j++;                 /* увеличивается на 1 локальная переменная */
  return (i+j); /*при первом вызове с запуска программы функция
                       вернёт 1, при втором вызове 2, ... */
 }
int main (void)
 {char arr[50] = "This is object of automatic storage duration";
       /* имеет автоматический тип, существует до выхода из main,
          начальные 45 элементов массива инициализированы элементами
          строки с закрывающим нулём, остальные не определены */
  char *line = "Simple line"; /*автоматический тип, существует до
       выхода из main, line инициализирован указателем на константу */
  int y; /* значение не определено, существует до выхода из main*/
  int z=10;       /* значение определено, существует до выхода из main*/
  char *ptr;      /* значение указателя не определено */
  ptr = malloc (50); /*значение по указателю не определено, объект по
                       указателю существует до вызова free */
  strcpy (ptr, arr);
  free (ptr);
  return 0;
 }
```

7.3. Глобальная и локальная память.

Переменная — именованная область памяти, имеющая тип. Данные, находящиеся в переменной (то есть по данному адресу памяти), называются *значением* этой переменной.

Область видимости переменной — это тот участок кода (функция, цикл, пространство имени), в котором эта переменная объявлена. Вне этого участка компилятор её не видит (она недоступна).

По зоне видимости различают *локальные* и *глобальные* переменные.

Локальность - это ограниченность места дееспособности.

Локальная переменная — переменная, объявленная внутри какой-либо функции. Областью видимости локальных переменных является тело функции, в которой эта переменная объявлена. Локальная переменная может быть проинициализирована константой или выражением, соответствующими ее типу.

Глобальная переменная — переменная, объявленная за пределами всех функций. Областью видимости глобальных переменных является вся программа. Глобальная переменная не локализована ни на каком уровне. Глобальная переменная может быть проинициализирована только соответствующей ее типу константой (но не выражением).

Инициализация глобальных переменных производится однократно перед началом исполнения всех пользовательских или специальных функций.

Локальные доступны только конкретной подпрограмме, глобальные — всей программе. С распространением модульного и объектного программирования, появились ещё и *общие* переменные (доступные для определённых уровней иерархии подпрограмм).

Переменные, описанные, например, внутри функций, не могут быть использованы за её пределами. В качестве примера можно рассмотреть такой код на Си++(пример 7.5).

Пример 7.5

```
#include <iostream
using namespace std;
 int i = 2;    //глобальная переменная (видна в любом участке кода программы)
int sum()
 {int k = 2;        //локальная переменная (видна только внутри функции sum( ))
  return i + k;
 }
int main()
 {cout << i << endl << sum() << endl; }
```

Разберёмся с этим примером подробнее. Переменная ***i*** описана вне любых функций в программе. Ее область видимости и действия — вся программа без ограничения. Ввиду того, что она доступна во всей программе, во всех ее функциях и блоках операторов, заключенных в ***{ },*** её называют *глобальной переменной*.

В примере 7.5 присутствует функция ***sum(),*** которая что-то делает. Внутри функции ***sum( )*** описана вторая переменная **—** переменная ***k.*** Эта переменная ***k*** находится внутри функции ***sum***, поэтому не может быть использована за пределами функции. То есть, если в функции ***main( )*** (вне функции ***sum( )***) дописать еще и вывод этой переменной ***k*** на экран, компилятор выдаст ошибку (пример 7.6), потому что компилятор переменную ***k*** не видит вне функции ***sum( ).***

Пример 7.6.

```
#include <iostream>
using namespace std;
  int i = 2; // глобальная переменная (видна в любом участке кода)
```

```
 int sum( )
   {int k = 2;  //локальная переменная (видна только внутри функции sum( ))
    return i + k;
  }
int main()
  { cout << i << endl << sum() << endl;
    cout << k << endl; // попытка обратиться к переменной k
}
```

Переменная ***k*** **—** *локальная*, и ее область видимости определена открывающей и закрывающей фигурными скобками функции ***sum()*** — ***{…},*** использовать её нельзя вне этой функции. После компиляции получим следующее сообщение:

IntelliSense: идентификатор ***k*** не определен **main.cpp**

Аналогичным образом область видимости распространяется и на внутренние блоки. Рассмотрим следующий код на Си++ (пример 7.7).

Пример 7.7.

```
#include <iostream>
using namespace std;
int i = 2;
int sum()
 {int k = 2;
   for (int i = 0; i < 10; i++) // эта i - локальная, объявленная в теле цикла
            k += 1;
    return i + k; // эта i - глобальная, объявленная вне функций
 }
int main()
  { cout << i << endl << sum() << endl;}
```

В этом примере определены следующие области видимости:

- глобальная – ***i = 2*** принадлежит глобальной области;
- локальная относительно функции ***sum( )*** — переменная ***k;***
- локальная относительно цикла ***for( )*** – вторая ***i.***

Несмотря на то, что у оператора ***for*** есть своя область видимости, сам оператор ***for*** принадлежит функции ***sum( ).*** А значит, он и все, что в нем находится, подчиняется локальной области видимости этой функции. То есть, все переменные, определенные в функции ***sum( )***, также действительны и в теле ***for,*** что позволяет работать оператору ***k += 1*** (он же ***k = k+1*** или ***k++***).

Интереснее дело обстоит с переменной ***i,*** которая описана внутри оператора ***for***. Несмотря на имя, идентичное глобальной переменной, описанной выше, это уже другая переменная.

Как только цикл отработал, переменная ***i,*** описанная в нём, становится недоступной. То есть, когда переменная описывается в параметрах цикла, менеджер памяти резервирует под неё память, а когда её область видимости заканчивается — эта память освобождается. Менеджер памяти у себя помечает, что этот участок памяти более не принадлежит кому-либо, и его можно отдать под запись других данных.

Для закрепления материала рассмотрим еще один легкий пример на Си++ (пример 7.8).

Пример 7.8.

```
#include <iostream>
using namespace std;
int main()
 {int a = 0;      //локальная переменная первого блока
    {int a = 2;     //локальная переменная второго блока
      cout << a << endl;
     }              //кончилась область видимости второго блока
  cout << a << endl;
 } кончилась область видимости первого блока
```

В примере 7.8 области видимости описаны блоками операторов, ограничивающимися ***{ }***. То есть первый ***cout*** выведет **2**, потому что определен во втором блоке операторов, в котором сработает этот ***cout.*** В этом же блоке определена *локальная* переменная с именем ***a,*** из этой переменной ***a*** функция вывода ***cout,*** прежде всего, и возьмет значение.

Результат работы этой простой программы.

```
2
0
Для продолжения нажмите любую клавишу . . .
```

За пределами второго блока, ограниченного ***{ },*** оператор ***cout*** выведет ***0,*** поскольку для него не существует той переменной ***a***, которая содержала значение **2**. Область видимости последнего оператора ***cout*** совпадает с другой переменной ***a***, которая содержит ***0***, и определена выше вложенного блока.

Замечание. В реальном программировании *никогда* не надо давать ***одинаковые*** имена переменным (за исключением тех, что классически используют для счетчиков циклов, например, ***i, j, k.*** Если имеется в программе 10 циклов в коде, и они не вложенные, то для каждого из них можно объявлять и определять счетчик с именем ***i***. Это нормально. В других случаях, всегда нужно давать переменным *уникальные* имена, чтобы было легче разбираться в коде программы.

В отличие от локальных переменных глобальные переменные видны всей программе и могут использоваться любым участком кода. Они

хранят свои значения на протяжении всей работы программы. Глобальные переменные создаются путем объявления вне функции. К ним можно получить доступ в любом выражении, независимо от того, в какой функции находится данное выражение.

В программе примера 7.9 переменная ***count*** объявлена вне функций, объявляется перед основной функцией ***main().*** Однако, она может быть помещена *в любое место* программы до первого использования, только не внутри функции. Общепринятым является объявление глобальных переменных в начале программы.

Пример 7.9.

```
#include <stdio.h>
using namespace std;
void func1(void) , func2(void);   //объявление функций func1 и func2
int count; /* count является глобальной переменной */
int main(void)
  {count = 100;
   func1 ();      //вызов функции func1
   return 0; /* сообщение об удачном завершении работы */
  }
void func1 (void)    // описание функции func1
 {func2 ();      //вызов функции func2
  printf("счетчик %d", count); /* выведет 100 */
 }
void func2(void);    //описание функции func2
 {int count;
  for(count=1; count<10; count++)       putchar('  ');
 }
```

Рассмотрим подробнее код этого фрагмента программы. Хотя ни ***main(),*** ни ***func1()*** не объявляют переменную ***count,*** они оба могут ее использовать, так как она описана как глобальная. Функция ***func2( )*** объявляет свою локальную переменную ***count.*** Когда ***func2()*** обращается к ***count***, она обращается только к своей локальной переменной, а не к глобальной. Напомним, что, если глобальная и локальная переменные имеют одно и то же имя, все ссылки на имя внутри функции, где объявлена локальная переменная, будут относиться к локальной переменной, и не будут иметь никакого влияния на глобальную. Это семантическое правило устанавливает порядок иерархии и устраняет неопределенность выполнения операций, что очень удобно.

Глобальные переменные хранятся в фиксированной области памяти, устанавливаемой компилятором. Глобальные переменные чрезвычайно

полезны, когда одни и те же данные используются в нескольких функциях программы. Следует избегать ненужного использования глобальных переменных по трем причинам.

1. Они используют память в течение всего времени работы программы, а не тогда, когда они необходимы
2. Использование глобальных переменных вместо локальных приводит к тому, что функции становятся более частными, поскольку зависят от переменных, определяемых снаружи рассматриваемой области.
3. Использование большого числа глобальных переменных может вызвать ошибки в программе из-за неизвестных и нежелательных эффектов.

Одним из важных требований структурных языков является разделение кода и данных. В языке Си разделение достигается благодаря использованию локальных переменных и функций. Например, ниже показаны два способа написания ***mul( )*** — простой функции, вычисляющей произведение двух целых чисел.

Два способа написания **mul( )**

Общий	***Частный***
int mul(int x, int y) {return(x*y);}	***int x, y; int mui(void) {return(x*y);}***

Обе функции возвращают произведение переменных ***x*** и ***y***.

При этом, *общая или параметризированная* версия может использоваться для вычисления произведения ***любых*** двух чисел, в то время как *частная версия* может использоваться для вычисления произведения ***только глобальных*** переменных ***x*** и ***y***.

Внешняя переменная — это переменная, описанная вне функции (в одном и том же файле или в другом), значение которой доступно из любой части программы. Внешняя переменная объявляется за пределами всех функций и является глобальной, область её видимости - вся программа. Внешнюю переменную можно также описать в функции, которая использует ее, при помощи ключевого слова (модификатора) ***extern***.

```
extern int Number;          // Внешняя переменная целого типа
```

Внешние переменные указываются в головной части программы, а именно перед любой функцией, в которой имеется обращение к внешней переменной. Использование внешних переменных очень удобно, если время от времени возникает необходимость запустить программу на выполнение с иными значениями переменных.

Область видимости определяет класс памяти, но иногда локальные (с точки зрения области видимости идентификатора) переменные могут быть *статическими* (по классу памяти).

Пример использования *статической локальной* переменной в функции ***calls_counter()*** на языке Си (пример 7.10).

Пример 7.10.

```
int calls_counter()
   {static int the_counter = 0;
    the_counter++;
    return the_counter;
   }
```

Все массивы являются *статическими* изначально, то есть имеют вид ***static***, даже если при инициализации явно это не указано.

Ограничение зоны видимости позволяет использовать *одинаковые* имена переменных в разных подпрограммах, при этом защищает от ошибок взаимного использования переменных.

Замечание. Переменные должны быть, по возможности, локальными.

Если управление в программе находится внутри какой-либо функции, то значения локальных переменных, объявленных в другой функции, недоступны. Значение любой глобальной переменной доступно из любой стандартной или пользовательской функции.

Рассмотрим простой пример 7.11, фрагмент кода которой представляет основную программу и функцию, вызываемую из основной программы.

Пример 7.11. *Составить программу, считающую тики.*

```
int start()              // Специальная функция start()
  {Tick++;               // Счётчик тиков
   printf ("Поступил тик № ",Tick);   // Сообщение, содержащее номер
   return 0;             // Оператор выхода из start()
  }
int main()
    {int i, Tick = 0;    // Глобальная переменная
     for (i = Tick; i < 100; i++)   { start();}
     return 0;
    }
```

В этой программе используется всего одна *глобальная* переменная ***Tick***. Она является глобальной, так как объявлена и инициализирована (в основной программе ***main()***) за пределами описания пользовательской функции ***start()***. Это значит, что от тика к тику значение этой переменной будет сохраняться. Рассмотрим подробности исполнения программы.

Отметим, что пользовательская функция ***start()*** запускается на выполнение в момент поступления очередного тика. В момент исполнения этого фрагмента кода произойдут следующие события:

1. Исполнение кода начинается с исполнения ***main()***. Объявление глобальной переменной ***Tick*** проинициализировано константой 0, поэтому её значение на этом этапе равно нулю.
2. Управление удерживается в основном фрагменте до поступления тика.
3. Поступил тик. Управление передаётся пользовательской функции ***start()***.
   3.1. В рамках исполнения пользовательской функции ***start()*** управление передаётся оператору:
   Tick++; // Счётчик тиков
   В результате исполнения этого оператора значение переменной ***Tick*** увеличится на 1 (целую единицу).
   3.2. Управление передаётся оператору:
   printf ("Поступил тик № ",Tick); //Сообщение, содержащее номер
   Исполнение стандартной функции ***printf ()*** приведёт к появлению сообщения:
   «***Поступил* тик № 1**»
   3.3. Управление передаётся оператору:
   return 0; // Оператор выхода из ***start()***
   В результате его исполнения пользовательская функция ***start()*** заканчивает свою работу, управление передаётся в главную часть фрагмента кода (в ***main()***).
   При этом глобальная переменная продолжает своё существование, её значение сохраняется равным 1.

Далее события будут повторяться, начиная с пункта 2. Переменная ***Tick*** снова будет участвовать в вычислениях, однако на втором тике, в момент запуска на исполнение функции ***start()*** (её значение равно 1) поэтому в результате исполнения оператора:

Tick++; // Счётчик тиков

значение переменной ***Tick*** увеличится на 1 и теперь уже будет равно 2. Исполнение функции ***printf ()*** приведёт к появлению сообщения:

«**Поступил тик № 2**»

Таким образом, значение переменной ***Tick*** будет увеличиваться на 1 при каждом запуске пользовательской функции ***start()***, то есть на каждом тике.

Решение подобных задач становится возможным только в случае использования переменных, сохраняющих своё значение после выхода из функции (в данном случае использована глобальная переменная). Использовать для той же цели локальные переменные бессмысленно: *локальная переменная теряет своё значение по окончании исполнения функции, в которой она объявлена.*

В этом очень легко убедиться, запустив на выполнение код фрагмента ниже, в котором переменная ***Tick*** объявлена как локальная переменная (то есть программа содержит алгоритмическую ошибку). Для этого достаточно описать пользовательскую функцию ***start()***.

Пример 7.12. *Видоизменённый пример 7.11 с алгоритмической ошибкой*

```
int start()              // Пользовательская функция start()
  {int Tick = 0;         // Локальная переменная
  Tick++;                // Счётчик тиков
  printf ("Поступил тик № ",Tick);   // Сообщение, содержащее номер
  return 0;                          // Оператор выхода из start()
 }
```

С точки зрения синтаксиса в представленном коде ошибок нет. Эта программа может быть успешно скомпилирована и запущена на исполнение. И она будет работать, однако, всё время будет сообщать один и тот же результат:

«**Поступил тик № 1**»

И это естественно, потому что переменная ***Tick*** инициализирована значением «***0***» в начале исполнения пользовательской функции ***start()*** при каждом её запуске. Последующее увеличение этого значения на единицу приведёт к тому, что к моменту сообщения значение ***Tick*** всякий раз будет равно ***1***.

Статические переменные

На физическом уровне локальные переменные представлены во *временной области памяти* соответствующей функции.

Существует способ расположить переменную, объявленную внутри функции, в *постоянной памяти* программы. Для этого при объявлении переменной перед типом переменной необходимо указать модификатор ***static***:

```
static int Tick;          // Статическая переменная целого типа
```

Пример 7.13. Решение задачи примера 7.11 с использованием статической переменной

```
int start()                  // Специальная функция start()
  {static int Tick;          // Статическая локальная переменная
  Tick++;                    // Счётчик тиков
  printft("Поступил тик № ",Tick);   // Сообщение, содержащее номер
  return 0;                  // Оператор выхода из start()
  }
```

Исполнение кодов примеров 7.11 и 7.13 дают одинаковый результат.

Статические переменные инициализируются однократно. Каждая статическая переменная может быть проинициализирована соответствующей ее типу константой (в отличие от простой локальной переменной, которая может быть проинициализирована любым выражением).

Статические переменные хранятся в постоянной области памяти программы, их значения не теряются при выходе из функции. Вместе с тем, статические переменные имеют ограничение, свойственное локальным переменным — областью видимости статической переменной остаётся функция, внутри которой эта переменная объявлена, — в отличие от глобальных переменных, значение которых доступно из любого места программы.

Когда ***static*** применяется к локальной переменной, это приводит к тому, что компилятор создает долговременную область для хранения переменной почти таким же способом, как это делается для глобальной переменной. Ключевое различие между статической локальной и глобальной переменными заключается в том, что статическая локальная переменная остается известной только в том блоке, в котором она была объявлена. Проще говоря, статическая локальная переменная — это локальная переменная, сохраняющая свое значение между вызовами функций.

Наличие статических локальных переменных очень важно для создания самостоятельных функций поскольку имеется несколько типов подпрограмм, сохраняющих значение между вызовами. Если использование статических переменных недопустимо, то следует использовать глобальные переменные, но это может привести к побочным эффектам. Ниже приведен простой пример того, как статитеская локальная переменная может использоваться в функции ***count()*** (пример 7.14).

Пример 7.14.

```
#include <stdio.h>
#include <conio.h>
int count (int i) ;   //Объявление функции
int main( )
   {do {count(0);}
    while(!kbhit());      /*Функция kbhit() возвращает истину, если нажата какая-либо клавиша на клавиатуре. В противном случае возвращается 0. В любом случае код клавиши не удаляется из входного буфера. Функция kbhit() содержится в библиотеке <conio.h>*/
    printf("count called %d times", count (1));
    return 0;
   }
int count (int i) /* Описание функции count( )*/
    {static int c = 0;      /*Описание переменной c как локальной
                             статической*/
     if (i) {return c;}
            else {c++;}
     return 0;
    }
```

Иногда полезно знать, как часто функция вызывается во время работы программы. Это возможно сделать через использование глобальных переменных, но лучшим вариантом является использование функции, сохраняющей информацию внутри себя, как это сделано в функции ***count()***.

В данном примере, если ***count()*** вызывается со значением ***0***, то переменная ***c*** увеличивается. (Скорее всего, в настоящих приложениях функция ***count()*** будет также выполнять некоторую другую полезную работу.)

Если ***count()*** вызывается с любым другим значением, то она возвращает число сделанных вызовов. Подсчет числа вызовов функции может быть полезен при разработке программы, которая вызывает эти функции достаточно часто, и требуется привлечь к вызовам внимание.

Вернемся к примерам *внешней переменной*. Описания внешней переменной могут выглядеть как в следующем примере (пример 7.15).

Пример 7.15.

```
int errupt;   /* Три переменные, описанные вне функции */
char coal;
double up;

main( )
  {extern int errupt;  /* объявлено, что три переменные в следующих*/
   extern char coal; /* строках являются внешними */
   extern double up;
```

Группу **extern**-описаний можно совсем опустить, если исходные определения переменных появляются в *том же файле и перед функцией*, которая их использует. Включение ключевого слова **extern** позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом файле. (Оба файла, конечно, должны быть скомпилированы, связаны или собраны в одно и то же время).

Если слово ***extern*** не включено в описание *внутри* функции, то под этим именем создается *новая автоматическая переменная*. Вы можете пометить вторую переменную как "автоматическую" с помощью слова ***auto*** и тем самым показать, что это ваше намерение, а не оплошность. Три примера демонстрируют четыре возможных комбинация описаний (пример 7.16):

Пример 7.16.

```
int hocus;
int main( )
  {extern int hocus;       /*переменная hocus описана внешней */
          ...
  }
int magic( )
  {extern int hocus;
     ...
  }
```

Здесь есть одна внешняя переменная ***hocus***, и она известна обеим функциям. ***main( )*** и ***magic( )*** (пример 7.17).

Пример 7.17.

```
int hocus ;
int main( )
   {extern int hocus;   /*Переменная hocus описана внешней */
        ...
   }
int magic( )
   {/*Переменная hocus не описана совсем */
        ...
   }
```

Снова есть одна внешняя переменная ***hocus***, известная обеим функциям. На этот раз она известна функции ***magic( )*** *по умолчанию* (пример 7.18).

Пример 7.18.

```
int hocus;
int main( )
   {int  hocus;      /*Переменная hocus описана и является автоматической по
                                умолчанию*/
     ...
   }
magic( )
   {auto int hocus; /*Переменная hocus описана автоматической*/
     ...
   }
```

В этом примере созданы три разные переменные с одинаковым именем.

В функции ***main( )*** переменная ***hocus***

- является автоматической по умолчанию и
- локальной,

в функции ***magic( )*** переменная ***hocus***

- явно описана автоматической и
- известна только для ***magic( )***, то есть локальна.

Внешняя переменная ***hocus*** неизвестна ни ***main( )***, ни ***magic( )***, но, обычно, известна любой другой функции в файле, которая не имеет своей собственной локальной переменной ***hocus***.

Эти примеры иллюстрируют *область действия внешних переменных*. Они существуют, пока работает вся программа, и, так как эти переменные доступны любой функции, то они не исчезнут, если какая-нибудь одна функция закончит свою работу.

Другим хорошим примером функции, требующей использования *статических локальных переменных*, является генератор последовательности чисел, создающий новое число, основываясь на старом. Это можно сделать, объявив *глобальную переменную*. Тем не менее, каждый раз, когда функция используется в программе, следует помнить об объявлении глобальной переменной и постоянно необходимо смотреть, не конфликтует ли переменная с другими, ранее объявленными, переменными. Использование глобальной переменной еще и приводит к тому, что функцию трудно поместить в библиотеку функций. Лучшим решением является объявление переменной, содержащей сгенерированное число как ***static***, как в вышеприведенном фрагменте (пример 7.13).

Пример 7.19.

```
int series( )
    {static int series_num=0;
     series_num = series_num+23;
     return(series_num);
    }
```

В данном примере переменная ***series_num*** *существует между вызовами* функций вместо того, чтобы каждый раз создаваться и уничтожаться как обычная *локальная* переменная. Это означает, что каждый вызов ***series( )*** может создать новый член серии, основываясь на последнем члене *без глобального объявления* переменной.

Отметим кое что необычное в функции ***series()***. Статическая переменная ***series_num*** инициализируется значением ***0***. Это означает, что при первом вызове функции series_num имеет значение ***0***. Хотя это приемлемо для некоторых приложений, большинство генераторов последовательности требуют какую-либо другую стартовую точку. Чтобы сделать это, требуется инициализировать ***series_num*** до первого вызова ***series( )***. Это может быть легко сделано, если ***series_num*** является *глобальной* переменной. Тем не менее, следует избегать использования ***series_num*** как глобальной переменной, лучше объявить ее как ***static***. Это приводит к описанному способу использования ***static***.

Область видимости объекта (переменной или функции) определяет набор функций или модулей, внутри которых допустимо использование имени этого объекта. Область видимости объекта начинается в точке объявления объект.

Время жизни объекта может быть глобальным и локальным.

Глобальными называют объекты, объявление которых дано вне функции. Они доступны (видимы) во всем файле, в котором они объявлены. В течение всего времени выполнения программы с глобальным объектом ассоциирована некоторая ячейка памяти.

Локальными называют объекты, объявление которых дано внутри блока или функции. Эти объекты доступны только внутри того блока, в котором они объявлены. Объектам с локальным временем жизни выделяется новая ячейка памяти каждый раз при осуществлении описания внутри блока. Когда выполнение блока завершается, память, выделенная под локальный объект, освобождается, и объект теряет своё значение (пример 7.20).

Область видимости локальной переменной ***k*** — функция ***autofunc()***. Каждый раз при входе в функцию с идентификатором ***k*** ассоциируется некоторая ячейка памяти, в которую помещается значение равное 1.

Пример 7.20.

```
#include <stdio.h>
int autofunc( )
   { int k = 1;    // локальный объект
     printf(" \n k = %d ", k);
     k = k + 1;
   }
int main()
   {  for (int i = 0; i <= 5; i++) // область видимости i – цикл
                    {autofunc();}
      return 0;
   }
```

Результат выполнения программы:

k = 1
k = 1
k = 1
k = 1
k = 1
k = 1

Пример 7.21. Та же программа, но с использованием глобального объекта.

```
#include <stdio.h>
int k = 1; // глобальный объект
int autofunc( )
   { printf(" \n k = %d ", k);
     k = k + 1;
   }
int main()
   {  for (int i = 0; i <= 5; i++)  // область видимости i – цикл
                    {autofunc();}
      return 0;
   }
```

Результат выполнения программы

k = 1
k = 2
k = 3
k = 4
k = 5
k = 6

С помощью глобальных переменных можно организовать обмен информацией между функциями. При этом вызываемая функция не будет принимать значения глобальных переменных в качестве формальных аргументов. Однако в этом случае существует опасность случайного изменения глобальных объектов другими функциями (пример 7.22).

Пример 7.22

```
#define _CRT_SECURE_NO_WARNINGS
/* _CRT_SECURE_NO_WARNINGS используется для совместимости с
                        классическими стандартными функциями*/
#include <stdio.h>
int x, y, z;   // глобальные переменные
int sum( )     {  z = x + y; }          //Описание функции sum( )
int main( )
   {  printf("x= ");
      scanf("%d", &x);
      printf("y= ");
      scanf("%d", &y);
      sum();
      printf("z= %d", z);
       return 0;
    }
```

Результат выполнения

x = 3
y = 4
z = 7

Чтобы изменить *диапазон значений* или *область действия объекта* или *видоизменение объектов* в языке Си применяется модификация объекта. Ключевые слова, которые применяются для модификации, называются модификаторами.

Пример 7.23.

Файл 1

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int x, y, z;
extern void func(void);
```

Файл 2

```
extern int x, y, z;
void func(void)
{  z = x + y;}
```

```
int main()
  {  printf("x= ");
     scanf("%d", &x);
     printf("y= ");
     scanf("%d", &y);
     func();
     printf("z= %d", z);
     return 0;
  }
```

Модификатор ***unsigned*** предназначен для того, чтобы объявлять беззнаковую целочисленную переменную, тем самым изменив диапазон представления этой переменной.

Модификатор ***extern*** предназначен для использования в данном программном модуле объекта, который объявлен в другом программном модуле (пример 7.17).

Модификатор ***static*** позволяет связать с идентификатором фиксированный адрес (ячейку памяти). Если объект расположен по некоторому *фиксированному адресу*, то он является *статическим*.

Объект, который располагается в *произвольном месте оперативной памяти*, называется *динамическим*. Если необходимо динамический объект сделать статическим, то используется модификатор static. Переменные, объявленные с использованием модификатора static сохраняют свои значения при входе и выходе из функции, однако не являются глобальными.

Пример 7.24.

```
#include <stdio.h>
void autofunc(void)
    {  static int k = 1;   // статический объект
       printf(" \n k = %d ", k);
       k = k + 1;
    }
int main()
    {  for (int i = 0; i <= 5; i++)
      autofunc();
      return 0;
    }
```

Переменная ***k*** в функции ***autofunc()*** зафиксирована в оперативной памяти. Инициализация ***k*** проводится только один раз — при первом вызове функции. При повторном обращении к функции ***autofunc( )*** инициализация переменной ***k*** не будет производиться. Значение переменной ***k*** и ее адрес сохраняются в оперативной памяти, однако эта переменная не будет доступна из других функций.

Результат выполнения программы.

k = 1
k = 2
k = 3
k = 4
k = 5
k = 6

Модификатор ***register*** предназначен для того, чтобы поместить переменную в один из регистров общего назначения центрального процессора при наличии свободного регистра. Благодаря этому повышается скорость работы с данными. Это необходимо для создания управляющих программ, где требуется высокая скорость обработки данных (пример 7.19).

Замечание. Важно отметить, что операция получения адреса **&** неприменима к регистровым переменным. Это отличает использование модификатора класса памяти ***register*** от остальных.

Пример 7.25.

```
#include <stdio.h>
int main()
  {  register int a = 0;
     for (int i = 1; i < 10; i++) {  a += i; }
     printf("%d", a);
     return 0;
  }
```

Вопросы для контроля

1. Как определяется объём памяти под основные структуры: «базовые» и более сложные (массивы, строки, структуры)?
2. Какие особенности имеет распределение с перекрытием, как оно ещё называется?
3. Какой размер памяти отводится разнообразным типам переменных?
4. Какие существуют способы хранения объектов (по типу распределения памяти), чем они характеризуются?
5. Опишите особенности локальной, глобальной, внешней переменной с точки зрения области видимости и взаимодействия функций в программе.

Глава 8. Методы управления доступом к данным в вычислительных системах с общей памятью.

Учебные цели:

- дать представление о ошибках доступа к памяти;
- дать представление о видах переменных и областей памяти;
- дать представление о приемах работы со статической и динамической памятью.

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Распределению подлежит вся оперативная память, не занятая операционной системой. Обычно ОС располагается в самых младших адресах, однако может занимать и самые старшие адреса.

Функциями ОС по управлению памятью являются:

- отслеживание свободной и занятой памяти,
- выделение памяти процессам и освобождение памяти при завершении процессов,
- вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место,
- а также настройка адресов программы на конкретную область физической памяти.

8.1. Выравнивание границ областей памяти.

Выравнивание данных в оперативной памяти компьютеров — способ размещения данных в памяти особым образом для ускорения доступа.

Центральные процессоры в качестве основной единицы при работе с памятью используют машинное слово, размер которого может быть различным. Однако размер слова всегда равен нескольким байтам (байт является наименьшей единицей, в которой отсчитываются адреса).

Как правило, машинное слово равно ***2k*** (***4***k) байтов, то есть состоит из одного, двух, четырёх, восьми байтов и так далее.

При сохранении какого-то объекта в памяти может случиться, что некое поле, состоящее из нескольких байтов, пересечёт «естественную границу» слов в памяти. Некоторые модели процессоров не могут обращаться к данным в памяти, нарушающим границы машинных слов. Некоторые могут обращаться к невыровненным данным дольше, нежели к данным, находящимся внутри целого «машинного слова» в памяти (рис.8.1).

На практике такое выравнивание означает, что адреса всех данных размером (превосходящие размер слова) должны делиться без остатка на ***2k***.

Рис. 8.1. Схема выравнивания

Память компьютера работает следующим образом. Память программы можно представить себе в виде большого массива байтов, а адрес — индекс в этом массиве (рис. 8.2 Способ 1).

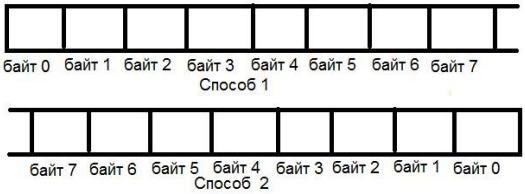


Рис.8.2. Способы адресации памяти

Иногда легче представлять адресацию в обратном порядке (рис. 8.2 Способ 2).

Байт, как минимально адресуемую единицу памяти, можно адресовать любым способом. Но обычно работают с более крупными единицами. Например, *машинное слово*: размер регистра процессора. Для 32-битной архитектуры это четырёхбайтовая структура.

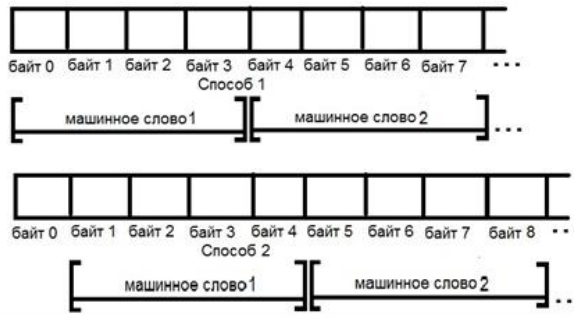


Рис.8.3. Способы размещения машинного слова

Теперь, байты прекрасно организовываются в четвёрки (рис. 8.3 Способ 1).

Заметим, что, в принципе, байты можно бы комбинировать в машинные слова и по-другому, то есть таким образом, что машинное слово могло бы начинаться по любому адресу (рис.8.3. Способ 2)!

Однако, большинство компьютеров устроено так, что машинные слова, скомбинированные в четвёрки способом 1 (рис. 8.3.), читаются быстро, а вот машинные слова второго типа (то есть, представленные на рисунке 8.3, способ 2, которые не получаются шагами по 4 от нуля) — медленно. (Хотя требования по выравниванию могут быть и сложнее, и не совпадать с кратностью машинного слова). Больше того, очень многие архитектуры вовсе не позволяют читать машинные слова, построенные по способу 2 рисунка 8.3, и для того, чтобы прочитать машинное слово 1 (рисунок 8.3. Способ 2): приходится читать машинное слово 1 по адресу 0 и машинное слово 2 по адресу 4, выбирать из них нужные байты, перетасовывать и собирать «вручную» в нужное машинное слово! Понятно, что это сложная и сравнительно дорогостоящая операция.

Поэтому компиляторы языка программирования Си (да и большинства других тоже) проводят такую оптимизацию: между полями структуры вставляются незначащие байты для того, чтобы у всех полей было хорошее выравнивание.

Рассмотрим, как организуется выделение памяти для переменных примера 8.1 для операционной системы с 32-разрядной картой памяти. Это означает, что выравнивание будет осуществляться не по четным адресам, то есть со сдвигом в 2 байта, а со сдвигом в 4 байта.

Пример 8.1. Выделение памяти для структуры, содержащей переменные разного типа

```
struct
  { char x;          // 1 байт
    int y;   // 4 байта
    char z;          // 1 байт
  };
```

Память выделяется так:

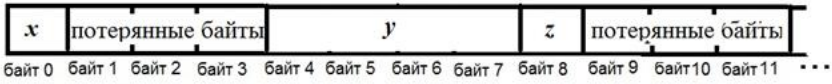


Рис. 8.4. Структура данных, где переменные выровнены по адресу, кратному 4

Если сама структура будет выровнена в памяти (об этом компилятор тоже заботится), то (рис. 8.4)

- переменная ***x*** ***(char*** 1 байт***)*** займет байт с относительным адресом 0,
- переменная ***y*** (***int*** 4 байта) нужно выделить последовательные 4 байта; поэтому байты с адресами 1-3 будут пропущены (потерянные байты.), и переменная ***y*** займет 4 байта, начиная с адреса 4,
- переменная ***z*** (***char*** 1 байт) займет байт с относительным адресом 8, так как последний байт, занятый переменной ***y***, имеет адрес 7, а следующий байт имеет адрес 8, кратный 4.

Таким образом, все переменные структуры будут выровнены, и доступ к ним будет быстрым (а на некоторых платформах доступ к памяти, вообще, только в этом случае возможен).

В этом случае структура примера 8.1 занимает больше памяти, чем если бы порядок переменных был таким:

Рис. 8.5. Более эффективное построение структуры примера 8.1.

Бóльшая структура означает бóльший расход памяти и бóльшие затраты на копирование, чтение этой структуры и тому подобное. Таким образом, переставив поля структуры в программе, можно сэкономить.

Заметим, что на самом деле универсальной оптимизации не существует. Например, для разных архитектур компьютера размеры машинных слов могут отличаться. Кроме того, и размеры сколько-нибудь сложных данных могут отличаться тоже, а, значит, вычисленный оптимальный порядок на одной системе может оказаться очень неоптимальным на другой системе.

Говоря об объеме памяти, занимаемой различными лексическими единицами и структурами данных языка, следует всегда учитывать один очень важный момент, а именно, *выравнивание границ* областей памяти, отводимых для различных лексических единиц.

Архитектура многих современных вычислительных систем предусматривает, что обработка данных выполняется более эффективно, если адрес, по которому выбираются данные, кратен определенному числу байт (как правило, это 2, 4, 8 или 16 байт). Современные компиляторы учитывают и особенности целевых вычислительных систем. При распределении данных они могут размещать области памяти под

лексические единицы наиболее оптимальным образом. Поскольку не всегда размер памяти, отводимой под лексическую единицу, кратен указанному числу байт, то в общем объеме памяти, отводимой под результирующую программу, могут появляться неиспользуемые области.

Например, если имеется описание переменных на языке Си:

static char c1, c2, c3;

то, зная, что под одну переменную типа ***char*** отводится 1 байт памяти, можно ожидать, что для описанных выше переменных потребуется всего 3 байта памяти.

Рис. 8.6.Выравнивание при кратности адресов для доступа к памяти в 4 байта

Однако если кратность адресов для доступа к памяти установлена в 4 байта, то под эти переменные будет отведено в целом 12 байт памяти, из которых 9 не будут использоваться (рис. 8.6).

Выравнивание границ областей памяти возможно, как по границам структур данных (и представляющих их лексем), так и внутри самих структур данных для составляющих их элементов. Например, если мы имеем описание массивов на языке Си:

char a1[3], a2[3];

то, при условии, что под одну переменную типа ***char*** отводится 1 байт памяти и кратность адресов установлена в 4 байта, в случае, когда выравнивание границ областей памяти выполняется только по границам самих областей, под описанные выше переменные потребуется 8 байт памяти (по 4 байта на каждый массив) (рис. 8.7).

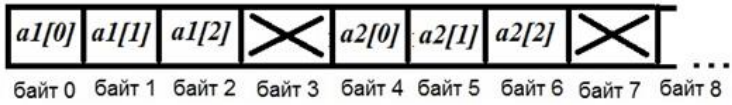


Рис. 8.7. Выравнивание массивов по границам областей

Если же выравнивание границ областей памяти выполняется и для составляющих элементов (в данном случае – элементов массивов), то под эти же переменные потребуется 24 байта памяти (по 4 байта на один элемент массива – и в целом 12 байтов на каждый массив) (рис. 8.8).

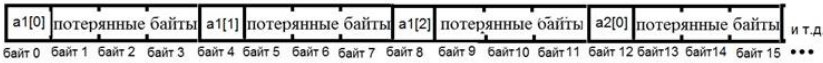


Рис. 8.8. Выравнивание элементов массивов по границам областей

Как правило, разработчику исходной программы не нужно знать, каким образом компилятор распределяет адреса под отводимые области памяти.

Чаще всего компилятор сам выбирает оптимальный метод, и, изменяя границы выделенных областей, он всегда корректно осуществляет доступ к ним. Вопрос об этом может встать, если с данными программы, написанной на одном языке, работают программы, написанные на другом языке программирования (чаще всего, на языке ассемблера), реже такие проблемы возникают при использовании двух различных компиляторов с одного и того же входного языка. Большинство компиляторов позволяют пользователю в этом случае самому указать, использовать или нет кратные адреса и какую границу кратности установить (если это вообще возможно с точки зрения архитектуры целевой вычислительной системы).

Если вернуться к рассмотренному выше примеру фрагмента программы на языке Си, то, если предположить, что кратность размещения данных в целевой вычислительной системе составляет 4 байта и выравнивание границ памяти выполняется только по границам структур, можно рассчитать новые значения объема памяти, требуемой для размещения описанных в этом фрагменте описаний переменных.

Пример 8.2. Вычисление объёма памяти фрагмента описаний

```
int arr1 [10][20] ;     //массив из 10 строк и 20 столбцов целых чисел
struct rec1         //структура rec1 из полей
  { int v1;          //целой переменной
    int a1[10][20];  //массива из 200 элементов
  }rr1;
struct rec2      // структура rec2 из полей
  {char v1;       //символьной пременной
    int v2;       //целой переменной
    union rr      //объединение
      { char v3;
        short int v4;
         int v5;
      };
struct rec2 arr2[100];   //массив структур типа rec2
 char vv1, vv2, vv3, vv4;
```

Объём памяти для двумерного массива ***arr1[10][20]*** будет составлять $V_1 = 10*20*4 = 800$ байт.

Объём памяти для структуры ***rec1*** составит $V_2 = 2+800 = 802$ байта; но с учетом кратности (4 байта), для его размещения потребуется $V_2' = 4+800 = 804$ байта.

Объём памяти для структуры ***rec2***, содержащей объединение ***rr*** (для ***rr*** потребуется ***Max(1, 2, 4)*** байтов) составит $V_3 = 1+4+Max(1,2,4) = 9$ байт; а с учетом кратности — $V_3' = 12$ байтов.

Объём памяти для массива структур ***arr2***, в том числе с учетом кратности составит $V_4' = 100*12 = 1200$ байтов.

Для размещения переменных ***vv1, vv2, vv3, vv4*** с учетом кратности потребуется ***4*4 = 16*** байтов памяти.

Таким образом, для размещения в памяти всего блока описанний фрагмента данных (***arr1[10][20] + rr1 + arr2[100] + vv1, vv2, vv3, vv4+***) потребуется выделить ***V' = 800 + 804 + 1200 +16 = 2820*** байтов памяти.

При тех же условиях, но с выравниванием границ памяти по элементам структур, необходимые объемы памяти в данном случае не изменятся. Размер типа ***arr1*** будет составлять $V_1 = 10*20*4 = 800$ байтов. Размер типа ***rec1*** с учетом кратности составит $V_2'' = 4+800 = 804$ байта. Размер типа ***rec2*** с учетом кратности составит $V_3' = 4+4+Max(4,4,4) = 12$ байтов, а размер типа ***arr2*** с учетом кратности составит $V_4' = 100*12 = 1200$ байтов.

Видно, что объем требуемой памяти в обоих случаях увеличился примерно на 12.5%, но за счет этого увеличивается эффективность обращения к данным и, следовательно, скорость выполнения результирующей программы.

8.2. Приемы работы со статической и динамической памятью.

В традиционных языках программирования, таких как Си, Фортран, Паскаль, существуют три вида памяти:

— статическая,
— стековая и
— динамическая.

С физической точки зрения никаких отличий в видах памяти нет: оперативная память представляет собой массив байтов, в которой каждый байт имеет адрес; адресация оперативной памяти начинается с нуля. Поэтому, говоря о видах памяти, имеют в виду способы (приёмы) организации работы с ней, включая выделение и освобождение памяти, а также методы доступа.

Выделение памяти происходит на этапе загрузки программы.

Статическая память выделяется еще до начала собственно выполнения программы, на стадии компиляции и сборки. Статические переменные имеют фиксированный адрес, известный до запуска программы на выполнение, и этот адрес не изменяется в процессе работы программы. Статические переменные создаются и инициализируются до входа в основную функцию ***main***, с которой начинается выполнение программы.

Данные, находящиеся в статически выделенной памяти, доступны при повторных вызовах программного модуля, в котором определена соответствующая переменная. Обычно статически размещаются в памяти константы, значение которых известно уже во время компиляции, и размещаются в таблицах, например, в таблице идентификаторов и символов.

Для статических переменных время жизни совпадает со временем работы всей программы. Отметим, что в некоторых языках все локальные переменные являются стек-динамическими, а все глобальные — статическими.

Статические глобальные переменные объявляются на верхнем уровне программы; область их видимости — вся программа. Статические локальные переменные объявляются внутри функции; область видимости статических локальных переменных — функция.

Автоматическая переменная (переменная со статической областью видимости) автоматически размещается в памяти, когда поток выполнения (выполняется вызов функции) входит в ее область видимости, и удаляется из памяти, когда поток выполнения (окончательно) покидает ее область видимости (осуществляется выход из функции). Размещаются автоматические переменные в области активации функции, в которой они объявлены, обеспечивают поддержку рекурсии, могут быть размещены в регистрах процессора (быстрой памяти).

Любая переменная, объявленная внутри некоторого блока кода, по умолчанию является автоматической (ключевое слово ***auto***), ее значение не определено до явного присваивания.

Локальные массивы создаются в стеке и имеют автоматическую продолжительность хранения и обязательно имеют фиксированный размер: при такой организации управления памятью не нужно вручную управлять выделением и освобождением памяти, статические массивы уничтожаются, когда заканчивается функция.

Динамическое выделение памяти осуществляется, например, для массивов, созданных с помощью оператора ***new[]***. Такие массивы имеют динамическую продолжительность хранения и хранятся в *куче* (технически "свободный магазин", глава 11). Они могут иметь любой размер, но для них нужно выделять и освобождать память самостоятельно, так как они не являются частью фрейма стека:

Рассмотрим пример обращения к элементам *статического массива* ***a*** на уровне процессора. Согласно выше сказанному, адрес начала массива известен еще при компиляции. Чтобы получить адрес элемента массива ***a[i]***, к адресу начала массива ***a*** прибавляется индекс ***i***, умноженный на размер одного элемента в байтах. Затем по вычисленному адресу идет обращение в память.

Доступ к элементам *динамического массива* получить значительно сложнее. На этапе компиляции адрес начала массива неизвестен, после выделения памяти этот адрес хранится в специальной переменной — указателе. Чтобы получить адрес элемента массива ***a[i]***, нужно сначала обратиться к памяти (взять значение указателя) и получить адрес начала массива. Потом к нему прибавить индекс, умноженный на размер элемента, и по вычисленному адресу элемента массива снова обратиться к памяти. Таким образом, имеем два обращения к памяти в отличие от одного у статических массивов.

Этот процесс доступа к динамическому массиву в компиляторе можно оптимизировать. Адрес начала массива можно запомнить в регистре до обращений к элементам массива. Однако, в таком случае, в программе с динамическими массивами занимается на один регистр больше.

Но не всегда такая оптимизация приводит к эффективности.

Рассмотрим пример перебора всех элементов массива в цикле. Если цикл оперирует с маленьким набором данных, адреса массивов и используемые переменные можно хранить в регистрах. С расширением набора элементов массива количество свободных регистров уменьшается, и тогда на каком-то этапе в теле цикла появится дополнительное обращение к памяти. Это произойдет тогда, когда все доступные регистры будут уже использованы. Именно в этих условиях происходит переход от статического массива к динамическому, и это может привести к существенному замедлению программы.

Отметим также, что без необходимости лучше не использовать глобальные данные, а использовать локальные переменные. Это оправдано не только идеологически: компилятор может разместить локальные переменные не в памяти, а в регистрах.

Умение программировать на низкоуровневых языках программирования, например, на ассемблере позволяет писать более оптимизированный и оптимизируемый код и на языках высокого уровня. Например, лучше использовать массив структур, а не несколько разных массивов. Такая методика управления памятью оправдана не только описанной логикой освобождения регистров и уменьшения количества обращений к памяти, но и логикой работы процессорного кеша[17]. Кроме того, при таком управлении памятью данные проще сохранять в файл.

Вопросы для контроля

1. Определите понятие выравнивания данных. Приведите примеры выравнивания «базовых переменных».
2. Какие виды памяти различают в традиционных языках программирования? Каковы их существенные особенности?
3. Какие виды статических переменных вы знаете? Где и когда они размещаются в памяти и когда её покидают?
4. Как вычисляется объем памяти требуемой для различных объектов (переменных различных типов, структур, массивов, объединений, перечислений) с учетом выравнивания?
5. Как осуществляется доступ к элементам динамического массива?

[17] Кэш процессора, иначе кэш (сверхоперативная память) используется процессором компьютера для уменьшения среднего времени доступа к оперативной памяти. Является одним из верхних уровней иерархии памяти. Кэш использует небольшую, очень быструю память, которая хранит копии часто используемых данных из основной памяти. Если большая часть запросов в память будет обрабатываться кэшем, средняя задержка обращения к памяти будет приближаться к задержкам работы кэша.

Глава 9. Способы внутреннего представления программ

Учебные цели:

- — дать общее представление о видах внутреннего представления программ;
- — расширить представление о синтаксических деревьях;
- — детализировать представление о преобразованиях дерева разбора в дерево операций.

Результатом работы синтаксического анализатора на основе КС-грамматики входного языка является последовательность правил грамматики, примененных для построения входной цепочки (см. главы 3, 4). По найденной последовательности, зная тип распознавателя, можно построить цепочку вывода или дерево вывода. В этом случае дерево вывода выступает в качестве дерева синтаксического разбора и представляет собой результат работы синтаксического анализатора в компиляторе.

Однако, ни цепочка вывода, ни дерево синтаксического разбора не являются целью работы компилятора. Для полного представления о структуре разобранной синтаксической конструкции входного языка, в принципе, достаточно знать последовательность номеров правил грамматики, примененных для ее построения. Форма представления этой информации может быть различной в зависимости как от реализации самого компилятора, так и от фазы компиляции. Эта форма называется *внутренним представлением программы* (иногда используются также термины *промежуточное представление* или *промежуточная программа*).

9.1. Виды внутреннего представления программ.

Возможны различные формы внутреннего представления синтаксических конструкций исходной программы в компиляторе. На этапе синтаксического разбора часто используется форма, именуемая деревом вывода (методы его построения рассматривались выше (см. главу 6). Но формы представления, используемые на этапах синтаксического анализа, оказываются неудобными в работе при генерации и оптимизации объектного кода. Поэтому перед оптимизацией и непосредственно перед генерацией объектного кода внутреннее представление программы может преобразовываться в одну из соответствующих форм записи.

Чтобы яснее представлять, как формируется то или иное внутреннее представление, напомним, какая информация используется компилятором.

Прежде всего, это таблица символов, где каждая запись связана с атрибутами, которые поддерживают компилятор на разных этапах. Элементами записей таблицы символов являются:

- — имена и константы переменных,
- — имена процедур и функций,

— литеральные константы и строки,
— временные объекты компилятора,
— метки на исходных языках.

Для элементов таблицы символов выбирается следующая информация, используемая в дальнейшем компилятором:

— тип данных и имя,
— описание функций,
— адрес смещения для получения адреса хранения значения,
— для структуры — указатель на структуру таблицы,
— для любого типа параметров (передача параметра по значению или по ссылке)
    - количество и тип аргументов, переданных функции,
    - базовый адрес.

Таблица символов строится и модифицируется на разных фазах анализа компилятора, соответственно, определены операции с таблицей символов.

Основные операции, определенные для атрибутов и элементов таблицы символов, включают операции, приведенные в таблице 9.1

Таблица 9.1. Основные операции, выполняемые над таблицей символов

Операция	Функция
allocate	выделение новой пустой области памяти в таблице символов
free	удаление всех записей и освобождение области памяти в таблице символов
lookup	поиск имени и возврат указателя на его запись
insert	вставка имени в таблицу символов и возврат указателя на его запись
set_attribute	связать атрибут с данной записью
get_attribute	получить атрибут с заданной записью

Все внутренние представления программы обычно содержат в себе два принципиально различных элемента – *операторы* и *операнды*. Различия между формами внутреннего представления заключаются лишь в том, как операторы и операнды соединяются между собой. Кроме этого, операторы и операнды должны отличаться друг от друга, если они встречаются в любом порядке. За различение операндов и операторов, как уже было сказано выше, отвечает разработчик компилятора, который руководствуется семантикой входного языка.

Известны следующие формы внутреннего представления программ:

1. структуры связных списков, представляющие синтаксические деревья;
2. многоадресный код с явно именуемым результатом (тетрады);
3. многоадресный код с неявно именуемым результатом (триады);
4. обратная (постфиксная) польская запись операций;
5. ассемблерный код или машинные команды.

В каждом конкретном компиляторе может использоваться одна из этих форм, выбранная разработчиками. Но чаще всего компилятор не

ограничивается использованием только одной формы для внутреннего представления программы. На различных фазах компиляции могут использоваться различные формы, которые по мере выполнения проходов компилятора преобразуются одна в другую.

Некоторые компиляторы, незначительно оптимизирующие результирующий код, генерируют объектный код по мере разбора исходной программы. В этом случае применяется схема СУ-компиляции (синтаксически управляемой компиляции), когда фазы синтаксического разбора, семантического анализа, подготовки и генерации объектного кода совмещены в одном проходе компилятора. Тогда внутреннее представление программы существует только условно в виде последовательности шагов алгоритма разбора.

Для каждой из указанных выше форм внутреннего представления программы используются соответствующие структуры данных для реализации таблицы символов. Перечислим эти структуры и особенности способа реализации их в процессе преобразования форм внутреннего представления программы.

1. *Список* (представляемый массивом):
   - в этом методе массив используется для хранения имен и связанной информации,
   - специальный указатель «***доступно***» сохраняется в конце всех сохраненных записей, и новые имена добавляются в порядке поступления,
   - поиск имени начинается каждый раз с начала списка до требуемого указателя и, если он не найден, выдаётся ошибка «***использование необъявленного имени***»**,**
   - при вставке нового имени необходимо убедиться, что это имя еще не присутствует в таблице символов; в противном случае возникает ошибка, например, «***Множественное определение имени***»**,**
   - вставка имени для небольших таблиц символов быстрая, имеет сложность ***O(1)***; поиск для больших таблиц оказывается медленным, в среднем ***O (n),***
   - преимущество метода, использующего массив для организации таблицы символов, состоит в том, что таблица символов занимает минимальное количество места.
2. *Связанный список*:
   - в этом методе реализация таблицы символов используется связанный список (динамическое распределение памяти), то есть каждая запись (элемент связанного списка) содержит поле ссылки,
   - поиск имен осуществляется в порядке, указанном ссылкой в поле ссылки,

— указатель на первый элемент связанного списка поддерживается неизменным для возможности не терять адрес начала списка,
— вставка имени для небольших таблиц символов быстрая, имеет сложность O(1); поиск для больших таблиц оказывается медленным, в среднем O (n).

3. *Хэш таблица:* (способ, чаще всего используемый для построения таблицы символов для первой из выше приведённых форм внутреннего представления программы):
   — в схеме хеширования поддерживаются две таблицы: хеш-таблица и таблица символов, и это наиболее часто используемый метод для реализации таблиц символов.
   — хеш-таблица — это массив с индексным диапазоном от 0 до размера таблицы; все записи являются указателями, указывающими на имена таблицы символов,
   — для поиска имени используется хеш-функция, которая приводит к любому целому числу от 0 до размера таблицы;
   — вставка и поиск выполняются очень быстро, со сложностью ***O(1),***
   — преимущество метода состоит в том, что возможен быстрый поиск, а недостатком является сложность реализации хеширования.
4. *Двоичное дерево поиска:*
   — этот подход к реализации таблицы символов состоит в том, что используется двоичное дерево поиска, то есть в каждой записи таблицы символов присутствуют два поля ссылки (левый и правый дочерние элементы).
   — все имена создаются как дочерние элементы корневого узла, которые всегда соответствуют правилам построения бинарного дерева поиска.
   — вставка и поиск выполняются в среднем со сложностью $\boldsymbol{O(\log_2 n)}$.

Внутренняя форма — это промежуточная форма представления исходной программы, пригодная

1) для простой машинной обработки (операторы располагаются в том же порядке, в котором они и будут исполняться);
2) для интерпретации.

Рассмотрим, как представимы во внутренней форме некоторые наиболее употребительные операторы. Начнем с польской формы.

Постфиксная польская форма записи. При построении обратной польской записи первоначальный алгоритм модифицируется так, чтобы операнды выбирались не из входного потока непосредственно, а из специально отведенного для этого *стека*. Договоримся, что в регистре ***R*** (определяемого функцией ***R()***) будет храниться текущий считываемый символ (верхний символ стека).

Опишем в общих чертах алгоритм построения обратной польской формы (в главе 10 рассмотрим его подробнее).

1. Если символ, содержащийся в ***R,*** является идентификатором или константой, то его значение заносится в стек и осуществляется считывание следующего символа
   (правило "***<операнд>::=идентификатор***").
2. Если символ, содержащийся в ***R,*** — бинарный оператор, то он применяется к двум верхним операндам в стеке, и затем операнды меняются на полученный результат, реализуя правило:
   "***<операнд>::=<операнд><операнд><оператор>***").
3. Если символ, содержащийся ***R,*** — унарный оператор, то он применяется к верхнему символу в стеке, который затем заменяется на полученный результат, реализуя правило:
   (правило "***<операнд>::=<операнд><оператор>***").

Упрощенно алгоритм вычисления польской формы (ПФ) выглядит как в примере 9.1.

Пример 9.1.

```
R = getchar;
while (R != #)
 {switch( R)
   { case 'идентификатор': push(R);
     case 'бин.оператор': pop(B); pop(A); Tmp = R(A,B); push(Tmp);
     case 'унарн.оператор': pop(A); Tmp = R(A); push(Tmp);
   }
  R = getchar;
 }
```

Замечание. Следует помнить, что существуют операторы, которые не заносят в стек результаты своей работы. Таким оператором является, к примеру, оператор присваивания языка Паскаль. Однако в языке Си (Си++) оператор присваивания *возвращает* значение.

Более подробно построение обратной польской записи рассмотрим в главе 10.

Синтаксические деревья. Это структура, представляющая собой результат работы синтаксического анализатора. Она отражает синтаксис конструкций входного языка и явно содержит в себе полную взаимосвязь операций. Очевидно также, что синтаксические деревья — это машинно-независимая форма внутреннего представления программы.

Недостаток синтаксических деревьев заключается в том, что они представляют собой сложные связные структуры, а потому не могут быть тривиальным образом преобразованы в линейную последовательность команд результирующей программы. Тем не менее, они удобны при работе с внутренним представлением программы на тех этапах, когда нет необходимости непосредственно обращаться к командам результирующей программы.

Синтаксические деревья могут быть преобразованы в другие формы внутреннего представления программы, представляющие собой линейные списки, с учетом семантики входного языка. Эти преобразования выполняются на основе принципов СУ-компиляции (синтаксически управляемой компиляции). В частности, схема преобразования синтаксического дерева в дерево операций приведена в главе 6 и еще будет разбираться в этой главе пункте 9.3.

Многоадресный код с явно именуемым результатом (тетрады). Тетрады представляют собой запись операций в форме из четырех составляющих: операция, два операнда и результат операции. Например, тетрады могут выглядеть так:

<операция> (<операнд1>, <операнд2>, <результат>).

Тетрады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме тетрад, они вычисляются одна за другой последовательно. Каждая тетрада в последовательности вычисляется так: операция, заданная тетрадой, выполняется над операндами и результат ее выполнения помещается в переменную, заданную результатом тетрады. Если какой-то из операндов (или оба операнда) в тетраде отсутствует (например, если тетрада представляет собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи и ее реализации).

Результат вычисления тетрады никогда не может быть опущен, потому что тогда тетрада полностью теряет смысл. Порядок вычисления тетрад может быть изменен, но только если допустить наличие тетрад, целенаправленно изменяющих этот порядок (то есть при наличии, тетрады, вызывающей переход на несколько шагов вперед или назад при каком-то условии).

Тетрады представляют собой линейную последовательность, а потому для них несложно написать тривиальный алгоритм, который будет преобразовывать последовательность тетрад в последовательность команд результирующей программы (либо последовательность команд ассемблера). В этом их преимущество перед синтаксическими деревьями. А в отличие от команд ассемблера тетрады не зависят от архитектуры вычислительной системы, на которую ориентирована результирующая программа. Поэтому они представляют собой машинно-независимую форму внутреннего представления программы.

Тетрады требуют больше памяти для своего представления, чем триады, они также не отражают явно взаимосвязь операций между собой. Кроме того, есть сложности с преобразованием тетрад в машинный код, так как они плохо отображаются в команды ассемблера и машинные коды, поскольку в наборах команд большинства современных компьютеров редко встречаются операции с тремя операндами.

Например, выражение ***A=B*C+D-B*10***, записанное в виде тетрад, будет иметь вид:

1. * (B, C, T1)
2. + (Tl, D, T2)
3. * (B, 10, T3)
4. – (T2, T3, T4)
5. := (T4, 0, A)

Здесь все операции обозначены соответствующими знаками (при этом присваивание также является операцией). Идентификаторы ***T1, ..., T4*** обозначают временные переменные, используемые для хранения результатов вычисления тетрад. Следует обратить внимание, что в последней тетраде (присваивание), которая требует только одного операнда, в качестве второго операнда выступает незначащий операнд ***«0»***.

Более подробно о построении внутреннего представления программ в виде многоадресного кода с явно именуемым результатом (тетрадами) рассмотрим в главе 13.

Многоадресный код с неявно именуемым результатом (триады). Триады представляют собой запись операций в форме из трех составляющих: операция и два операнда. Например, триады могут иметь вид:

<операция>(<операнд1>, <операнд2>).

Особенностью триад является то, что один или оба операнда могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады. Поэтому триады при записи последовательно нумеруют для удобства указания ссылок одних триад на другие (в реализации компилятора в качестве ссылок можно использовать не номера триад, а непосредственно ссылки в виде указателей — тогда при изменении нумерации и порядка следования триад менять ссылки не требуется).

Триады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме триад, они вычисляются одна за другой последовательно. Каждая триада в последовательности вычисляется так: операция, заданная триадой, выполняется над операндами, а если в качестве одного из операндов (или обоих операндов) выступает ссылка на другую триаду, то берется результат вычисления этой другой триады.

Результат вычисления триады нужно сохранять во временной памяти, так как он может быть затребован последующими триадами. Если какой-то из операндов в триаде отсутствует (например, если триада представляет собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи и ее реализации). Порядок вычисления триад, как и для тетрад, может быть

изменен, но только если допустить наличие триад, целенаправленно изменяющих этот порядок (например, триады, вызывающие переход на несколько шагов вперед или назад при каком-то условии).

Триады представляют собой линейную последовательность, а потому для них несложно написать тривиальный алгоритм, который будет преобразовывать последовательность триад в последовательность команд результирующей программы (либо последовательность команд ассемблера). В этом их преимущество перед синтаксическими деревьями. Однако здесь требуется также и алгоритм, отвечающий за распределение памяти, необходимой для хранения промежуточных результатов вычисления, так как временные переменные для этой цели не используются. В этом отличие триад от тетрад.

Так же, как и тетрады, триады не зависят от архитектуры вычислительной системы, на которую ориентирована результирующая программа. Поэтому они представляют собой машинно-независимую форму внутреннего представления программы.

Триады требуют меньше памяти для своего представления, чем тетрады, они также явно отражают взаимосвязь операций между собой, что делает их применение удобным. Необходимость иметь алгоритм, отвечающий за распределение памяти для хранения промежуточных результатов, не является недостатком, так как удобно распределять результаты не только по доступным ячейкам временной памяти, но и по имеющимся регистрам процессора. Это дает определенные преимущества. Триады ближе к двухадресным машинным командам, чем тетрады, а именно эти команды более всего распространены в наборах команд большинства современных компьютеров.

Например, выражение ***A = B * C + D – B * 10***, записанное в виде триад, будет иметь вид:

1. * (B, C)
2. + (^1, D)
3. * (B, 10)
4. – (^2, ^3)
5. := (A, ^4)

Здесь операции обозначены соответствующим знаком (при этом присвоение также является операцией), а знак «^» означает ссылку операнда одной триады на результат другой.

Более подробно о построении внутреннего представления программ в виде многоадресного кода с неявно именуемым результатом (триадами) рассмотрим в главе 13.

*Ассемблерный код и машинные команды***.** Машинные команды удобны тем, что при их использовании внутреннее представление программы

полностью соответствует объектному коду и не требуются сложные преобразования. Команды ассемблера изначально создавались в качестве мнемонической записи машинных команд, то есть представляют собой форму записи машинных команд, потому в качестве формы внутреннего представления программы ассемблерные команды практически ничем не отличаются от машинных команд.

Однако использование команд ассемблера или машинных команд для внутреннего представления программы требует дополнительных структур для отображения взаимосвязи операций. В этом случае внутреннее представление программы получается зависимым от архитектуры вычислительной системы, на которую ориентирован результирующий код. Значит, при ориентации компилятора на другой результирующий код потребуется перестраивать как само внутреннее представление программы, так и методы его обработки. А вот при использовании триад или тетрад перестройки структуры не требуется.

Машинные команды представляют низкоуровневый язык, на котором должна быть записана результирующая программа. Значит, компилятор должен уметь работать с командами низкоуровнего языка (машинными командами). Более того, добиться наиболее эффективной результирующей программы можно только обрабатывая машинные команды, или их представление в форме команд ассемблера). Поэтому любой компилятор работает с представлением результирующей программы в форме машинных команд, однако их обработка происходит, как правило, на завершающих этапах фазы генерации кода. Более подробно о преобразовании в ассемблерный (машинный0 код) разберём в главе 13.

9.2. Синтаксические деревья.

Для изображения синтаксической структуры, как мы знаем, используется *граф*. Графом, в математике, называется произвольное множество объектов с заданными на нем отношениями. Элементы этого множества называются узлами или вершинами графа. Если узлами графа являются синтаксические единицы, а отношения являются синтаксическими, то граф называется синтаксическим. При изображении графа узлы рисуются в виде небольших окружностей, а отношения — в виде стрелок (дуг), направленных от первого члена отношения ко второму.

В информатике рассматривают *абстрактное синтаксическое дерево* (АСД) — *Abstract Syntax Tree, AST* — конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены операторам языка программирования, а листья — соответствующим операндам. Таким образом, листья являются пустыми операторами и представляют только переменные и константы.

Синтаксические деревья используются в парсерах[18] для промежуточного представления программы между деревом разбора (конкретным синтаксическим деревом) и структурой данных, которая затем используется в качестве внутреннего представления в компиляторе или интерпретаторе компьютерной программы для оптимизации и генерации кода. Возможные варианты подобных структур описываются абстрактным синтаксисом.

Абстрактное синтаксическое дерево отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы. Классическим примером такого отсутствия являются группирующие скобки, так как в АСД группировка операндов явно задаётся структурой дерева.

Для языка, который описывается контекстно-свободной грамматикой, какими являются почти все языки программирования, создание абстрактного дерева в синтаксическом анализаторе является тривиальной задачей (смотрите главы 2, 4, 6). Большинство правил в грамматике создают новую вершину, а символы в правиле становятся рёбрами. Правила, которые ничего не привносят в АСД (например, группирующие правила), просто заменяются в вершине одним из своих символов (пример разбора в главе 6 пункт 6.3). Кроме того, анализатор может создать полное дерево разбора и затем пройти по нему, удаляя узлы и рёбра, которые не используются в абстрактном синтаксисе, для получения АСД.

Результатом синтаксического разбора, как мы выяснили ранее, является *дерево вывода*. Оно содержит массу избыточной информации, которая для дальнейшей работы компилятора не требуется. Эта информация включает в себя все нетерминальные символы, содержащиеся в узлах дерева, — после того как дерево построено, они не несут никакой смысловой нагрузки и не представляют для дальнейшей работы интереса.

В синтаксическом дереве внутренние узлы (вершины) соответствуют операциям, а листья представляют собой операнды. Как правило, листья синтаксического дерева связаны с записями в таблице идентификаторов. Структура синтаксического дерева отражает синтаксис языка программирования, на котором написана исходная программа.

Синтаксические деревья могут быть построены компилятором для любой части входной программы. Не всегда синтаксическому дереву должен соответствовать фрагмент кода результирующей программы — например, возможно построение синтаксических деревьев для декларативной

[18] *Парсер* (от англ. *parser*) или *граббер* (от англ. *grabber*) — программное обеспечение, предназначенное для анализа и разбора исходных данных, с целью их обработки и дальнейшей использования в требуемом виде.

Процесс *парсинга* (англ. *parsing*) или *скраппинга* (англ. *skrapping*) обычно подразумевает выявление уникальных *признаков* или *закономерностей* употребления интересующего нас фрагмента данных, с целью построения соответствующих *правил* их «выделения» и «захвата» в исходных данных.

(описательного блока) части языка. В этом случае операции, имеющиеся в дереве, не требуют порождения объектного кода, но несут информацию о действиях, которые должен выполнить сам компилятор над соответствующими элементами. В том случае, когда синтаксическому дереву соответствует некоторая последовательность операций, влекущая порождение фрагмента объектного кода, говорят о *дереве операций*.

Дерево операций можно непосредственно построить из дерева вывода, порожденного синтаксическим анализатором. Для этого достаточно исключить из дерева вывода цепочки нетерминальных символов, а также узлы, не несущие семантической (смысловой) нагрузки при генерации кода. Примером таких узлов могут служить различные скобки, которые меняют порядок выполнения операций и операторов, но после построения дерева никакой смысловой нагрузки не несут, так как им не соответствует никакой объектный код.

Алгоритм преобразования дерева семантического разбора в дерево операций подробно разобран в главе 6 пункт 6.3.

Вне зависимости от того, какой именно язык программирования используется, исходный текст, всегда будет проходить этап парсинга, то есть превращения исходного текста в структуру данных, которая называется абстрактным синтаксическим деревом.

Абстрактные синтаксические деревья не только дают структурированное представление исходного кода, они, кроме того, играют важнейшую роль в семантическом анализе, в ходе которого компилятор проверяет правильность программных конструкций и корректность использования их элементов. После формирования AST и выполнения проверок эта структура используется для генерирования байт-кода или машинного кода.

9.3. Преобразование дерева разбора в дерево операций.

Синтаксическое дерево (дерево операций) — это структура, представляющая собой результат работы синтаксического анализатора. Она отражает синтаксис конструкций входного языка и явно содержит в себе полную взаимосвязь операций.

В синтаксическом дереве внутренние узлы (вершины) соответствуют операциям, а листья представляют собой операнды. Как правило, листья синтаксического дерева связаны с записями в таблице идентификаторов. Структура синтаксического дерева отражает синтаксис языка программирования, на котором написана исходная программа.

В том случае, когда синтаксическому дереву соответствует некоторая последовательность операций, влекущая порождение фрагмента объектного кода, говорят о дереве операций.

Дерево операций можно непосредственно построить из дерева вывода, порожденного синтаксическим анализатором. Подробный пример преобразования дерева вывода в дерево операций приведен в главе 6 пункт 6.3. Как было показано, для этого достаточно исключить из дерева вывода цепочки нетерминальных символов, а также узлы, не несущие семантической (смысловой) нагрузки при генерации кода (алгоритм преобразования дерева вывода в дерево операций приведен там же, в главе 6 пункт 6.3). Примером таких узлов могут служить различные скобки, которые меняют порядок выполнения операций и операторов, но после построения дерева никакой смысловой нагрузки не несут, так как им не соответствует никакой объектный код.

Непосредственно из грамматики, описывающей синтаксис входного языка, невозможно определить, какой узел в дереве является операцией, а какой — операндом. Аналогичным образом невозможно определить, каким операциям должен соответствовать объектный код в результирующей программе, а каким операциям — не должен. Эти соответствия определяется только, исходя из семантики («смысла»») языка входной программы. Поэтому только разработчик компилятора может четко определить, как при построении дерева операций должны различаться операнды и сами операции, а также то, какие операции являются семантически незначащими для порождения объектного кода.

Описанный в пункте 6.3 алгоритм преобразования дерева вывода в дерево операций всегда работает с узлом дерева, который считается текущим и стремится исключить из дерева, все узлы, помеченные нетерминальными символами. То, какие из символов считать семантически незначащими, а какие считать, знаками операций, решает разработчик компилятора. Если семантика языка задана корректно, то в результате работы алгоритма из дерева будут исключены все нетерминальные символы.

В результате применения алгоритма преобразования деревьев синтаксического разбора, в дерево операций, получаем дерево операции. Причем, несмотря на то, что исходные синтаксические деревья имели различную структуру, зависящую от используемой грамматики, результирующее дерево операций всегда имеет одну и ту же структуру, зависящую только от семантики входного языка.

Дерево операций является формой внутреннего представления программы, которой удобно пользоваться на этапах синтаксического разбора, семантического анализа и подготовки к генерации кода, когда еще нет необходимости работать непосредственно с кодами команд результирующей программы.

Преимущества внутреннего представления программы в виде дерева операций, состоят в том, что:

— четко отражает связь всех операций между собой, поэтому его удобно использовать для преобразований, связанных с перестановкой и переупорядочиванием операций без изменений конечного результата;
— синтаксические деревья — это машинно-независимая форма внутреннего представления программы.

Недостаток собственно синтаксических деревьев заключается в том, что они представляют собой сложные связанные структуры, а поэтому не могут быть тривиальным образом преобразованы в линейную последовательность команд результирующей программы. Однако, они удобны при работе с внутренним представлением программы на тех этапах, когда нет необходимости непосредственно обращаться к командам результирующей программы.

Синтаксические деревья могут быть преобразованы в другие формы внутреннего представления программы, представляющие собой линейные списки, с учетом семантики входного языка. Эти преобразования выполняются на основе принципов синтаксически управляемой компиляции (СУ-компиляции).

Вопросы для контроля

1. Дайте определение понятия внутреннее представления программы. Приведите примеры видов внутреннего представления программы.

2. Какие элементы таблиц компилятора используются в процессе формирования внутреннего представления программы?

3. Перечислите наиболее известные формы внутреннего представления программ, дайте краткую характеристику структур, используемых для каждой из них.

4. Каковы особенности формы внутреннего представления программ в виде ассемблерного или машинного кода

5. Назовите преимущества внутреннего представления программы в виде дерева операций.

Глава 10. Стековая организация памяти

Учебные цели:

- *дать представление о стеке;*
- *дать представление об обратной польской записи арифметических выражений.*

Современные разработчики программных продуктов используют всё более продвинутые языки программирования, которые позволяют писать меньше кода и получать отличные результаты, но за это приходится платить. Сегодня разработчики всё реже работают с низкоуровневыми языками программирования, поэтому нормальным явлением становится то, что многие из начинающих программистов недостаточно глубоко понимают структуру и функциональность таких понятий, как:

- стек и куча,
- процесс компиляции, его детальная картина формирования внутренних структур и функционирования,
- разница между статической и динамической типизацией,
- и многие другие важные для разработки программных продуктов понятия.

10.1. Применение стековой организации памяти

Прежде всего рассмотрим такие понятия как *стек* и *куча*. И стек, и куча относятся к различным к процессам размещения в оперативной памяти, где происходит управление памятью, и стратегии этого управления кардинально отличаются.

Стек — это область оперативной памяти, которая создаётся для каждого потока[19]. Он работает в режиме ***LIFO*** (*Last In, First Out*: первый пришел, последний ушел), то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека. Каждый раз, когда функция объявляет новую переменную, она добавляется в стек, а когда эта переменная пропадает из области видимости (например, когда функция заканчивается), она автоматически удаляется из стека в порядке ***LIFO***. Когда стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных.

Из-за такой природы стека управление памятью оказывается весьма логичным и простым для выполнения на ЦП (центральном процессоре);

[19] *Поток* данных (англ. *stream*) в современном мультипрограммировании — это абстракция, используемая для чтения или записи файлов (в том числе заданий). в единой манере. Потоки являются удобным унифицированным программным интерфейсом для чтения или записи файлов (в том числе специальных и, в частности, связанных с устройствами), сокетов и передачи данных между процессами.

это приводит к высокой скорости, в особенности потому, что время цикла обновления байта стека очень мало, то есть этот байт, скорее всего, привязан к кэшу процессора. Тем не менее, у такой строгой формы управления есть и недостатки.

Размер стека — это фиксированная величина, и превышение лимита выделенной на стеке памяти приведёт к *переполнению стека*. Размер стека задаётся при создании потока, у каждой переменной также есть максимальный размер, зависящий от типа данных. Это позволяет ограничивать размер некоторых переменных (например, целочисленных) и вынуждает заранее объявлять размер более сложных типов данных (например, массивов), поскольку стек не позволит им изменить размер стека. Кроме того, переменные, расположенные на стеке, всегда являются локальными.

В итоге стек позволяет управлять памятью наиболее эффективным образом. Но если нужно использовать динамические структуры данных или глобальные переменные, стоит обратить внимание на другую структуру управления памятью — кучу.

Куча — это хранилище памяти, также расположенное в ОЗУ (в оперативном запоминающем устройстве), которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для переменных. Когда выделяется в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении. Именно так определяются *глобальные* переменные. По завершении приложения все выделенные участки памяти освобождаются.

Размер кучи задаётся при запуске приложения, но, в отличие от стека, он ограничен лишь физическими границами памяти для кучи, и это позволяет создавать динамические переменные.

Можно взаимодействовать с кучей посредством ссылок, обычно называемых указателями (напомним, что указатели — это переменные, чьи значения являются адресами других переменных). Создавая указатель, программист указывает на местоположение памяти в куче для этого указателя, что задаёт начальное значение переменной и говорит программе, где получить доступ к этому значению.

Из-за динамической природы кучи центральный процессор (ЦП) *не принимает участия в контроле* над структурой кучи. В этом случае в языках без сборщика мусора (яркий пример — языки программирования Си, Си++) разработчику нужно вручную освобождать участки памяти, которые больше не нужны. Если этого не делать, могут возникнуть утечки (несвоевременные освобождения памяти) и фрагментация памяти (превращение памяти в «мелкодырочную» структуру), что существенно замедлит работу кучи. Одной из причин замедления работы кучи является периодическое выполнение дефрагментации для сбора уже неиспользуемой «мелкодырчатой памяти» и переструктуризации её для получения непрерывной области памяти большого размера.

В сравнении со стеком, куча работает медленнее, так как переменные разбросаны по памяти, а не сидят на верхушке стека. Некорректное управление памятью в куче приводит к замедлению её работы; тем не менее, это не уменьшает её важности — если вам нужно работать с динамическими или глобальными переменными, придётся пользоваться кучей.

Таким образом, стек — это очень быстрое хранилище памяти, работающее по принципу ***LIFO*** и управляемое процессором. Но эти преимущества приводят к ограниченному размеру стека и специальному способу получения значений. Для того, чтобы избежать этих ограничений, можно пользоваться кучей — она позволяет создавать динамические и глобальные переменные — но управлять памятью должен либо сборщик мусора, либо сам программист, к тому же работает куча медленнее.

10.2. Реализация обратной польской записи арифметических выражений

Полезный пример демонстрации работы стека представляет процесс вычисления арифметического выражения на основе такого вида внутреннего представления программы, как обратная польская запись

В главе 9 пункт 9.1 был приведен классический алгоритм вычисления арифметического выражения с выбором операндов из входного потока непосредственно.

Модифицируем первоначальный алгоритм так, чтобы операнды выбирались не из входного потока непосредственно, а из специально отведенного для этого *стека*. Как и раньше, договоримся, что в регистре ***R*** (определяемого функцией ***R()***) будет храниться текущий считываемый символ (верхний символ стека).

1. Если символ, содержащийся в ***R,*** является идентификатором или константой, то его значение заносится в стек и осуществляется считывание следующего символа (правило "***<операнд>::=идентификатор***").
2. Если символ, содержащийся в ***R,*** — бинарный оператор, то он применяется к двум верхним операндам в стеке, и затем они меняются на полученный результат (правило "***<операнд>::=<операнд><операнд><оператор>***").
3. Если символ, содержащийся в ***R,*** — унарный оператор, то он применяется к верхнему символу в стеке, который затем заменяется на полученный результат (правило "***<операнд>::=<операнд><оператор>***").

Упрощенно алгоритм вычисления арифметического выражения с использованием обратной польской формы (ПФ) выглядит как в примере 10.1.

Пример 10.1.

```
R = getchar;
while (R != #)
 {switch( R)
   { case 'идентификатор': push(R);
     case 'бин.оператор': pop(B); pop(A); Tmp = R(A,B); push(Tmp);
     case 'унарн.оператор': pop(A); Tmp = R(A); push(Tmp);
   }
  R = getchar;
 }
```

Замечание. Следует помнить, что существуют операторы, которые не заносят в стек результаты своей работы. Таким оператором является, к примеру, оператор присваивания языка Паскаль. Однако в языке Си (Си++) оператор присваивания *возвращает* значение.

Обратную польскую форму легко расширять. Рассмотрим включение в ПФ (польской формы) других, отличных от арифметических, операторов и конструкций.

Договоримся, что символом $ мы будем предварять имена операторов, чтобы не спутать их с аргументами.

Как же реализуются в этих условиях *безусловные переходы.*

Безусловный переход на метку (аналог ***GOTO L***):

L $BRL (Branch to label),

где ***L*** — имя в таблице символов. Значение его — адрес перехода. Основная проблема при реализации этого оператора — определение адреса перехода.

Возможным решением является введение фиктивного оператора, соответствующего метке, на которую будет осуществлен переход с последующим поиском его в массиве польской формы.

Безусловный переход на символ ***C $BR***

Под символом здесь понимается номер позиции (индекс) в массиве, содержащем польскую форму.

Условные переходы

<операнд1> <операнд2> $BRxZ | $BRM | $BRP | $BRMZ | $BRPZ, где

<операнд1> — значение арифметического выражения,

<операнд2> — номер или место символа в польской цепочке (адрес).

Здесь ***$BRx*** — код оператора.

При этом можно ввести самые разнообразные условия перехода, например:

$BRZ — переход по значению **0**,

$BRM — переход по значению **<0**,

$BRP — переход по значению **>0**,

$BRMZ — переход по значению **<=0**,

$BRPZ — переход по значению **>=0**,

и тому подобное.

При выполнении условия перехода в качестве следующего символа берется тот символ, адрес которого определяется вторым операндом. В противном случае работа продолжается как обычно.

Условные конструкции. Полный условный оператор выглядит следующим образом:

IF <выражение> { <инстр1>} ELSE {<инстр2>};

В представлении обратной польской формы (ПФ) получим:

<выражение> <c_1> $BRZ <инстр$_1$> <$c_2$> $BR <инстр$_2$>

Предполагается, что для любого значение выражения **«0»** означает *«ложь»*, а неравенство нулю *«истина»*.

В вышеприведённом представлении следующие обозначения:

<c_1> — номер символа, с которого начинается ***<инстр$_2$>***,

<c_2> — номер символа, следующего за <инстр$_2$>,

$BR — оператор безусловного перехода на символ,

$BRZ — оператор перехода на символ c_1, если значение ***<выражение>*** равно нулю).

Рис. 10.1. Обратная польская форма полного условного оператора

Замечание. При выполнении операторов перехода из стека лишь извлекается необходимое количество символов, а обратно ничего не возвращается.

Обратите внимание, что применение обратной польской формы для условной конструкции в виде

<выр> <инстр1> <инстр2> $IF

неприемлемо, потому что к моменту выполнения оператора ***$IF*** все три операнда должны быть уже вычислены или выполнены, что, неверно (так как в польской записи используются при вычислении только два известных операнда).

Изучим теперь, как в польской обратной записи представляются:

- о*писание массивов,*
- *обращение к элементу массива,*
- *и вызовы подпрограмм.*

Начнем с описания массивов. Пусть дано следующее *описание массива*:

int A[U1-L1][Uk-Lk].

В польской обратной форме это *описание массива* представляется в виде

L1 U1 ... Lk Uk A $ADEC,

где ***$ADEC*** — оператор объявления массива. Оператор ***$ADEC*** имеет *переменное количество аргументов*, зависящее от числа индексов.

Операнд ***A*** – адрес в таблице символов. При вычислении ***$ADEC*** из этого элемента таблицы извлекается информация о размерности массива ***A*** и, следовательно, о количестве элементов массива А операндов ***$ADEC.***

Отсюда понятно, почему операнд ***A*** должен располагаться непосредственно перед оператором ***$ADEC***. А именно, сначала в стек записывается адрес ***A***, из ***$ADEC*** извлекается количество элементов массива, затем в стек кладутся элементы массива А в соответствующем количестве, взятом из ***$ADEC.***

Обратимся теперь к случаю о*бращения к элементу массива.*

Выборка элемента ***A[i]…[k]*** представляемая ив общем виде конструкцией ***A[<выр>,..,<выр>]*** записывается в виде ***<выр> .. <выр> A $SUBS***

Оператор ***$SUBS***, используя элемент ***A*** таблицы символов и индексные выражения, вычисляет адрес элемента массива. Затем операнды исключаются из стека и на их место заносится новый операнд, специфицирующий тип элемента массива и его адрес.

Представление в польской обратной форме *вызова функций, процедур, подпрограмм.*

Методика обращения к элементу массива представляется очень важной для понимания того, как происходит вызов функций и процедур. Дело в том, что при обращении к функции также необходимо извлекать откуда-то необходимое количество аргументов. А количество их можно определить по содержимому таблицы имен. Поэтому вызов функции вида $f(x_1, x_2)$ в польской форме записи будет выглядеть как

$$x_1 x_2 f\$CALL,$$

где x_1 и x_2 — аргументы, ***f*** — имя функции, ***$CALL*** — команда передачи управления по адресу, определяемому именем функции. При этом предполагается, что мы заранее занесли в таблицу имен информацию о том:

- сколько и каких аргументов у функции ***f,***
- а также ее адрес — индекс начала кода подпрограммы в массиве для обратной польской формы.

Не случайно поэтому в подавляющем количестве языков программирования имена функций ассоциируются с их адресами.

Представление *циклов* в обратной польской форме.

Реализация циклов аналогично выше сказанному не вызывает сложностей. Во-первых, имея оператор безусловного перехода и условный оператор, можно всегда сконструировать цикл "вручную" (пример 10.2).

Пример 10.2. Представление *циклов* в обратной польской форме

Например, цикл вида

FOR (I=N1; I< N2; I++) Operator

может быть сконструирован на низкоуровневом языке программирования:

I = N1;
L1: IF (I>N2) GOTO L2;
Operator;
I=I+1;
GOTO L1;
L2:

Тем не менее, если необходимо реализовать этот цикл как оператор языка, то в польской форме записи он будет выглядеть так:

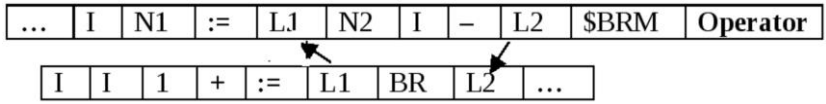


Рис. 10.2. Схема представления цикла примера 10.2 для обратной польской формы

Рассмотрим фрагмент программы (пример 10.3):

Пример 10.3

```
int K;
int A[i-j];
K = 0;
L: if (i>j)      {K := K+A[i-j]*6}
   else
     {i = i+1;
      return L;
     }
  }.
```

Добавим еще два оператора без операндов – ***$BLOCK*** ("начало блока") и ***$BLOCKEND*** ("конец блока").

Эти операторы иногда очень важны — в некоторых языках программирования с их помощью можно вводить локальные, "внутриблоковые" переменные, вызывать автоматически деструкторы и тому подобное. Кроме того, понадобится оператор присваивания «=», который ничего не заносит обратно в стек. Польская обратная форма фрагмента примера 10.3 выглядит следующим образом:

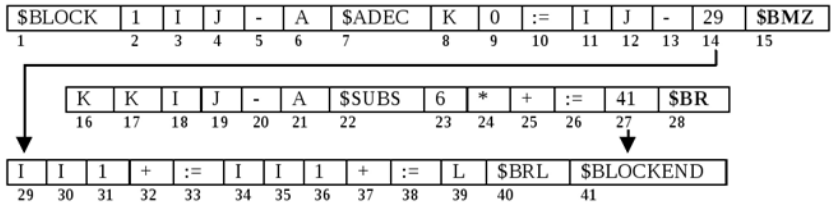


Рис. 10.3. Схема представления цикла примера 10.3 для обратной польской формы

Замечание. Хранение оператора, ключевого слова и других элементов языка в массиве обратной польской формы единообразно: в ней содержится лишь числовой код (например, в виде отрицательного числа). Операнды тоже представлены числами: индексами таблицы имен, где хранятся и переменные, и константы.

Кроме стека для команд, описаний массивов, вызовов функций строится также *стек аргументов*, необходимый для вычисления польской обратной формы, в котором хранятся значения рабочих (промежуточных) переменных. В качестве таких переменных могут быть и целые, и действительные числа, строки, символы, адреса и тому подобные элементы языка. Следовательно, необходимо предусмотреть хранение, помимо собственно значения, и типа аргумента, и размера и, возможно, многой другой информации. На это очень важно обращать особое внимание при разработке компиляторов.

Одним из возможных вариантов организации рабочего стека является использование так называемого *тегового* стека. В теговом стеке каждому хранимому элементу предшествует некий описатель его типа, называемый *тегом*. Тег может содержать самую разнообразную информацию, описывающую данные: тип, размер, права доступа и тому подобное.

Так, например, для простейшего интерпретатора можно различать такие типы данных, как числовые, строковые и адресные. В качестве адреса в простейшем случае может использоваться целое число — номер записи в таблице имен.

Выше все наши примеры касались вида *обратной* польской записи, чаще всего используемой при разработке трансляторов. Однако, существует вид польской формы, называемой *префиксной,* а также привычный вид формы, называемой *инфиксной*.

Инфиксная нотация — это привычная запись арифметического выражения в виде ***a+c***, когда операторы записываются между операндами, с которыми они взаимодействуют. В *постфиксной нотации* (обратная польская запись) операторы записаны после операндов: ***ac+***. В префиксной записи операторы записываются перед операндами: ***+ac***.

Рассмотрим пример *перевода из инфиксной нотации в постфиксную (обратную польскую запись),* используя алгоритм Дейкстры.

Пример 10.4. Переведём выражение ***(1+2)*4*** в обратную польскую форму

Для реализации метода понадобится стек, куда будут записываться математические операторы (возьмём простейший набор операторов: «+», «-», «*», «/»). Заготовим 2 стека: один — для построения обратной польской формы, второй — для знаков операций. В конце строки символов ***(1+2)*4*** поставим знак «\» для сигнализации о конце строки. Анализировать будем слева направо строку с маркером конца строки: ***(1+2)*4***|

Шаг 1. Сначала видим открывающую скобку «**(**», отправляем его в стек (правый) для операторов. Следующий символ — число «***1***», отправляем в левый стек для получения обратной польской формы (рис. 10.4).

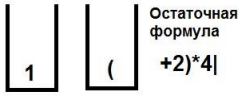


Рис. 10.4. Шаг 1

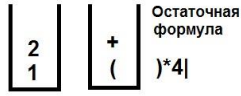


Рис. 10.5. Шаг 2

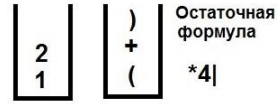


Рис. 10.6. Шаг 3

Шаг 2. Следующий символ — знак *«+»*, кладём его в стек операций (правый). Далее символ — число *«2»*, записываем его в левый стек (рис. 10.5).

Шаг 3. Следующий символ — знак закрывающая скобка «**)**», кладём его в стек операций (правый, рис. 10.6).

Шаг 4. Как только нашли закрывающую скобку «**)**», движемся вниз по правому стеку, выбираем часть содержимого правого стека между скобками открывающей «**(**» и закрывающей «**)**»**:** это знак плюс «**+**»**,** переносим выбранный фрагмент («+») в левый стек (рис. 10.7). Правый стек стал пустым.

Шаг 5. Следующий символ — знак умножения «*******», кладём его в стек операций (правый). Далее символ — число *«4»*, записываем его в левый стек (рис. 10.8).

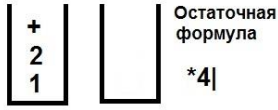


Рис. 10.7. Шаг 4

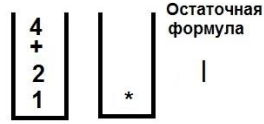


Рис. 10.8. Шаг 5

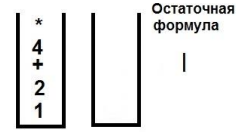


Рис. 10.9. Шаг 6

Шаг 6. Следующий символ — маркер конца строки «|», перекладываем символ «*******» из стека операций (правый стек) в стек обратной польской формы (левый стек). Маркер конца строки никуда не записывается, так как не является частью исходной строки (рис. 10.9), а означает прекращение процесса построения обратной польской формы.

Получили обратную польскую форму *«12+4*»* для инфиксной записи выражения ***(1+2)*4.***

С вычислением все гораздо проще. Снова будем использовать стек, куда будем помещать все операнды (числа). На примере только что полученного выражения в постфиксной нотации продемонстрируем получение постфиксной записи для вычислений арифметических выражений. Необходимо перебирать все элементы выражения, числа помещать в стек, а при встрече оператора проводить соответствующие манипуляции с двумя последними числами в стеке.

Таблица 10.1. Вычисление обратной польской формы ***12+4****

Шаг	*ОПЗ*	*Стек с операндами*	*Действия*
Исходные значения	*1 2 + 4 **	*нет*	
1	*2 + 4 **	*1*	
2	*+ 4 **	*1 2*	
3	*4 **	*3*	Выделили знак «+». Суммируем два последних числа в стеке (1+2), результат оставляем в стеке (3)
4	***	*3, 4*	
5		*12*	Выделили знак «*». Перемножаем два последних числа (3*4) результат оставляем в стеке (12)

Теперь исследуем пример посложнее: рассмотрим выражение в обратной польской записи

8 2 5 * + 1 3 2 * + 4 — /,

которое в инфиксной нотации соответствует выражению

(8 + 2 * 5) / (1 + 3 * 2 — 4):

Таблица 10.2. Вычисление обратной польской формы ***8 2 5 * + 1 3 2 * + 4 —/***

Шаг	**Остаток обратной польской записи**	**Стек с операндами**	**Действия**
1	*2 5 * + 1 3 2 * + 4 — /*	*8*	
2	*5 * + 1 3 2 * + 4 — /*	*8, 2*	
3	*** + 1 3 2 * + 4 — /*	*8, 2, 5*	
4	*+ 1 3 2 * + 4 — /*	*8, 10*	Умножаем 2 на 5
5	*1 3 2 * + 4 — /*	*18*	Суммируем 8 и 10
6	*3 2 * + 4 — /*	*18, 1*	
7	*2 * + 4 — /*	*18, 1, 3*	
8	*** + 4 — /*	*18, 1, 3, 2*	
9	*+ 4 — /*	*18, 1, 6*	Умножаем 3 на 2
10	*4 — /*	*18, 7*	Суммируем 1 и 6
11	*— /*	*18, 7, 4*	
12	*/*	*18, 3*	Вычитаем из 7 4
13		*6*	Делим 18 на 3

Вопросы для контроля

1. Дайте определение понятия стек. Дайте краткую характеристику функционирования стека.

2. Дайте определение понятия куча. Дайте краткую характеристику функционирования кучи.

3. Какая из структур организации памяти работает с фиксированной, а какая с динамической памятью?

4. В чём состоит процесс преобразования арифметического выражения в инфиксной форме в обратную польскую форму?

5. Как формируется обратная польская форма для операторов (арифметических выражений, безусловных переходов, условных выражений)?

Глава 11. Организация списочной структуры памяти.

Учебные цели:

- *проанализировать примеры работы со структурами;*
- *познакомить с реализацией функций над списками;*
- *разобрать пример работы с деревьями.*

11.1. Примеры работы со структурами

Структура в языках Си/Си++ — это тип данных, создаваемый программистом, предназначенный для объединения данных различных типов в единое целое. Прежде чем использовать в своей программе структуру, необходимо её описать, то есть описать её внутреннее устройство.

Структура обыкновенно имеет несколько полей разного типа, в том числе и пользовательского. Поля структуры расположены в памяти друг за другом. Тип поля определяет сдвиг относительно предыдущего поля. Имя поля — это сдвиг относительно адреса экземпляра. На самом деле размер структуры не всегда равен сумме размеров её полей: это связано с тем, что компилятор оптимизирует расположение структуры в памяти и может поля небольшого размера подгонять до чётных (или кратных 4) адресов (глава 8 пункт 8.1).

Рассмотрим два примера описания структур ***n1*** и ***n2*** и размещение элементов структуры в памяти с учетом выравнивания.

В *структуре* ***n1*** элементы структуры (рисунок 11.1 левый столбец) целые переменные ***x*** и ***y*** имеют размер в 2 байта, поэтому их можно экономно разместить друг за другом: например ***int x***, начиная с относительного адреса ***0***, а ***int y*** с адреса ***2***.

Пример 11.1. *Описания структур*

Структура *n1*

```
struct N1
 {int x;
  int y;
 } n1;
```

Структура *n2*

```
struct N2
 {int id;
 char*login;
 char*pasword;
 } n2;
```

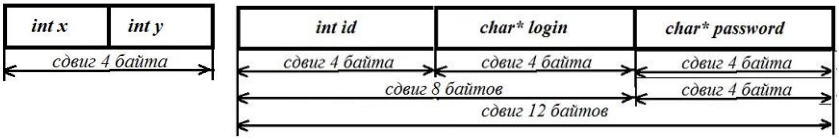


Рис. 11.1. Представление структуры в памяти
для структуры ***n1*** (слева) и для структуры ***n2*** (справа)

В описании структуры ***n2*** (рисунок 11.1 правый столбец) элементы с учетом выравнивания расположены следующим образом: первый элемент описания (целая переменная ***id***) имеет размер 2 байта и, предположим, располагается по относительному адресу «***0»***.

Следующая переменная ***char* login*** является указателем и имеет размер 4 байта, поэтому должнна располагаться с адреса, кратного 4, а именно, с относительного адреса 4. Третья переменная ***char* pasword*** является также указателем и имеет размер 4 байта, поэтому должнна также располагаться с адреса, кратного 4, а именно, с относительного адреса 8.

Рассмотрим более сложный пример (пример 11.2), содержащий несколько структур.

Пример 11.2.

```
#include <conio.h>
#include <stdio.h>
struct Test1
  {char a;
   char b;
   int c;
  } A;
struct Test2
  {int x;
   int y;
  } B;
struct Test3
  {char a;
   char b;
   char c;
   int d;
  } C;
int  main()
  {printf("sizeof(A) = %d\n", sizeof(A));
   printf("sizeof(B) = %d\n", sizeof(B));
   printf("sizeof(C) = %d\n", sizeof(C));
   getch();
   return 0;
  }
```

Первая структура должна иметь размер 6 байт, вторая — 8 байт, третья 7 — байт, однако на 32-разрядной машине компилятор сделает их все три равными 8 байтам (рис. 11.2—11.4).

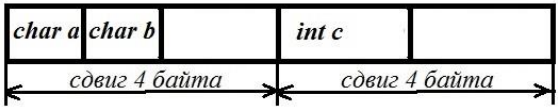


Рис. 11.2. Размещение в памяти структуры ***Test 1***

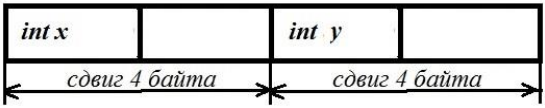


Рис. 11.3. Размещение в памяти структуры ***Test 2***

И, наконец, размещение элементов третьей структуры ***Test3***.

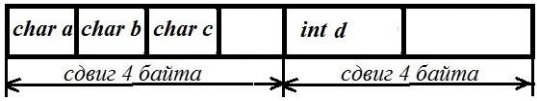


Рис. 11.4. Размещение в памяти структуры ***Test 3***

Стандарт гарантирует, что поля структуры расположены друг за другом, как в описании, но не гарантирует, что непрерывным блоком.

Хотя существует возможность изменить упаковку структур в памяти. Можно явно указать компилятору, каким образом производить упаковку полей структуры, объединений или полей класса. Каким образом это делать, зависит от компилятора. Один из самых распространённых способов — использование директивы ***#pragma pack()***. Эта директива записывается перед описанием структур.

Таким образом, процессом выравнивания можно управлять:

— оптимально разместив переменные в описании программы,

— или директивой ***#pragma pack( )***, принудительно определяя процесс расположения байтов в памяти.

Непосредственно управлять выравниванием нужно, когда существует значительное ограничение памяти, хотя такой способ серьёзно замедляет работу программы.

У днрективы ***#pragma pack()*** есть несколько разновидностей. Так, например, параметр директивы указывает значение в байтах, используемое для упаковки. Если параметр компилятора не задан для модуля, по умолчанию берётся значение ***n***, равное ***8***. Допустимыми значениями параметра являются значения 1, 2, 4, 8 и 16. Выравнивание каждого поля происходит по адресу, кратному ***n*** или сумме нескольких полей объекта, в зависимости от того, какая из этих величин меньше.

Замечание. Использование ***#pragma pack( )*** в большинстве разработок не приветствуется, потому что логика работы программы не должна зависеть от внутреннего представления структуры (кроме разработки системных программ или занятий хакерством, когда ломаются чужие программы и сети).

Основными операциями над списками являются:

— переход к очередному элементу списка;
— добавление в список нового элемента;
— поиск заданного элемента;
— удаление элемента из списка,
— сортировка списков,
— слияние списков

Выполнение этих операций основывается на работе с указателями.

В отличие от очередей, в которых элементы списка добавляются в конец, а удаляются с начала списка (режим ***FIFO)*** и стеков, в которых элемент списка добавляется в конец и удаляется с конца (режим ***LIFO***), элемент в список может быть добавлен в любое место списка и удалён из любого места списка.

Над списками выполняются различные операции, рассмотрим классический набор операций:

— создание пустого списка,
— добавление в список нового элемента,
— поиск элемента в списке,
— удаление элемента из списка,
— добавление элемента в список после заданного элемента

11.2. Реализация функций над списками.

Рассмотрим реализацию указанных выше операций над списками, организованных *с помощью массива*. Будем формировать двумерный массив, в котором в первой строке содержатся значения элементов массива, а во второй строке — индекс следующего элемента списка, ***0*** во второй строке означает конец списка.

Для удобства будем считать, что индексация элементов массива начинается с единицы. Пусть массив определен для 10 элементов списка (то есть ***maxlist*** определяет максимальное количество элементов списка).

Создание пустого списка

Прежде всего, определим значение указателей для пустого списка.

Пусть переменная ***US*** задаёт *адрес первого элемента* списка *свободных мест*, а переменная ***UN*** — *адрес первого элемента списка*. Для пустого списка ***US=1***, ***UN = 0***. Приведем пример создания пустого списка при помощи функции ***New***.

Пример 11.3. *Функция создания пустого списка*

```
void New()
  {for (i = l; i < maxlist-l; i++)
   List[1][i] = i+1;
   List[1][maxlist] = 0;
   US = 1;
   UN = 0;
  }
```

Таблица 11.1. Пустой список, представленный массивом

Характеристики	**Значения**									
Индексы массива	1	2	3	4	5	6	7	8	9	10
Значение элемента списка (***UN***)	0	0	0	0	0	0	0	0	0	0
Индекс — указатель на следующий элемент списка (***US***)	2	3	4	5	6	7	8	9	10	0

Добавление в список нового элемента

Для добавления элемента ***x*** в список необходимо в рабочую переменную ***Pos*** запомнить *адрес первого элемента списка свободных мест*. Затем нужно поместить по этому адресу элемент ***x*** и в качестве указателя на следующий элемент списка записать номер первого элемента списка. Теперь указателем первого элемента списка будет указатель ***US***, так как по его адресу записан новый элемент. Он из разряда свободных элементов переходит в разряд занятых. Указателем списка свободных мест будет ***Pos***, то есть элемент, который следует в списке свободных мест за только что исключенным из него элементом.

Приведем пример кода функции ***Insert (int, int, int)***, которая добавляет элемент в список.

Пример 11.4. *Функция добавления элемента в список*

```
void Insert (int US, int UN, int X)
  {if (Sp = = 0) exit(l); // Список не содержит элементов
   Pos = List[1][US];   /*Находит адрес первого элемента списка
                          свободных мест*/
   List[0][US] = X; // Помещает по этому адресу значение X
   List[1][US] = UN; /* В качестве указателя на следующий элемент
                        помещает номер первого элемента списка*/
   UN = US;
   US = POS;
  }
```

Рассмотрим подробно операцию добавления элемента ***x*** в список. Перед началом операции указатель списка свободных мест ***US=1***, а первый элемент списка ***UN=0***.

Добавим в список элемент ***A***, используя функцию ***Insert (int, int, int)***. После выполнения операции: ***US = 2***, а ***UN = A*** (таблица 11.2).

Таблица 11.2. *Добавление в список элемента списка*

Характеристики	**Значения**									
Индексы массива	1	2	3	4	5	6	7	8	9	10
Значение элемента списка (***UN***)	***A***	0	0	0	0	0	0	0	0	0
Индекс — указатель на следующий элемент списка (***US***)	2	3	4	5	6	7	8	9	10	0

Добавим в список элемент ***B***, используя функцию ***Insert(int, int, int).*** Теперь ***US = 3, UN = B*** (таблица 11.3).

Таблица 11.3. *Добавление в список элемента* ***B***

Характеристики	**Значения**									
Индексы массива	1	2	3	4	5	6	7	8	9	10
Значение элемента списка (***UN***)	***A***	***B***	0	0	0	0	0	0	0	0
Индекс — указатель на следующий элемент списка (***US***)	2	3	4	5	6	7	8	9	10	0

Аналогичным образом добавим в таблицу элементы ***C*** и ***D.*** ***US*** и ***UN*** изменились следующим образом ***US = 5,*** а ***UN = D*** (таблица 11.4).

Таблица 11.4. *Список после добавлениям в него элементов* ***C*** *и* ***D***

Характеристики	**Значения**									
Индексы массива	1	2	3	4	5	6	7	8	9	10
Значение элемента списка (***UN***)	***A***	***B***	***C***	***D***	0	0	0	0	0	0
Индекс — указатель на следующий элемент списка (***US***)	2	3	4	5	6	7	8	9	10	0

Теперь список свободных позиций содержит элементы с индексами 5, 6, 7, 8, 9, 10, а первым элементом списка является элемент ***D***. Логическое расположение элементов в списке будет ***D, C, B, A***. Этот порядок определяется указателями.

Поиск элемента в списке

Поиск элемента ***x*** в списке осуществляется следующим образом. Переменной ***Lop*** присваиваем значение указателя первого элемента списка. Далее просматривается список до тех пор, пока не найдется нужный элемент, либо поиск не достигнет конца списка. Если по окончанию процедуры ***Lop = 0***, значит, в списке нет элемента со значением ***x***.

Функция ***Find (int, int, int)*** организовывает поиск элемента со значением ***x*** в заданном списке (пример 11.5).

Пример 11.5. *Функция поиска элемента в списке*

```
void Find(int X, int UN)
  {Lop = UN;
   while (List[0][Lop] ! = X && Lop ! = 0)
        Lop = List[1][Lop];  /*Присваивает переменной Lop ссылку на
                                следующий элемент*/
  }
```

Удаление элемента из списка

Чтобы удалить элемент ***x*** из списка, необходимо сначала найти его в списке, например, при помощи функции ***Find (int, int, int)***.

При удалении изменяется указатель записи элемента, идущего в списке перед элементом ***x*** таким образом, чтобы он указывал на элемент, идущий в списке после элемента ***x***. Пусть ***Pos*** — индекс элемента массива, который необходимо удалить, а ***K*** — индекс элемента массива, который следует за элементом ***List[1] [POS]***.

Функция ***Delete (int, int, int)*** удаляет элемент ***x*** из списка (пример 11.6).

Пример 11.6. *Удаление элемента из списка*

```
void Delete (int K, int Pos, int US)
   {List[1][K] = List[1][Pos];
    List[1][Pos] = US;
    US = Pos;
  }
```

Продемонстрируем, как происходит удаление элемента ***x*** на примере списка из таблицы 11.4. Удалим элемент ***C*** из списка при помощи функции ***Delete (int, int, int)***.

Таблица 11.5. *Удаление элемента из списка*

Характеристики	**Значения**									
Индексы массива	1	2	3	4	5	6	7	8	9	10
Значение элемента списка (***UN***)	***A***	***B***	***C***	***D***	0	0	0	0	0	0
Индекс — указатель на следующий элемент списка (***US***)	0	1	5	2	6	7	8	9	10	0

После удаления элемента ***C*** операций указатель списка свободных мест ***US = 3***, а первым элементом списка ***UN*** = ***D***.

Добавление элемента в список после заданного элемента

Необходимо сначала найти заданный элемент ***x***, а затем вставить после него новый элемент ***E***. При добавлении нового элемента в список после элемента ***x*** по адресу первого элемента списка свободных мест записывается значение нового элемента. Указатель нового элемента принимает значение, равное адресу следующего за ним элемента списка, а указатель предшествующего ему элемента списка принимает значение адреса нового элемента.

Эти операции реализует функция ***Insert_after(int, int, int)***.

Пример 11.7. *Добавление элемента в список после заданного элемента*

```
void Insert_after (int Pos, int US, int E)
  {List[0][US] = E;
   List[1][US] = List[1][Pos];
   List[1][Pos] = US;
  }
```

В примере 11.7 ***POS*** — это индекс элемента ***x***.

После добавления элемента ***E*** после элемента ***B*** в таблице 11.5 получим список, приведенный в таблице 11.6.

Таблица 11.6. *Добавление элемента* ***D*** *после элемента* ***A***

Характеристики	**Значения**									
Индексы массива	1	2	3	4	5	6	7	8	9	10
Значение элемента списка (***UN***)	***A***	***B***	***C***	***D***	0	0	0	0	0	0
Индекс — указатель на следующий элемент списка (***US***)	0	1	5	2	6	7	8	9	10	0

Сортировка списков

При работе со списками очень часто возникает необходимость перестановки элементов списка в определенном порядке. Такая задача называется *сортировкой* списка, и для ее решения существуют различные методы.

Если в задаче требуется отсортировать все элементы списка, то он сортируется как обычный массив, любым из ранее предложенных методов (например, таблицы 11.7, 11.8).

Таблица 11.7. *Исходный список*

Характеристики	**Значения**				
Индексы массива	1	2	3	4	5
Значение элемента списка (***UN***)	***3***	***1***	***6***	***3***	***9***
Индекс — указатель на следующий элемент списка (***US***)	2	3	4	5	0

Таблица 11.8. *Отсортированный список*

Характеристики	**Значения**				
Индексы массива	1	2	3	4	5
Значение элемента списка (***UN***)	***1***	***3***	***3***	***6***	***9***
Индекс — указатель на следующий элемент списка (***US***)	2	3	4	5	0

В некоторых задачах возникает необходимость сортировки списка таким образом, чтобы по указателям можно было восстановить исходный список (например, таблицы 11.9, 11.10).

Таблица 11.9. *Исходный список*

Характеристики	**Значения**				
Индексы массива	1	2	3	4	5
Значение элемента списка (***UN***)	***3***	***1***	***6***	***3***	***9***
Индекс - указатель на следующий элемент списка (***US***)	2	3	4	5	0

Таблица 11.10. *Отсортированный список*

Характеристики	**Значения**				
Индексы массива	1	2	3	4	5
Значение элемента списка (***UN***)	***1***	***3***	***3***	***6***	***9***
Индекс - указатель на следующий элемент списка (***US***)	3	2	5	4	0

В данном случае при сортировке необходимо менять местами не только элементы списка, но и их указатели. Обмен значениями можно организовать функцией ***swap(x, y)***, которая будет менять местами элементы и указатели списка в позициях ***x*** и ***y***.

Пример 11.8. *Сортировка обменом*

```
void Swap (int x, int y)
  {int Temp;
   Temp = List[0][x];
   List[0][y] = Temp;
   Temp = List[1][x];
   List[1][x] = List[1][y];
   List[1][y] = Temp;
  }
```

Данная сортировка списка необходима для того, чтобы на некотором ***i***-ом шаге можно было вернуться на шаг под номером ***j*** и выполнить другие действия.

Слияние списков

Упорядоченные списки ***List_1*** и ***List_2*** длиной ***M*** и ***N*** сливаются в один упорядоченный список ***List_res*** длины ***M+N,*** если каждый элемент из ***List_1*** и ***List_2*** входит в ***List_res*** ровно один раз. В табл. 11.11—11.13 приведен пример *слияния списков*.

Таблица 11.11. *Упорядоченный список* ***List_1***

Характеристики	**Значения**				
Индексы массива	1	2	3	4	5
Значение элемента списка (***UN***)	***6***	***17***	***23***	***40***	***80***
Индекс — указатель на следующий элемент списка (***US***)	2	3	4	5	0

Таблица 11.12. *Упорядоченный список* ***List_2***

Характеристики	**Значения**				
Индексы массива	1	2	3	4	5
Значение элемента списка (***UN***)	***2***	***3***	***34***	***1***	***32***
Индекс — указатель на следующий элемент списка (***US***)	2	3	4	5	0

Таблица 11.13. *Упорядоченный список* ***List_res***

Характеристики	**Значения**									
Индексы массива	1	2	3	4	5	6	7	8	9	10
Значение элемента списка (***UN***)	***1***	***2***	***3***	***6***	17	23	32	34	40	80
Индекс — указатель на следующий элемент списка (***US***)	2	3	4	5	6	7	8	9	10	0

Алгоритм слияния двух списков весьма прост. Его основная идея состоит в следующем. Для слияния списков ***List_1*** и ***List_2*** список ***List_res*** сначала полагается пустым, а затем к нему последовательно приписывается первый узел из ***List_1*** или ***List_2***, оказавшийся меньшим и отсутствующий в ***List_res***.

11.3. Пример работы с деревьями

Применение абстрактных синтаксических деревьев

Абстрактные синтаксические деревья используются не только в интерпретаторах и компиляторах. Они, в мире компьютеров, оказываются полезными и во многих других областях. Один из наиболее часто встречающихся вариантов их применения — статический анализ кода. Статические анализаторы не выполняют передаваемый им код. Однако, несмотря на это, им нужно понимать структуру программ.

Предположим, необходимо разработать инструмент, который находит в коде часто встречающиеся структуры. Отчёты такого инструмента помогут в рефакторинге, позволят уменьшить дублирование кода. Сделать это можно, пользуясь обычным сравнением строк, но такой подход окажется весьма примитивным, возможности его будут ограниченными.

На самом деле, если необходимо создать подобный инструмент, не нужно писать собственный парсер[20] для *JavaScript*. Существует множество опенсорсных (с открытым кодом) реализаций подобных программ, которые полностью совместимы со спецификацией (стандартом) *ECMAScript*[21]. Например — *Esprima*[22] и *Acorn*. Существуют и инструменты, которые могут помочь в работе с тем, что генерируют парсеры, а именно, в работе с абстрактными синтаксическими деревьями.

Прежде всего, абстрактные синтаксические деревья широко используются при разработке транспиляторов[23]. Предположим, необходимо разработать транспилятор, преобразующий код на ***Python*** в код на ***JavaScript***. Подобный проект может быть основан на идее, в соответствии с которой используется транслятор для создания абстрактного синтаксического дерева на основе Python-кода, которое, в свою очередь, преобразуется в код на JavaScript.

Возникает вопрос, как процесс транспиляции организован. Как известно, абстрактные синтаксические деревья — это всего лишь альтернативный способ представления кода на некоем языке программирования. Иначе говоря, исходный текст (исходный код, который выглядит как обычный текст, при написании которого следуют определённым правилам, формирующим язык), преобразуется в AST (абстрактное синтаксическое дерево). Таким образом, после парсинга исходный код превращается в древовидную структуру, которая содержит ту же информацию, что и исходный текст программы. В результате можно осуществить не только переход от исходного кода к AST, но и обратное преобразование, превратив абстрактное синтаксическое дерево в текстовое представление кода программы.

Парсинг (трансляция объектно-ориентированного скриптового языка) JavaScript-кода.

Рассмотрим простую JavaScript-функцию и построим для этого фрагмента кода абстрактное синтаксическое дерево.

Пример 11.9.

```
function foo(x)
  {if (x > 10)
        { var a = 2;
          return a * x;
        }
```

---

[20] ***Парсер*** (*Parser*) — объектно-ориентированный скриптовый язык программирования, созданный для генерации *HTML*-страниц на веб-сервере с поддержкой *CGI*. Разработан Студией Артемия Лебедева и выпущен под лицензией, сходной с *GNU GPL*. Язык специально спроектирован и оптимизирован для того, чтобы было удобно создавать простые сайты. Модульность языка позволяет легко наращивать функциональность.

[21] ***ECMAScript*** — это встраиваемый расширяемый не имеющий средств ввода-вывода язык программирования, используемый в качестве основы для построения других скриптовых языков.

[22] Esprima — парсер для JavaScript, написанный на JavaScript/

[23] Термин *транспилер* (*transpiler*), или *транспилятор*, следует принимать как устоявшийся термин для *трансляции кода между двумя языками*, у которых примерно одинаковый уровень абстракции или другие зависимости. Иначе говоря, каждый транспилятор является транслятором, но не каждый транслятор — транспилятором.

```
  return x + 10;
}
```

Парсер создаст абстрактное синтаксическое дерево, которое схематично представлено на рисунке 11.6.

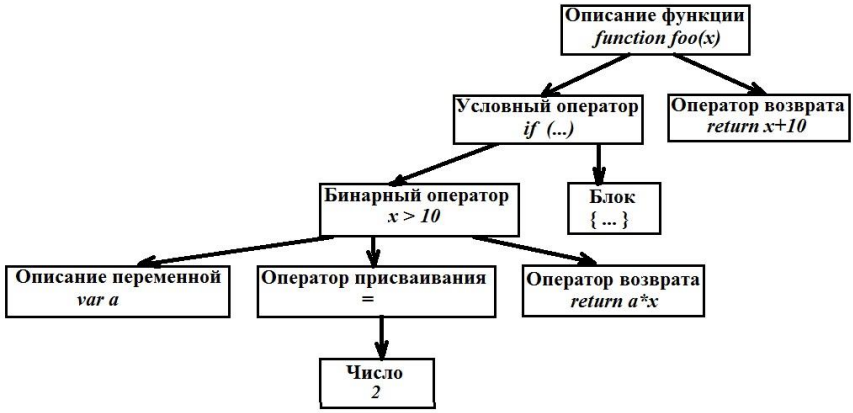


Рис. 11.6. Абстрактное синтаксическое дерево

На этом рисунке — упрощённое представление результатов работы парсера. Настоящее абстрактное синтаксическое дерево гораздо сложнее. В данном случае главная цель — получить представление о том, во что, в первую очередь, превращается исходный код, прежде чем он будет выполнен.

Построение бинарного дерева.

Важным понятием древовидной структуры является понятие двоичного дерева поиска. Напомним правило построения двоичного дерева поиска: элементы, у которых значение некоторого признака меньше, чем у корня, всегда включаются слева от некоторого поддерева, а элементы со значениями, большими, чем у корня — справа.

Этот принцип используется и при *формировании* двоичного дерева, и при *поиске* в нем элементов.

Таким образом, при поиске элемента с некоторым значением признака происходит спуск по дереву, начиная от корня, причем выбор ветви следующего шага (направо или налево согласно значению искомого признака) происходит в каждом очередном узле на этом пути. При поиске элемента результатом будет:

— либо найденный узел с заданным значением признака,

— либо поиск закончится листом с «нулевой» ссылкой, а требуемый элемент отсутствует на проделанном по дереву пути.

Если поиск был проделан для включения очередного узла в дерево, то в результате будет найден узел с пустой ссылкой (пустыми ссылками), к

которому справа или слева в соответствии со значением признака и будет присоединен новый узел.

Рассмотрим пример формирования двоичного дерева. Предположим, что нужно сформировать двоичное дерево, узлы (элементы) которого имеют следующие значения признака: ***20, 10, 35, 15, 17, 27, 24, 8, 30***. В этом же порядке они и будут поступать для включения в двоичное дерево.

Первым узлом в дереве (корнем) станет узел со значением ***20.***

Замечание. Поиск места подключения очередного элемента *всегда начинается с корня.*

К корню ***20*** слева подключается элемент ***10***. К корню справа подключается элемент ***35***. Далее элемент ***15*** подключается справа к ***10,*** проходя путь: от корня ***20*** — налево, от элемента ***10*** — направо, подключение элемента 15, так как дальше пути нет. Процесс продолжается до исчерпания включаемых элементов. Результат продемонстрирован на рисунке 11.7.

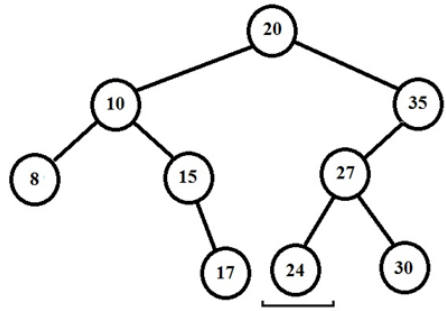


Рис. 11.7. Построение бинарного дерева.

Получили бинарное дерево со значениями элементов:

20, 10, 35, 15, 17, 27, 24, 8, 30.

При создании двоичного дерева отдельно решается вопрос с возможностью включения элементов с дублирующими признаками. Алгоритм двоичного поиска позволяет при включении корректно расположить узлы в дереве, однако при поиске уже включенных элементов возникают сложности, так как при стандартном варианте поиска будет выбираться только один из дублей.

Для обнаружения всех дублей должен быть применен алгоритм такого обхода дерева, при котором каждый узел дерева будет выбран один раз.

Решение задач работы с бинарным деревом.

Элемент дерева используется для хранения какой-либо информации, следовательно, он должен содержать информационные поля, возможно разнотипные. Элемент двоичного дерева связан в общем случае с двумя прямыми потомками, а при необходимости может быть добавлена и третья связь — с непосредственным предком. Отсюда следует, что по структуре

элемент дерева (узел) похож на элемент списка и может быть описан так же, как и в списке: в дереве должна существовать возможность доступа к его «первому» элементу — корню дерева. Возможность доступа реализуется через необходимую принадлежность дерева – поле-ссылка ***ROOT***, в котором записывается ссылка на корневой элемент.

Пример 11.10. *Описания полей и элементов, необходимых для построения дерева*

```
typedef struct tree
  {int key;
   struct tree *left;
   struct tree *right;
   struct tree *parent;
} node;
```

В этом фрагменте кода используем директиву ***typedef*** для создания нового типа, чтобы в дальнейшем не писать слово ***struct.*** Элементы (поля) структуры:

- ***int key —*** ключ, он может быть любого типа, не только ***int,***
- ***struct tree *left —*** указатель на левое поддерево,
- ***struct tree *right —*** указатель на правое поддерево,
- ***struct tree *parent —*** указатель на родителя,
- ***node —*** название структуры.

Пример 11.10 описания показывает, что описание элемента списка и узла дерева, по сути, ничем не отличаются друг от друга. Различия в технологии действий тоже невелики — основные действия выполняются над ссылками, адресами узлов. Основные различия — в алгоритмах.

При работе с двоичным деревом возможны следующие основные задачи:

1. инициализация дерева (отдельной функцией),
2. создание элемента, узла дерева,
3. включение его в дерево по алгоритму двоичного поиска,
4. нахождение в дереве узла с заданным значением ключевого признака,
5. определение максимальной глубины дерева,
6. определение количества узлов дерева,
7. определение количества листьев дерева,
8. ряд других задач.

Приведем примеры функций, реализующих основные задачи работы с бинарным деревом.

Пример 11.11. *Инициализация дерева.*

Инициализацию дерева лучше прописать отдельной функцией для того, чтобы облегчить процесс добавления узлов в дерево. Таким образом создается *корень* бинарного дерева поиска.

```
node *create(node *root, int key)
  { node *tmp = malloc(sizeof(node));/*Выделение памяти под корень дерева
```

```
    tmp -> key = key;                    // Присваивание значения ключу
    tmp -> parent = NULL;   /* Присваивание указателю на родителя
                                значения NULL*/
    tmp -> left = tmp -> right = NULL; //*Присваивание указателю на левое
                                и правое поддерево значения NULL*/
    root = tmp;
    return root;
  }
```

Пример 11.12. *Добавление узла в дерево*

```
node *add(node *root, int key)
  {node *root2 = root, *root3 = NULL;
   node *tmp = malloc(sizeof(node)); // Выделение памяти под узел дерева
   tmp -> key = key; // Присваивание значения ключу
/* Поиск нужной позиции для вставки (руководствуемся правилом
   вставки элементов) */
   while (root2 != NULL)
     { root3 = root2;
      if (key < root2 -> key)
            root2 = root2 -> left;
       else
            root2 = root2 -> right;
      }
/* Присваивание указателю на родителя значения указателя root3
   (указатель root3 был найден выше) */
   tmp -> parent = root3;
// Присваивание указателю на левое и правое поддерево значения NULL
   tmp -> left = NULL;
   tmp -> right = NULL;
/* Вставляем узел в дерево (руководствуемся правилом вставки элементов */
   if (key < root3 -> key)
            root3 -> left = tmp;
      else
            root3 -> right = tmp;
   return root;
  }
```

Некоторые замечания к фрагменту кода:

— ***tmp -> left*** и ***tmp -> right*** имеют значение ***NULL***, так как указатель ***tmp*** расположен в конце дерева,

— указатель ***root2*** использовался для того, чтобы сохранить адрес на родителя вставляемого узла**.**

— здесь не проверяется дерево на пустоту, так как ранее дерево было инициализировано (то есть имеется корень),
— элемент с дублирующим ключевым признаком в дерево не включается.

Замечание. Приведённый *алгоритм движения по дереву* может быть положен в основу задачи определения максимального уровня (глубины) двоичного дерева, определения, есть ли в дереве элемент с заданным значением ключевого признака и так далее, то есть таких задач, решение которых основывается на алгоритме двоичного поиска по дереву.

Однако не все задачи могут быть решены с применением двоичного поиска, например, подсчет общего числа узлов дерева. Для этого требуется алгоритм, позволяющий однократно посещать каждый узел дерева.

При посещении любого узла возможно однократное выполнение следующих трех действий:

1. обработать узел (конкретный набор действий при этом не важен). Обозначим это действие через ***О*** (обработка);
2. перейти по левой ссылке (обозначение — ***Л***);
3. перейти по правой ссылке (обозначение — ***П***).

Можно организовать *обход узлов* двоичного дерева, однократно выполняя над каждым узлом эту последовательность действий. Действия могут быть скомбинированы в произвольном порядке, но он должен быть постоянным в конкретной задаче обхода дерева.

Покажем на примере дерева, представленного ранее на рисунке 11.7 варианты обхода дерева.

1. Обход вида **ОЛП**. Такой обход называется «в прямом порядке», «в глубину». Он даёт следующий порядок посещения узлов:
   20, 10, 8, 15, 17, 35, 27, 24, 30
2. Обход вида **ЛОП**. Он называется «симметричным» и даёт следующий порядок посещения узлов:
   8, 10, 15, 17, 20, 24, 27, 30, 35
3. Обход вида **ЛПО**. Он называется «в обратном порядке» и даёт следующий порядок посещения узлов:
   8, 17, 15, 10, 24, 30, 27, 35, 20

Анализируя задачи, требующие сплошного обхода дерева, отметим, что для части из таких задач порядок обхода, в целом, не важен. Так для задач подсчета числа узлов дерева, числа листьев/не листьев, элементов, обладающих заданной информацией и так далее, порядок обхода не важен.

Ниже, в примерах 11.16-11.18 будут приведены примеры функций, реализующих эти 3 типа обходов дерева.

Пример 11.13. *Нахождение конкретного узла в дереве*

Поиск в дереве реализовать очень просто. Как и раньше, руководствоваться нужно следующим правилом: слева расположены элементы с меньшим значением ключа, справа — с большим.

```
node *search(node * root, int key)
  {if ((root == NULL) || (root -> key = key))
                                        return root;
/*Если дерево пусто или ключ корня равен искомому ключу, то возвращается указатель на корень*.
    if (key < root -> key)                    // Поиск нужного узла
        return search(root -> left, key);
      else
        return search(root -> right, key);
  }
```

Приведённая функция *рекурсивная*, поэтому комментарий *«Если дерево пусто или ключ корня равен искомому ключу, то возвращается указатель на корень»* является не совсем верным, потому что ***root*** указывает на корень только во время *первой итерации*, далее ***root*** ссылается на другие узлы дерева, но из-за рекурсивности функции условие

```
if ((root == NULL) || (root -> key = key))
```

будет проверяться всегда.

Пример 11.14. *Нахождение узла в дереве с минимальным и максимальным ключом*

```
// Минимальный элемент дерева
node *min(node *root)
  { node *l = root;
    while (l -> left != NULL)
          l = l -> left;
    return l;
  }
// Максимальный элемент дерева
node *max(node *root)
  {node *r = root;
    while (r -> right != NULL)
          r = r -> right;
    return r;
  }
```

Эти функции являются очень полезными, особенно часто используются во время удаления узла из дерева.

Пример 11.15. *Удаление узла из дерева*

Удаление элемента из дерева реализуется намного сложнее, чем в списках. Это связано с особенностью построения деревьев.

1. Самый простой случай: у удаляемого узла *нет левого и правого поддерева* (узел ***13***). В этой ситуации просто *удаляется* данный лист (узел).

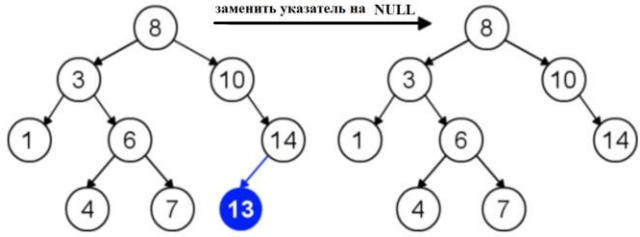


Рис. 11.8.Удаление узла, у которого нет ни левого, ни правого поддерева

2. У удаляемого узла *одно поддерево*. В этой ситуации просто удаляется данный узел, а на его место ставится поддерево (рис. 11.9).

3. Самый сложный случай: у удаляемого узла *существуют оба поддерева*. В этой ситуации необходимо сначала найти следующий за удаляемым элемент, а потом его поставить на место удаляемого элемента (рис. 11.10).

Для всех этих случаев будем использовать общую функцию поиска элемента, следующего за удаляемым элементом и, собственно, функцию удаления узла из дерева.

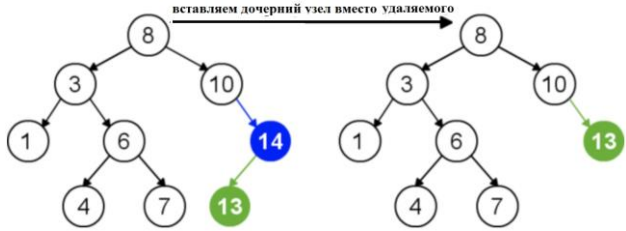


Рис. 11.9.Удаление узла, у которого одно поддерево

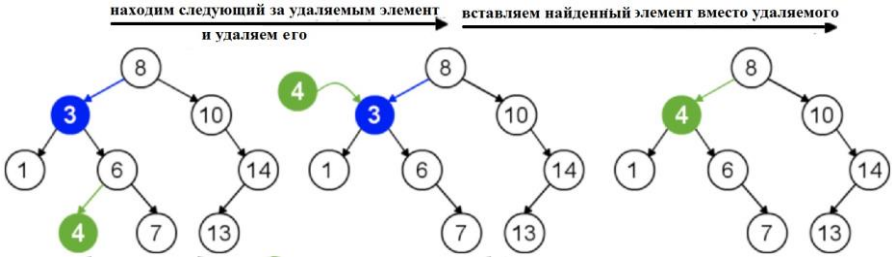


Рис. 11.10.Удаление узла, у которого одно поддерево

Общая функция поиска, следующего за удаляемым элемента:

```
node *succ(node *root)
  {node *p = root, *l = NULL;
/* Если есть правое поддерево, то ищем минимальный элемент в этом поддереве*/
```

```
    if (p -> right != NULL)
            return min(p -> right);
```
/* Правое дерево пусто, идем по родителям до тех пор, пока не найдем родителя, для которого наше поддерево левое */
```
    l = p -> parent;
    while ((l != NULL) && (p == l -> right))
        { p = l;
          l = l -> parent;
        }
    return l;
  }
```
Во втором случае из данного кода нужна только первая часть кода, так как поиск следующего за удаляемым элемента нужен тогда, когда у удаляемого узла существуют оба поддерева.

Функция удаления узла из дерева:
```
node *delete(node *root, int key)
  { node *p = root, *l = NULL, *m = NULL;
    l = search(root, key);    // Поиск удаляемого узла по ключу
// 1 случай
    if ((l -> left == NULL) && (l -> right == NULL))
          { m = l -> parent;
            if (l == m -> right)    m -> right == NULL;
                  else              m -> left == NULL;
             free(l);
           }
// 2 случай, 1 вариант — поддерево справа
    if ((l -> left == NULL) && (l -> right != NULL))
          { m = l -> parent;
            if (l == m -> right)   m -> right == l -> right;
                else               m -> left == l -> right;
             free(l);
          }
// 2 случай, 2 вариант — поддерево слева
    if ((l -> left != NULL) && (l -> right == NULL))
         {m = l -> parent;
          if (l == m -> right)     m -> right == l -> left;
              else        m -> left == l -> left;
            free(l);
         }
// 3 случай
    if ((l -> left != NULL) && (l -> right != NULL))
           {m = succ(l);
            l -> key = m -> key;
```

```
            if (m -> right == NULL)     m -> parent -> left = NULL;
                else        m -> parent -> left = m -> right;
             free(m);
           }
  return root;
}
```

Как было показано выше в замечании к примеру 11.12 (*Добавление узла в дерево*) принято выделять три типа обхода дерева: в прямом, обратном и симметричном порядке. Напомним, что на рисунке 11.7 изображено бинарное дерево со значениями в узлах: ***20, 10, 35, 15, 17, 27, 24, 8, 30***, если читать значения узлов по уровням слева направо**.** Этому дереву соответствует дерево рисунка 11.16, причём выдержано следующее соответствие:

20	10	35	15	17	27	24	8	30
A	B	C	D	E	F	G	H	J

Пример 11.16. *Вывод элементов дерева*

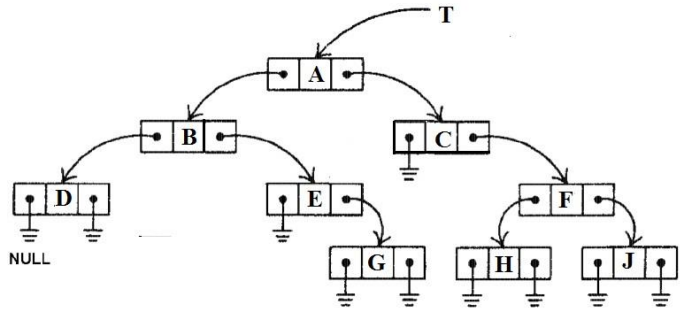


Рис. 11.16. Схема связей в дереве

Приведём коды функций, реализующие обходы дерева.

1. *Обход дерева в прямом порядке.*

Результат вывода:

20	10	8	15	17	35	27	24	30
A	B	H	D	E	C	F	G	J

```
void preorder(node *root)
  { if (root = = NULL)     return 0;
    if (root -> key)          printf("%d ", root -> key);
    preorder(root -> left);
    preorder(root -> right);
  }
```

2. *Обход дерева в обратном порядке.*

Результат вывода:

8	17	15	10	24	30	27	35	20
H	E	D	B	G	J	F	C	A

```
void postorder(node *root)
   {if (root == NULL)          return 0;
    postorder(root -> left);
    postorder(root -> right);
     if (root -> key)           printf("%d ", root -> key);
   }
```

3. Обход дерева в симметричном порядке.

Результат вывода:

8	10	15	17	20	24	27	30	35
H	B	D	E	A	G	F	A	C

```
void inorder(node *root)
     {if (root == NULL)          return 0;
      inorder(root -> left);
      if (root -> key)           printf("%d ", root -> key);
      inorder(root -> right);
     }
```

Однако такая задача, как уничтожение бинарного дерева с освобождением памяти, требует использования обхода только *в обратном порядке.*

Рекурсия в такой программе действует точно так же, как и в рекурсивных функциях работы со списками: создается цепочка вызовов функций, каждая из которых рекурсивно обращается к себе и затем ожидает завершения вызванной рекурсивной функции. При этом начинается последовательное завершение вызванных функций с возвратом управления в вызывающую функцию. Способ обхода меняется с изменением порядка обращений к функциям.

В заключение следует сказать о том, что рекурсивный обход дерева применим в большинстве задач, однако необходимо все же различать варианты эффективного применения двоичного поиска и сплошного обхода.

Вопросы для контроля

1. Как располагаются элементы структуры разного типа в оперативной памяти с учетом выравнивания? Приведите примеры. Укажите, какой объем памяти имеет структура в Вашем примере.
2. Какие функции над списками характерны для стека, кучи?
3. Опишите процесс поиска нужного элемента в списке.
4. Опишите процесс подключения очередного элемента в бинарном дереве.
5. Какие варианты обхода бинарного дерева Вы знаете, в чём их особенности?

Глава 12. Тестирование и отладка системного программного обеспечения

Учебные цели:

- *дать представление о системных средствах загрузки и отладки программного обеспечения;*
- *познакомить с приемами тестирования и отладки системного программного обеспечения;*
- *познакомить с методами оптимизации программ*

Как известно, в любой системе программирования основным ее модулем всегда является компилятор. На эффективность программ, создаваемых в среде программирования, прежде всего, влияют технические характеристики компилятора. Но важными составляющими, организующими связи между частями программы (модулями пользователя и модулями библиотек), оптимизирующими результат являются *компоновщик, загрузчик, отладчик*.

12.1. Системные средства загрузки и отладки программного обеспечения. Функции компоновщика, загрузчика, отладчика.

Компоновщик (редактор связей, линкер). Назначение и функции компоновщика.

Компоновщик (или редактор связей) предназначен для связывания между собой объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав системы программирования.

Объектный файл (или набор объектных файлов), который получен в результате фазы анализа, не может быть исполнен до тех пор, пока все модули и секции не будут в нем увязаны между собой. Эту функцию выполняет редактор связей (компоновщик): в результате его работы создается единый файл (часто называемый «исполняемым файлом»), который содержит весь текст результирующей программы на языке машинных кодов. Компоновщик может порождать сообщение об ошибке, если при попытке собрать объектные файлы в единое целое он не смог обнаружить какой-либо необходимой составляющей.

Функция компоновщика достаточно проста: сначала он выбирает из первого (основного) объектного модуля программную секцию ***main*** и присваивает ей начальный адрес (например, базовый относительный адрес ***0***). Программные секции остальных объектных модулей получают адреса относительно этого начального адреса в порядке следования. При этом может выполняться также функция выравнивания начальных адресов программных секций. Одновременно с объединением текстов программных секций объединяются

— секции данных,
— таблицы идентификаторов и
— внешних имен.

Разрешаются (вычисляются) межсекционные ссылки.

Процедура *разрешения ссылок* сводится к вычислению значений адресных констант, процедур, функций и переменных с учетом перемещений секций относительно начала собираемого программного модуля.

Если при этом обнаруживаются ссылки к внешним переменным, отсутствующим в списке объектных модулей, компоновщик (редактор связей) организует их поиск в библиотеках, доступных в системе программирования. Если же и в библиотеке необходимую составляющую найти не удается, формируется сообщение об ошибке.

Обычно компоновщик формирует простейший программный модуль, создаваемый как единое целое. Однако в более сложных случаях компоновщик может создавать и другие модули:

— программные модули с оверлейной структурой,
— объектные модули библиотек и
— модули динамически подключаемых библиотек.

Загрузчики и отладчики. Функции загрузчика

Компилятор, порождающий объектные файлы, а затем и компоновщик, объединяющий их в единое целое, не могут знать точно, в какой реальной области оперативной памяти компьютера будет располагаться программа в момент ее выполнения. Поэтому они работают не с реальными адресами ячеек ОЗУ (оперативного запоминающего устройства), а с некоторыми относительными адресами. Относительные адреса отсчитываются от некоторой условной точки, принятой за начало области памяти, занимаемой результирующей программой (обычно это точка начала первого (основного) модуля программы с базовым относительным адресом ***0***). Большинство объектных модулей в современных системах программирования строятся на основе таких относительных адресов.

Безусловно, ни одна программа не исполняется в этих относительных адресах. Поэтому требуется модуль, который выполняет преобразование относительных адресов в реальные (абсолютные) адреса непосредственно в момент запуска программы на выполнение. Этот процесс называется *трансляцией адресов* и выполняет его специальный модуль, называемый *загрузчиком*.

Однако, *загрузчик* не всегда является составной частью системы программирования, потому что функции, которые он выполняет, непосредственно зависят от архитектуры целевой вычислительной системы, в которой выполняется результирующая программа, созданная системой программирования.

На первых этапах развития операционных систем (ОС) загрузчики существовали в виде отдельных модулей, которые создавали так называемый «образ задачи»: выполняли трансляцию адресов и готовили программу к выполнению. Такая схема была характерна для многих ОС, начиная с 80-х годов (например, для ОСРВ на ЭВМ типа СМ-1, ОС RSX/11 или RAFOS на ЭВМ типа СМ-4 и тому подобное). Образ задачи можно было сохранить на внешнем носителе или же создавать его вновь всякий раз при подготовке программы к выполнению.

С развитием архитектуры вычислительных средств компьютеров появилась возможность выполнять *трансляцию адресов непосредственно* в момент запуска программы на выполнение. Для этого потребовалось в состав исполняемого файла включить соответствующую таблицу, содержащую перечень ссылок на адреса, которые необходимо подвергнуть трансляции. В момент запуска исполняемого файла ОС обрабатывала эту таблицу и преобразовывала относительные адреса в абсолютные.

Такая схема, например, характерна для операционных систем типа MS-DOS, которые широко распространены были в среде персональных компьютеров 2000-х годов. В этой схеме модуль загрузчика как таковой отсутствует (фактически он входит в состав ОС), а система программирования ответственна только за подготовку таблицы трансляции адресов: эту функцию выполняет компоновщик.

В современных операционных системах существуют сложные методы преобразования адресов, которые работают непосредственно уже во время выполнения программы. Эти методы основаны на возможностях, аппаратно заложенных в архитектуру вычислительных комплексов. Методы трансляции адресов могут быть основаны на сегментной, страничной и сегментно-страничной организации памяти. Тогда для выполнения трансляции адресов в момент запуска программы должны быть подготовлены соответствующие системные таблицы. Эти функции целиком теперь ложатся на модули операционной системы (ОС), поэтому они уже не выполняются в системах программирования.

Еще одним модулем системы программирования, функции которого тесно связаны с выполнением программы, является *отладчик*.

Отладчик — это программный модуль, который позволяет выполнить основные задачи, связанные с мониторингом процесса выполнения результирующей прикладной программы. Этот процесс называется *отладкой* и включает в себя следующие основные возможности:

— последовательное пошаговое выполнение результирующей программы на основе шагов по машинным командам или по операторам входного языка;
— выполнение результирующей программы до достижения ею одной из заданных точек останова (адресов останова);

— выполнение результирующей программы до наступления некоторых заданных условий, связанных с данными и адресами, обрабатываемыми этой программой;
— просмотр содержимого областей памяти, занятых командами или данными результирующей программы.

Первоначально *отладчики* представляли собой отдельные программные модули, которые могли обрабатывать результирующую программу на языке машинных команд. Возможности отладчиков, в основном, сводились к моделированию выполнения результирующих программ в архитектуре соответствующей вычислительной системы. Выполнение могло идти непрерывно либо по шагам. Дальнейшее развитие отладчиков связано со следующими принципиальными моментами:

— появлением интегрированных сред разработки;
— появление возможностей аппаратной поддержки средств отладки во многих вычислительных системах.

Первый шаг дал возможность разработчикам программ работать не в терминах машинных команд, а в терминах исходного языка программирования: это значительно сократило трудозатраты на отладку программного обеспечения. В результате этих изменений отладчики перестали быть отдельными модулями и стали интегрированной частью систем программирования, потому что задачей отладчика стала теперь функция поддержки работы с таблицами идентификаторов и выполнение процесса, обратного идентификации лексических единиц языка. Это связано тем, что в такой среде отладка программы идет в терминах имен, данных пользователем, а не в терминах внутренних имен, присвоенных компилятором. Соответствующие изменения потребовались также в функциях компиляторов и компоновщиков, поскольку они должны были включать таблицу имен в состав объектных и исполняемых файлов для ее обработки отладчиком.

Второй шаг позволил значительно расширить возможности *средств отладки:* для них теперь не требовалось моделировать работу и архитектуру соответствующей вычислительной системы. Выполнение результирующей программы в режиме отладки стало возможным в той же среде, в которой шел процесс создания программы. Таким образом, отладчику отводились теперь только функции перевода вычислительной системы в соответствующий режим перед запуском результирующей программы на отладку. Чаще всего, эти функции являются приоритетными, так как почти всегда требуют установки системных таблиц и флагов процессора вычислительной системы.

Отладчики в современных системах программирования представляют собой модули с развитым интерфейсом пользователя, работающие непосредственно с текстом и модулями исходной программы. Многие их

функции интегрированы с функциями текстовых редакторов исходных текстов, входящих в состав систем программирования.

Уделим здесь место *библиотекам подпрограмм как составной части систем программирования.*

Библиотеки подпрограмм составляют существенную часть систем программирования. Состав доступных библиотек подпрограмм и дружественность их пользовательского интерфейса во многом определяет возможности систем программирования и позиции этих систем программирования на рынке средств разработки программного обеспечения.

Библиотеки подпрограмм входили в состав средств разработки, начиная с самых ранних этапов их развития. Даже когда компиляторы еще представляли собой отдельные программные модули, они уже были связаны с соответствующими библиотеками, поскольку компиляция, так или иначе предусматривает связь программ со стандартными функциями исходного языка. Эти функции обязательно должны входить в состав библиотек.

С точки зрения системы программирования, библиотеки подпрограмм состоят из двух основных компонентов. Это собственно файл (или множество файлов библиотеки), содержащий объектный код или набор файлов описаний функций, программ, констант и переменных, составляющих библиотеку.

12.2. Приемы тестирования и отладки системного программного обеспечения.

Как мы выяснили, *отладка* программы — это процесс поиска и устранения ошибок[24] в программе, который производится после её выполнения на компьютере.

Отметим, что однажды полученная в результате компиляции целевая программа может в дальнейшем выполняться много раз с различными входными данными.

Далеко не всегда исходные программы корректны с точки зрения исходного языка. Более того, некорректные программы подаются на вход компилятору значительно чаще, чем корректные — таков уж современный процесс разработки программ. Поэтому крайне важной частью процесса компиляции является точная диагностика ошибок, допущенных во входной программе. Впрочем, это замечание верно не только для компиляторов, но и для интерпретаторов

Тестирование — это процесс проверки правильности работы всей программы или ее составных частей.

---

[24] ***Программная ошибка*** (*жаргонное* **баг**) — означает ошибку в программе или в системе, из-за которой программа выдает неожиданное поведение и, как следствие, результат. Большинство программных ошибок возникают из-за ошибок, допущенных разработчиками программы в её исходном коде, либо в её дизайне. Некоторые ошибки возникают из-за некорректной работы инструментов разработчика, например, из-за компилятора, вырабатывающего некорректный код.

С разработкой алгоритма, выполнением проектирования и кодирования программы создание программного продукта не заканчивается, кроме этого, необходимо обеспечить соответствие требованиям и спецификациям исходного задания. Многократно проводимые исследования показали, что чем раньше обнаруживаются те или иные несоответствия или ошибки, тем больше вероятность их исправления и ниже стоимость разработки программного продукта.

Современные технологии разработки программного обеспечения предусматривают раннее обнаружение ошибок за счет выполнения контроля результатов всех этапов и стадий разработки. На начальных этапах контроль осуществляют вручную или с использованием ***CASE***[25]-средств, на последних — процесс контроля принимает форму тестирования.

Тестирование — это процесс выполнения программы, задачей которого является выявление ошибок. Известно, что никакое тестирование не может доказать отсутствие ошибок в сложном программном обеспечении, поскольку выполнение полного тестирования становится невозможным, и всегда есть вероятность, что остались невыявленные ошибки. Соблюдение основных правил тестирования и научно обоснованный подбор тестов может уменьшить количество ошибок. Процесс разработки программного обеспечения согласно современной модели жизненного цикла предполагает три стадии тестирования:

- *автономное* тестирование компонентов программного обеспечения;
- *комплексное* тестирование разрабатываемого программного обеспечения;
- системное или *оценочное* тестирование на соответствие основным критериям качества.

Для повышения качества тестирования рекомендуется соблюдать следующие основные принципы:

1. предполагаемые результаты должны быть известны до тестирования;
2. следует избегать тестирования программы автором;
3. необходимо досконально изучать результаты каждого теста;
4. необходимо проверять действия программы на неверных данных;
5. необходимо проверять программу на неожиданные побочные эффекты на неверных данных.

Справедливо утверждение, что вероятность наличия необнаруженных ошибок в части программы пропорциональна количеству ошибок уже найденных в этой части. Удачным считают тест, который обнаруживает *хотя*

[25] ***CASE*** (англ. *computer-aided software engineering*) — набор инструментов и методов программной инженерии для проектирования программного обеспечения, который помогает обеспечить высокое качество программ, отсутствие ошибок и простоту в обслуживании программных продуктов. Под термином ***CASE*** также понимают совокупность методов и средств проектирования информационных систем с использованием ***CASE***-инструментов.

бы одну ошибку. Формирование набора тестов имеет большое значение, потому что тестирование является одним из наиболее трудоемких этапов создания программного обеспечения. Доля стоимости тестирования в общей стоимости разработки возрастает при увеличении сложности программного обеспечения и повышении требований к их качеству.

Существуют два принципиально различных подхода к формированию тестовых наборов: *структурный* и *функциональный*.

Структурный подход базируется на том, что *известна* структура тестируемого программного обеспечения, в том числе его алгоритмы («*стеклянный ящик*»). Тесты строятся для проверки правильности реализации *заданной логики* в коде программы.

Функциональный подход основывается на том, что структура программного обеспечения *не известна* («*черный ящик*»). В этом случае тесты строят, опираясь на функциональные спецификации. Этот подход называют также *подходом, управляемым данными,* так как при его использовании тесты строят на базе различных способов декомпозиции множества данных. Наборы тестов, полученные в соответствии с методами этих подходов, объединяют, обеспечивая всестороннее тестирование программного обеспечения.

Заметим, что *функциональными спецификациями* в системной инженерии и разработке программного обеспечения называют комплект документов, описывающих требуемые характеристики разрабатываемой или тестируемой системы. Спецификации

- помогают устранить дублирование и несоответствия,
- позволяют точно оценить необходимые действия и ресурсы,
- выступают в качестве согласующего и справочного документов о внесённых изменениях,
- предоставляют документацию, описывающую особенности проблемы.

Документация описывает необходимые для пользователя системы входные и выходные параметры. Более подробно — это документ, который понятно и точно описывает существенные технические требования для объектов, процессов или операций.

Функциональные спецификации дают точное представление о решении проблемы, повышая эффективность разработки системы и оценивая стоимость альтернативных путей проектирования. Они служат указанием для испытателей для верификации (качественной оценки) каждого технического требования.

При этом функциональная спецификация не определяет операции, происходящие внутри данной системы и каким образом будет реализована её функция. Вместо этого, она рассматривает взаимодействие с внешними агентами (например, персонал, использующий программное обеспечение; периферийные устройства компьютера или другие компьютеры), которые могут взаимодействовать с системой.

Ручной контроль используют на ранних этапах разработки. Все проектные решения анализируются с точки зрения их *правильности и целесообразности* как можно раньше, пока их можно легко пересмотреть. Различают *статический* и *динамический* подходы к ручному контролю.

При *статическом* подходе анализируют структуру, управляющие и информационные связи программы, ее входные и выходные данные. При *динамическом* — выполняют *ручное тестирование* (вручную моделируют процесс выполнения программы на заданных исходных данных). Исходными данными для таких проверок являются: техническое задание, спецификации, структурная и функциональная схемы программного продукта, схемы отдельных компонентов, а для более поздних этапов — алгоритмы и тексты программ, а также тестовые наборы. Доказано, что ручной контроль способствует существенному увеличению производительности и повышению надежности программ и с его помощью можно находить от 30 до 70 % ошибок логического проектирования и кодирования. Основными методами ручного контроля являются:

— *инспекции исходного текста*,
— *сквозные просмотры*,
— *проверка за столом*,
— *оценки программ*.

В основе *структурного тестирования* лежит концепция максимально полного тестирования всех *путей*, предусмотренных алгоритмом (последовательности операторов программы, выполняемых при конкретном варианте исходных данных).

Недостатки метода структурного тестирования:

— построенные тестовые наборы не обнаруживают пропущенных маршрутов и ошибок, зависящих от заложенных данных;
— не дают гарантии, что программа правильна.

Другим способом проверки программ является *функциональное тестирование*: программа рассматривается как «*черный ящик*», то есть целью тестирования является выяснение обстоятельств, когда поведение программы *не соответствует* спецификации. Для обнаружения всех ошибок необходимо выполнить *исчерпывающее* тестирование (при всех возможных наборах данных), что для большинства случаев невозможно. Поэтому обычно выполняют «*разумное*» или «*приемлемое*» тестирование, ограничивающееся прогонами программы на небольшом подмножестве всех возможных входных данных. При функциональном тестировании различают следующие методы формирования тестовых наборов:

— *эквивалентное разбиение*;
— *анализ граничных значений*;

— *анализ причинно-следственных связей (*рассмотрение путей при реализации условных операторов);
— *предположение об ошибке*.

При *комплексном тестировании* используют тесты, построенные по методам эквивалентных классов, граничных условий и предположении об ошибках, поскольку структурное тестирование для него не применимо.

Одним из самых сложных является вопрос о завершении тестирования, так как невозможно гарантировать, что в программе не осталось ошибок.

Часто тестирование завершают потому, что закончилось время, отведенное на его выполнение. Его сворачивают, обходясь *минимальным тестированием*, которое предполагает:

— *тестирование* граничных значений,
— тщательную проверку руководства,
— тестирование минимальных конфигураций технических средств, возможности редактирования команд и
— повторения их в любой последовательности,
— устойчивости к ошибкам пользователя.

После завершения *комплексного тестирования* приступают к *оценочному тестированию*, цель которого — поиск несоответствий техническому заданию. Оценочное тестирование включает тестирование:

— удобства использования, в том числе, на предельных объемах, на предельных нагрузках,
— удобства эксплуатации, защиты, производительности, требований к памяти, конфигурации оборудования, совместимости,
— удобства установки,
— удобства обслуживания, надежности, восстановления, использования документации, процедуры сопровождения.

Отладка — это процесс *локализации* (определения *оператора* программы, выполнение которого вызвало нарушение вычислительного процесса) и исправления ошибок, обнаруженных при тестировании программного обеспечения.

Для исправления ошибки необходимо определить ее причину. Отладка требует от программиста глубоких знаний специфики управления используемыми техническими средствами, операционной системы, среды и языка программирования, реализуемых процессов, природы и специфики ошибок, методик отладки и соответствующих программных средств.

Отладка психологически дискомфортна (нужно искать собственные ошибки в условиях ограниченного времени); оставляет возможность взаимовлияния ошибок в разных частях программы. При этом четко сформулированные методики отладки отсутствуют.

Различают:

1. *синтаксические ошибки, которые* сопровождаются комментарием с указанием их местоположения, фиксируются компилятором (транслятором) при выполнении синтаксического и частично семантического анализа;
2. *ошибки компоновки*, которые обнаруживаются компоновщиком (редактором связей) при объединении модулей программы;
3. *ошибки выполнения, которые* обнаруживаются аппаратными средствами, операционной системой или пользователем при выполнении программы, проявляются разными способами и, в свою очередь, делятся на группы:
   - ошибки *определения исходных данных* (ошибки передачи, ошибки преобразования, ошибки перезаписи и ошибки данных);
   - *логические* ошибки *проектирования*
     - •неприменимый метод,
     - •неверный алгоритм,
     - •неверная структура данных,
     - •другие
   - *логические* ошибки *кодирования*
     - •ошибки некорректного использования переменных,
     - •ошибки некорректного использования вычислений,
     - •межмодульного интерфейса,
     - •реализации алгоритма,
     - •другие;
   - ошибки *накопления погрешностей* результатов вычислений
     - •игнорирование ограничений разрядной сетки и
     - •способов уменьшения погрешности).

Отладка программы в любом случае предполагает *обдумывание и логическое осмысление* всей имеющейся информации об ошибке.

Большинство ошибок можно обнаружить по косвенным признакам посредством тщательного анализа текстов программ и результатов тестирования без получения дополнительной информации с помощью следующих методов:

- *ручного тестирования* (при обнаружении ошибки нужно выполнить тестируемую программу вручную, используя тестовый набор, при работе с которым была обнаружена ошибка);
- *индукции* (основан на тщательном анализе симптомов ошибки, которые могут проявляться как неверные результаты вычислений или как сообщение об ошибке);
- *дедукции* (вначале формируют множество причин, которые могли бы вызвать данное проявление ошибки, а затем анализируя причины, *исключают* те, которые противоречат имеющимся данным);

— *обратного прослеживания* (для точки вывода неверного результата строится гипотеза о значениях основных переменных, которые могли бы привести к получению *данного* результата, а затем, исходя из этой гипотезы, делают предположения о значениях переменных в предыдущей точке).

Для получения дополнительной информации об ошибке выполняют добавочные тесты и используют специальные методы и средства:

— *отладочный вывод*;
— *интегрированные средства отладки*;
— *независимые отладчики*.

Например, общая методика *отладки* программных продуктов, написанных для выполнения в операционных системах *MS DOS* и *Win32*:

— *1 этап* — изучение проявления ошибки;
— *2 этап* — определение локализации ошибки;
— *3 этап* — определение причины ошибки;
— *4 этап* — исправление ошибки;
— *5 этап* — повторное тестирование.

Процесс *отладки* можно существенно упростить, если следовать основным рекомендациям *структурного подхода* к программированию:

— программу наращивать «сверху-вниз», от интерфейса к обрабатывающим подпрограммам, тестируя ее по ходу добавления подпрограмм;
— выводить пользователю вводимые им данные для контроля и проверять их на допустимость сразу после ввода;
— предусматривать вывод основных данных во всех узловых точках алгоритма (ветвлениях, вызовах подпрограмм).

12.3. Методы оптимизации программ

Оптимизацию можно выполнять на любой стадии генерации кода, начиная от завершения синтаксического разбора и вплоть до последнего этапа, когда порождается код результирующей программы. Если компилятор использует несколько различных форм внутреннего представления программы, то каждая из них может быть подвергнута оптимизации, причем различные формы внутреннего представления ориентированы на различные методы оптимизации. Таким образом, оптимизация в компиляторе может выполняться несколько раз на этапе генерации кода.

Принципиально различаются два основных вида оптимизирующих преобразований:

— преобразования исходной программы (в форме ее внутреннего представления в компиляторе), не зависящие от результирующего объектного языка;
— преобразования результирующей объектной программы.

Первый вид преобразований *не зависит от архитектуры* целевой вычислительной системы, на которой будет выполняться результирующая программа. Обычно он основан на выполнении хорошо известных и обоснованных математических и логических преобразований, производимых над внутренним представлением программы.

Второй вид преобразований *может зависеть* не только от свойств объектного языка, но и от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. Так, например, при оптимизации может учитываться объем кэш-памяти и методы организации конвейерных операций центрального процессора. В большинстве случаев эти преобразования сильно зависят от реализации компилятора и являются «ноу-хау» производителей компилятора. Но, именно этот тип оптимизирующих преобразований позволяет существенно повысить эффективность результирующего кода.

У современных компиляторов существуют возможности выбора не только общего критерия оптимизации, но и отдельных методов, которые будут использоваться при выполнении оптимизации.

Методы преобразования программы зависят от типов синтаксических конструкций исходного языка. Теоретически разработаны методы оптимизации для многих типовых конструкций языков программирования.

Оптимизация может выполняться для следующих типовых синтаксических конструкций:

- линейных участков программы;
- логических выражений;
- вызовов процедур и функций;
- других конструкций входного языка.

Во всех случаях могут использоваться как машинно-зависимые, так и машинно-независимые методы оптимизации.

Оптимизация линейных участков программ

Линейный участок программы — это выполняемая по порядку последовательность операций, имеющая один вход и один выход. Чаще всего линейный участок содержит последовательность вычислений, состоящих из арифметических операций и операторов присваивания значений переменным.

Для операций, составляющих линейный участок программы, могут применяться следующие виды оптимизирующих преобразований:

- удаление бесполезных присваиваний;
- исключение избыточных вычислений (лишних операций);
- свертка операций объектного кода;
- перестановка операций;
- арифметические преобразования.

Удаление бесполезных присваиваний заключается в том, что если в составе линейного участка программы имеется операция присваивания значения некоторой произвольной переменной A с номером ***i***, а также операция присваивания значения той же переменной ***A*** с номером ***j, i < j*** и ни в одной операции между ***i*** и ***j*** не используется значение переменной ***A***, то операция присваивания значения с номером ***i*** является бесполезной. Фактически бесполезная операция присваивания значения дает переменной значение, которое нигде не используется. Такая операция может быть исключена без ущерба для смысла программы.

В ряде случаев на этапе семантического анализа можно сделать простейшие вычисления, например, ***4*1024*** или же убрать из дальнейшего рассмотрения конструкции типа ***if (0)***.

Это полезно делать, так как компиляция даже маленькой программы компилятором, агрессивно выполняющим оптимизации, требует памяти порядка 300 Кб, большие программы могут потребовать 10-20Мб. Логика выполнения компилятора трудно предсказуема (предсказание переходов не работает примерно в 10% случаев), кэш-память в силу принципиальных особенностей обработки данных в компиляторе не влияет на увеличение производительности так сильно, как в других программах.

В общем случае, бесполезными могут оказаться не только операции присваивания, но и любые другие операции линейного участка, результат выполнения которых нигде не используется.

Например, во фрагменте программы

```
A = B * C;
D = B + C;
A = D * C;
```

операция присваивания ***A = B * C***; является бесполезной и может быть удалена. Вместе с удалением операции присваивания здесь может быть удалена и операция умножения, которая в результате также окажется бесполезной.

Обнаружение бесполезных операций присваивания далеко не всегда столь очевидно, как было показано в примере выше. Проблемы могут возникнуть, если между двумя операциями присваивания в линейном участке выполняются действия над указателями (адресами памяти) или выполняются вызовы функций, имеющих, так называемый, «побочный эффект».

Например, в следующем фрагменте программы

```
P = &A;
A = B * C;
D = P* + C;
A = D * C;
```

операция присвоения ***A = B * C***; уже не является бесполезной, хотя это и не столь очевидно. В этом случае неверно следовать простому принципу о том, что если переменная, которой присвоено значение в операции с

номером ***i***, не встречается ни в одной операции между ***i*** и ***j***, то операция присваивания с номером ***i*** является бесполезной.

Исключение избыточных вычислений (лишних операций) заключается в нахождении и удалении из объектного кода операций, которые повторно обрабатывают одни и те же операнды. Операция линейного участка с порядковым номером ***i*** считается лишней, если существует идентичная ей операция с порядковым номером ***j, j < i*** и никакой операнд, обрабатываемый этой операцией, не изменялся никакой операцией, имеющей порядковый номер между ***i*** и ***j***.

Свертка объектного кода — это выполнение во время компиляции тех операций исходной программы, для которых значения операндов уже известны. Тривиальным примером свертки является вычисление выражений, все операнды которых являются константами.

Например, выражение ***A = 2 * B * C * 3***; может быть преобразовано к виду ***A = 6 * B * C***.

Более сложные варианты алгоритма свертки принимают во внимание также и переменные, и функции, значения для которых уже известны.

Хорошим стилем программирования является объединение вместе операций, производимых над константами, чтобы облегчить компилятору выполнение свертки.

Перестановка операций заключается в изменении порядка следования операций, которое может повысить эффективность программы, но не будет влиять на конечный результат вычислений.

Например, операции умножения в выражении ***A = 2 * B * 3 * C***; можно переставить без изменения конечного результата и выполнить в следующем порядке ***A = (2 * 3) * (B * C)***. Тогда представляется возможным выполнить свертку и сократить количество операций.

Арифметические преобразования представляют собой выполнение изменения характера и порядка следования операций на основании известных алгебраических и логических тождеств.

Например, выражение ***A = B * C + B * D*** может быть заменено на ***A = B * (C + D)***. Конечный результат при этом не изменится, но объектный код будет содержать на одну операцию умножения меньше.

К арифметическим преобразованиям можно отнести и такие действия, как замена

- операции возведения в степень на операцию умножения;
- операции целочисленного умножения на операцию умножения на константу,
- операции целочисленного умножения, кратного 2, на выполнение операций сдвига.

Во всех этих случаях удается повысить быстродействие программы заменой сложных операций на более простые.

12.4. Основные принципы и методы оптимизации кода.

Общие принципы оптимизации кода

В подавляющем большинстве случаев генерация кода выполняется компилятором не для всей исходной программы в целом, а последовательно для отдельных ее конструкций. Для построения результирующего кода различных синтаксических конструкций входного языка используется метод СУ-перевода (синтаксически-управляемого перевода). Он объединяет цепочки построенного кода по структуре дерева без учета их взаимосвязей.

Построенный таким образом код результирующей программы может содержать лишние команды и данные. Это снижает эффективность выполнения результирующей программы. В принципе, компилятор может завершить на этом генерацию кода, поскольку результирующая программа построена и является эквивалентной по смыслу (семантике) программе на входном языке. Однако эффективность результирующей программы важна для ее разработчика, поэтому большинство современных компиляторов выполняют еще один этап компиляции — оптимизацию результирующей программы (или просто *оптимизацию*), чтобы повысить ее эффективность, насколько это возможно.

Важно отметить два момента.

Во-первых, выделение оптимизации в отдельный этап генерации кода — это вынужденный шаг. Компилятор вынужден производить оптимизацию построенного кода, поскольку он не может выполнить семантический анализ всей входной программы в целом, оценить ее смысл и, исходя из него, построить результирующую программу.

Во-вторых, оптимизация — это необязательный этап компиляции. Компилятор может вообще не выполнять оптимизацию, и при этом результирующая программа будет правильной, а сам компилятор будет полностью выполнять свои функции. Однако практически все современные компиляторы, так или иначе, выполняют оптимизацию, поскольку их разработчики стремятся завоевать хорошие позиции на рынке средств разработки программного обеспечения.

Приведем определение понятия «*оптимизация*».

Оптимизация программы — это обработка, связанная с переупорядочиванием и изменением операций в компилируемой программе с целью получения более эффективной результирующей объектной программы. Оптимизация выполняется на этапах подготовки к генерации и непосредственно при генерации объектного кода.

В качестве показателей эффективности результирующей программы можно использовать два критерия:

- объем памяти, необходимый для выполнения результирующей программы, и
- скорость выполнения (быстродействие) программы.

Далеко не всегда удается выполнить оптимизацию так, чтобы она удовлетворяла обоим этим критериям. Зачастую сокращение необходимого программе объема данных ведет к уменьшению ее быстродействия, и наоборот. Поэтому для оптимизации обычно выбирается один из упомянутых критериев или баланс между обоими. Выбор критерия оптимизации обычно выполняется в настройках компилятора.

Но даже выбрав критерий оптимизации, в общем случае, практически невозможно построить код результирующей программы, который бы являлся самым коротким или самым быстрым кодом, соответствующим входной программе. Дело в том, что нет алгоритмического способа нахождения самой короткой или самой быстрой результирующей программы, эквивалентной заданной исходной программе. Эта задача в принципе неразрешима.

Существуют алгоритмы, которые можно ускорять сколь угодно много раз для большого числа возможных входных данных, и, при этом, для других наборов входных данных они окажутся неоптимальными. К тому же компилятор обладает весьма ограниченными средствами анализа семантики всей входной программы в целом. Все, что можно сделать на этапе оптимизации, — это выполнить над заданной программой последовательность преобразований в надежде сделать ее более эффективной.

Чтобы оценить эффективность результирующей программы, полученной с помощью того или иного компилятора, часто прибегают к сравнению ее с эквивалентной программой (то есть с программой, реализующей тот же алгоритм), полученной из исходной программы, но написанной на языке ассемблера.

Лучшие оптимизирующие компиляторы могут получать результирующие объектные программы из сложных исходных программ, написанных на языках высокого уровня, почти не уступающие по качеству программам на языке ассемблера. Обычно соотношение эффективности программ, построенных с помощью компиляторов с языков высокого уровня, и программ, построенных с помощью ассемблера, составляет 1,1–1,3. То есть объектная программа, построенная с помощью компилятора с языка высокого уровня, обычно содержит на 10–30 % больше команд, чем эквивалентная ей объектная программа, построенная с помощью ассемблера, а значит, выполняется на 10–30 % медленнее.

Это весьма неплохие результаты, достигнутые компиляторами с языков высокого уровня, если сравнить трудозатраты на разработку программ на языке ассемблера и языке высокого уровня. Далеко не каждую программу можно реализовать на языке ассемблера в приемлемые сроки (а значит, и выполнить напрямую приведенное выше сравнение можно только для узкого круга программ).

Оптимизацию можно выполнять на любой стадии генерации кода, начиная от завершения синтаксического разбора и вплоть до последнего этапа, когда порождается код результирующей программы. Если компилятор использует несколько различных форм внутреннего представления программы, то каждая из них может быть подвергнута оптимизации, причем различные формы внутреннего представления ориентированы на различные методы оптимизации. Таким образом, оптимизация в компиляторе может выполняться несколько раз на этапе генерации кода.

Принципиально различаются два основных вида оптимизирующих преобразований:

— преобразования *исходной программы* (в форме ее внутреннего представления в компиляторе), не зависящие от результирующего объектного языка;
— преобразования *результирующей объектной программы*.

Первый вид преобразований не зависит от архитектуры целевой вычислительной системы, на которой будет выполняться результирующая программа. Обычно он основан на выполнении хорошо известных и обоснованных математических и логических преобразований, производимых над внутренним представлением программы.

Второй вид преобразований может зависеть не только от свойств объектного языка (что очевидно), но и от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. Так, например, при оптимизации может учитываться объем кэш-памяти и методы организации конвейерных операций центрального процессора. В большинстве случаев эти преобразования сильно зависят от реализации компилятора и являются «ноу-хау» производителей компилятора. Именно этот тип оптимизирующих преобразований позволяет существенно повысить эффективность результирующего кода.

Используемые методы оптимизации ни при каких условиях не должны приводить к изменению «смысла» исходной программы (то есть к таким ситуациям, когда результат выполнения программы изменяется после ее оптимизации).

Для преобразований первого вида проблем обычно не возникает.

Преобразования второго вида могут вызывать сложности, поскольку не все методы оптимизации, используемые создателями компиляторов, могут быть теоретически обоснованы и доказаны для всех возможных видов исходных программ. Именно эти преобразования могут повлиять на смысл исходной программы. Поэтому у современных компиляторов существуют возможности выбора не только общего критерия оптимизации, но и отдельных методов, которые будут использоваться при выполнении оптимизации.

Нередко оптимизация ведет к тому, что смысл программы оказывается не совсем таким, каким его ожидал увидеть разработчик программы, но это не из-за наличия ошибки в оптимизирующей части компилятора, а потому, что пользователь не принимал во внимание некоторые аспекты программы, связанные с оптимизацией.

Например, компилятор может исключить из программы вызов некоторой функции с заранее известным результатом, но, если эта функция имела «побочный эффект» — изменяла некоторые значения в глобальной памяти — смысл программы может измениться. Чаще всего это говорит о плохом стиле программирования исходной программы. Такие ошибки трудноуловимы, составляют проблемы для их нахождения.

Замечание. Разработчику программы следует обратить внимание на предупреждения, выдаваемые семантическим анализатором, или отключить оптимизацию. Применение оптимизации также *нецелесообразно* в процессе *отладки исходной* программы.

Методы преобразования программы зависят от типов синтаксических конструкций исходного языка. Теоретически разработаны методы оптимизации для многих типовых конструкций языков программирования.

Вопросы для контроля

1. Для чего предназначены такие системные средства как компоновщик, загрузчик, отладчик?

2. Опишите процесс формирования исполняемой программы компоновщиком. Как используется относительная адресация?

3. В чем разница между процессом отладки и процессом тестирования

4. Какие виды подходов к формированию текстовых наборов существуют? В чём их особенности.

5. Какие ошибки следует различать при отладке программы?

Глава 13. Исследование методов оптимизации программ

Учебные цели:

- *дать представление о методах генерации кода;*
- *познакомить с приемами оптимизации вычислений;*
- *разобрать пример оптимизации основных операторов.*

Сконцентрируем наши познания.

Процесс компиляции, как известно, состоит из двух основных частей: анализа (*analysis*) и синтеза (*synthesis*). Анализирующая часть компилятора разбивает исходную программу на составляющие ее элементы (конструкции языка — *language constructions*) и создает промежуточное представление исходной программы. Синтезирующая часть из промежуточного представления создает новую, целевую, программу.

На разных этапах фазы анализа может быть проведена локальная или глобальная оптимизация программы, и затем полученный код переведен в одно из внутренних представлений. Это делается для целей оптимизации и/или удобства генерации кода. Важно отметить, что все эти преобразования машинно-независимы.

Простейшая форма промежуточного представления — синтаксическое дерево программы (главы 6, 9, 11).

Для эффективной работы в фазе генерации в качестве внутреннего представления могут использоваться *префиксная или постфиксная польская запись*, ориентированный граф, тройки, четверки и тому подобные формы.

Так как сложность выполнения оптимизаций обычно пропорциональна квадрату или кубу от величины внутреннего представления программы, лучше явные случаи оптимизации делать на ранних этапах, чем в общем потоке.

Еще одна цель преобразования программы во внутреннее представление — желание иметь *переносимый кодификатор*. В этом случае только последняя фаза компиляции (*генерация кода*) является *машинно-зависимой*.

13.1. Методы генерации кода (тройки, четверки, обратная польская запись).

Генерация кода — последняя фаза трансляции.

Результатом её является либо ассемблерный модуль, либо объектный (либо загрузочный) модуль.

Код генерируется при обходе дерева, построенного анализатором. Обычно в современных трансляторах генерация кода осуществляется параллельно с построением дерева, но может осуществляться и как отдельный проход.

В процессе трансляции программы используется промежуточное представление программы, предназначенное для эффективной генерации

кода и проведения различных оптимизаций программы. Обычно промежуточный код получается разбиением сложной структуры исходного языка на более удобные для обращения элементы.

Сама форма промежуточного представления зависит от целей его использования. Наиболее часто используемыми формами промежуточного представления является ориентированный граф, тройки, четверки, префиксная и постфиксная запись.

В процессе г*енерации кода* могут выполняться некоторые *локальные оптимизации*, такие, как:

— распределение ресурсов;
— выбор длинных или коротких переходов;
— учет стоимости команд при выборе конкретной последовательности команд.

Для *генерации кода* разработаны различные *методы* такие, как

— таблицы решений,
— сопоставление образцов, включающее динамическое программирование,
— различные синтаксические методы.

Генерация кода осуществляется в два этапа:

1. генерация не зависящего от машины промежуточного кода;
2. генерация машинного кода для конкретной ЭВМ.

Во многих компиляторах оба эти процесса осуществляются за один проход.

В главах 6, 9 и 11 были подробно рассмотрены примеры, касающиеся внутреннего представления в виде ориентированного графа, деревьев, а в главе 11 — примеры внутреннего представления в виде обратной польской записи. Заметим, что как выяснялось в этих главах, обе эти формы промежуточного кода машинно-независимы. В этой главе уделим внимание другим важным внутренним представлениям, используемым при генерации кодов.

Одним из распространенных видов промежуточного кода является четверки или тетрады.

Тетрады или *трёхадресный код с явно именуемым результатом*.

Тетрады представляют собой линейную последовательность команд, а именно, запись операций в форме из четырех составляющих: операция, два операнда и результат операции. Например, тетрады могут выглядеть так*:*

операция (операнд_1, операнд_2, результат_операции).

При вычислении выражения, записанного в форме тетрад, тетрады вычисляются одна за другой последовательно. Каждая тетрада в последовательности вычисляется так: операция, заданная тетрадой, выполняется над операндами, и результат ее выполнения помещается в переменную, заданную результатом тетрады. Если какой-то из операндов (или оба операнда) в тетраде отсутствует (например, если тетрада представляет

собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи и ее реализации).

Результат же вычисления тетрады никогда опущен быть не может, иначе тетрада полностью теряет смысл. Порядок вычисления тетрад может быть изменен, но только если допустить наличие тетрад, целенаправленно изменяющих этот порядок (например, тетрады, вызывающие переход на несколько шагов вперед или назад при каком-то условии).

Отметим важную особенность: так как тетрады представляют собой линейную последовательность, для них несложно написать тривиальный алгоритм, который будет преобразовывать последовательность тетрад в последовательность команд результирующей программы (либо последовательность команд ассемблера). В этом их преимущество перед синтаксическими деревьями. А в отличие от команд ассемблера тетрады не зависят от архитектуры вычислительной системы, на которую ориентирована результирующая программа. Поэтому они представляют собой *машинно-независимую форму* внутреннего представления программы.

Тетрады требуют больше памяти для своего представления, чем триады (которые рассмотрим ниже), и также, как и триады, не отражают явно взаимосвязь операций между собой. Кроме того, есть сложности с преобразованием тетрад в машинный код, так как они плохо отображаются в команды ассемблера и машинные коды, поскольку в наборах команд большинства современных компьютеров редко встречаются операции с тремя операндами.

Трехадресный код (тетрады) представляет собой последовательность инструкций вида

x = y op z

где ***x***, ***y*** и ***z*** — имена, константы или вре́менные переменные, генерируемые компилятором; ***op*** означает некоторый оператор, например, арифметический оператор для работы с числами с фиксированной или «плавающей» точкой или оператор для работы с логическими значениями. Например, выражение исходного языка наподобие ***x + y * z*** может быть транслировано в следующую последовательность.

```
t1 = y * z;
t2 = x + t1;
```

Здесь ***t1*** и ***t2*** — сгенерированные компилятором *вре́менные* имена. Использование имен для вычисленных программой промежуточных значений обеспечивает трехадресному коду, в отличие от постфиксной записи, возможность легкого переупорядочения.

Термин «трехадресный код» отражает тот факт, что каждая инструкция обычно содержит три адреса — два для операндов и один для результата.

Приведём список некоторых основных трехадресных инструкций, используемых в большинстве языков программирования:

1. Инструкции присваивания вида ***x = y op z,*** где ***op*** — бинарная арифметическая или логическая операция.
2. Инструкция присваивания вида ***x = op y,*** где ***op*** — унарная операция. Основные унарные операции включают унарный минус, логическое отрицание, операторы сдвига и операторы преобразования, которые, например, преобразуют число с фиксированной точкой в число с плавающей точкой.
3. Инструкции копирования вида ***x = y***, в которых значение ***y*** присваивается ***x***.
4. Безусловный переход ***goto L.*** После этой инструкции будет выполнена трехадресная инструкция с меткой ***L*** (переход на инструкцию, перед которой стоит метка ***L***).
5. Условный переход типа ***if x relop y goto L***. Эта инструкция применяет оператор отношения ***relop*** (<, >= и тому подобное) к ***x*** и ***y***, и следующей выполняется инструкция с меткой ***L***, если соотношение ***x relop y*** верно. В противном случае выполняется следующая за условным переходом инструкция.
6. Индексированные присвоения типа ***x = y[i]; x[i] = y;.*** Первая инструкция присваивает ***x*** значение, находящееся в ***i***-й ячейке памяти по отношению к ***y***. Инструкция ***x[i] = y*** заносит в ***i***-ю ячейку памяти по отношению к ***x*** значение ***y***. В обеих инструкциях ***x, y*** и ***i*** ссылаются на объекты данных.
7. Присвоение адресов и указателей вида ***x = &y, x = *y*** и ****x =y***. Первая инструкция устанавливает значение ***x*** равным положению ***y*** в памяти. Предположительно, ***y*** представляет собой имя, возможно временное, обозначающее выражение с ***l***-значением типа ***A [i][j]***, а ***x*** — имя указателя или временное имя. Таким образом, ***r***-значение ***x*** представляет собой значение некоторого объекта. Во второй инструкции под **y** подразумевается указатель или временная переменная, ***l***-значение которой представляет собой местоположение ячейки памяти. В результате ***l***-значение ***x*** становится равным содержимому этой ячейки. И наконец, инструкция ***x = y** устанавливает ***l***-значение объекта, указываемого ***x***, равным ***l***-значению ***y***.

Выбор приемлемых операторов, представляет собой важный вопрос в создании промежуточного представления. Очевидно, что множество операторов должно быть достаточно богатым, чтобы позволить реализовать все операции исходного языка. Небольшое множество операторов легче реализуется на *новой целевой* машине, однако ограниченное множество инструкций может привести к генерации длинных последовательностей инструкций промежуточного представления для некоторых конструкций исходного языка и добавить работы оптимизатору и генератору целевого кода.

Пример 13.1. Запись оператора присваивания алгебраического выражения в тетрадах

Например, выражение ***a = b*c + d - b*10***, записанное в виде тетрад, будет иметь вид:

1. * (b, c, t1)	***t1=b*c***
2. + (t1, d, t2)	***t2=t1+d***
3. * (b, 10, t3)	***t3= b*10***
4. – (t2, t3, t4)	***t4=t2-t3***
5. = (t4, 0, a)	***a=t4*** второй операнд не используется

Здесь все операции обозначены соответствующими знаками (при этом присваивание также является операцией). Идентификаторы ***t1, t2, t3, t4*** обозначают *вре́менные* переменные, используемые для хранения результатов вычисления тетрад. Следует обратить внимание на то, что в последней тетраде (операция присваивания), которая требует только одного операнда, согласно выше описанным правилам, в качестве второго операнда выступает незначащий операнд «***0***».

Рассмотрим еще один, несколько более сложный, пример, а именно, преобразование алгебраического выражения:

(-a+b) * (c+d).

Пример 13.2.

Это алгебраическое выражение можно представить в виде последовательности четверок следующим образом:

1. –(a, 0, t1)	***t1 =-a***
2. +(t1, b, t2)	***t2 = t1+b***
3. +(c, d, t3)	***t3 = c+d***
4. *(t2, t3, t4)	***t4 = t2 * t2***

Здесь вре́менные переменные ***t1, t2, t3, t4*** соответствуют идентификаторам, присвоенным компилятором. Четверки можно считать промежуточным кодом высокого уровня. Напомним, такой код называют трехадресным кодом с явно именуемым результатом, так как используются два адреса для операндов и один — для результата.

Трехадресный код с неявно именуемым результатом (триады).

Триады представляют собой запись операций в форме из трех составляющих: операция и два операнда. Например, триады могут иметь вид:

операция (операнд_1, операнд_2).

Особенностью триад является то, что один или оба операнда могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады. Поэтому триады при записи последовательно нумеруют для удобства указания ссылок одних триад на другие (в реализации компилятора в качестве ссылок можно использовать не номера триад, а непосредственно ссылки в виде указателей — тогда при изменении нумерации и порядка следования триад менять ссылки не требуется).

Триады представляют собой линейную последовательность команд, как и тетрады. При вычислении выражения, записанного в форме триад, они вычисляются одна за другой последовательно. Каждая триада в последовательности вычисляется так: операция, заданная триадой, выполняется над операндами, а если в качестве одного из операндов (или обоих операндов) выступает ссылка на другую триаду, то берется результат вычисления той триады. Результат вычисления триады нужно сохранять во временной памяти, так как он может быть затребован последующими триадами. Если какой-то из операндов в триаде отсутствует (например, если триада представляет собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи и ее реализации). Порядок вычисления триад, как и для тетрад, может быть изменен, но только если допустить наличие триад, целенаправленно изменяющих этот порядок (например, триады, вызывающие переход на несколько шагов вперед или назад при каком-то условии).

Триады, как и тетрады, как было уже сказано, представляют собой линейную последовательность, а потому для них также несложно написать тривиальный алгоритм, который будет преобразовывать последовательность триад в последовательность команд результирующей программы (либо последовательность команд ассемблера). В этом их преимущество перед синтаксическими деревьями. Однако здесь требуется также и алгоритм, отвечающий за распределение памяти, необходимой для хранения промежуточных результатов вычисления, так как временные переменные для этой цели не используются. В этом отличие триад от тетрад.

Так же, как и тетрады, триады не зависят от архитектуры вычислительной системы, на которую ориентирована результирующая программа. Поэтому они представляют собой машинно-независимую форму внутреннего представления программы.

Триады требуют меньше памяти для своего представления, чем тетрады, они также явно отражают взаимосвязь операций между собой, что делает их применение удобным. Необходимость иметь алгоритм, отвечающий за распределение памяти для хранения промежуточных результатов, не является недостатком, так как удобно распределять результаты не только по доступным ячейкам временной памяти, но и по имеющимся регистрам процессора. Это дает определенные преимущества. Триады ближе к двухадресным машинным командам, чем тетрады, а именно эти команды более всего распространены в наборах команд большинства современных компьютеров.

Пример 13.3. Запись оператора присваивания алгебраического выражения в триадах

Например, выражение ***a=b*c + d - b*10,*** записанное в виде триад, будет иметь вид:

1. * (b, c)	***r1= b*c***
2. + (&1, d)	***r2=r1+d***
3. * (b, 10)	***r3= b*10***
4. – (&2, &3)	***r4=r2-r3***
5. = (a, &4)	***a=r4***

Здесь операции обозначены соответствующим знаком (при этом присваивание также является операцией), а знак «***&***» означает ссылку операнда одной триады на результат другой.

Приведем еще один сравнительный пример преобразования алгебраического выражения в представление в виде четверок и троек.

Пример 13.4. Запись алгебраического выражения в виде четвёрок и троек

Выражение ***a + b + c * d*** можно представить в виде четверок:

1. +(a, b, t1)	***t1 = a + b***
2. *(c, d, t2)	***t2 = c * d***
3. +(t1, t2, t3)	***t3 = t1 + t2,***

где вре́менные переменные ***t1, t2, t3*** соответствуют идентификаторам, присвоенным компилятором.

И в виде троек:

1. +(a, b)	***r1 = a + b***
2. *(c, d)	***r2 = c * d***
3. +(r1, r2)	***r1 = r1 + r2,***

где для хранения результата используется позиция операнда (в данном случае регистр), так как сам операнд является тройкой.

Как тройки, так и четверки можно распространить не только на выражения, но и на другие конструкции языка.

Пример 13.5. Простой пример для оператора присваивания

Оператор присваивания ***a = b*** в виде четверки представляется как

1. =(b, 0, t1)	***t1 = b***
2. =(t1, 0, a)	***a = t1,***

а в виде тройки — как

1. =(b, t1)	***r1 = b***
2. =(t1, a)	***a = r1.***

Ассемблерный код и машинные команды.

Машинные команды удобны тем, что при их использовании внутреннее представление программы полностью соответствует объектному коду и сложные преобразования не требуются. Команды ассемблера представляют собой лишь форму записи машинных команд, а потому в качестве формы внутреннего представления программы практически ничем не отличаются от них.

Однако использование команд ассемблера или машинных команд для внутреннего представления программы требует дополнительных структур для отображения взаимосвязи операций. Очевидно, что в этом случае внутреннее представление программы получается зависимым от архитектуры вычислительной системы, на которую ориентирован результирующий код. Значит, при ориентации компилятора на другой результирующий код потребуется перестраивать как само внутреннее представление программы, так и методы его обработки (при использовании триад или тетрад этого не требуется).

Тем не менее, машинные команды — это язык, на котором должна быть записана результирующая программа. Поэтому компилятор, так или иначе, должен работать с ними. Кроме того, только обрабатывая машинные команды (или их представление в форме команд ассемблера), можно добиться наиболее эффективной результирующей программы. Отсюда следует, что любой компилятор работает с представлением результирующей программы в форме машинных команд, однако их обработка происходит, как правило, на завершающих этапах фазы генерации кода.

Приведем примеры представления алгебраических выражений в виде промежуточного кода, представленного префиксной и постфиксной нотацией. Записи промежуточного кода в таком виде не менее популярны, чем триады или тетрады.

В префиксной нотации каждый знак операции появляется перед своими операндами, а в постфиксной — после операндов.

Например, инфиксное выражение ***a + b*** в префиксной нотации имеет вид ***+ab***, а в постфиксной — ***ab+***. Префиксная нотация известна также как *польская* запись, а постфиксная — как *обратная польская* запись (запись Лукашевича).

Пример 13.6. Запись алгебраического выражения ***(a+b) * (c+d)*** в префиксной и постфиксной формах

Это выражение записывается следующим образом в

префиксной форме	*постфиксной форме*
****+ab+cd***	***ab+cd+****

В префиксной и постфиксной нотации скобки уже не употребляются, так как в такой записи никогда не возникает сомнения относительно того, какие операнды принадлежат тем или иным знакам операций. В этих нотациях не существует приоритета знака операции.

Перегруппировку в результате преобразования алгебраического выражения ***(a+b)*(c+d)*** в ***ab+cd+**** можно осуществить также с помощью *стека*. При этом алгоритм сведется к следующим трем действиям:

1. вывести в выходную строку идентификатор, когда он встретится при чтении инфиксного выражения слева направо;
2. поместить в стек знак операции, когда он встретится;

3. когда встретится конец выражения (или подвыражения), вывести в выходную строку знак операции, который находится на верхушке стека.

Префиксные и постфиксные выражения можно также получать из представления выражения в виде бинарного дерева (рис. 13.1).

Пример 13.7. Преобразование бинарного дерева в префиксное выражение

Рассмотрим представление алгебраического выражения ***(a+b)*c+d*** в виде бинарного дерева (рис. 13.1).

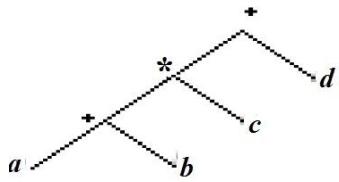


Рис. 13.1. Бинарное дерево алгебраического выражения ***(a+b)*c+d***

Чтобы получить представление префиксного выражения, дерево обходят сверху вниз в следующем порядке:

— посещение корня;
— обход левого поддерева сверху;
— обход правого поддерева сверху.

В результате обхода получим требуемое инфиксное выражение ***+*+abcd***.

13.2. Оптимизация вычисления логических выражений.

Особенность оптимизации логических выражений заключается в том, что не всегда необходимо полностью выполнять вычисление всего выражения для того, чтобы знать его результат. Иногда по результату первой операции или даже по значению одного операнда можно заранее определить результат вычисления всего выражения.

Операция называется предопределенной для некоторого значения операнда, если ее результат зависит только от этого операнда и остается неизменным (инвариантным) относительно значений других операндов.

Операция логического сложения (***or***) является предопределенной для логического значения «*истина*» (***true***), а операция логического умножения — предопределена для логического значения «*ложь*» (***false***).

Эти факты могут быть использованы для оптимизации логических выражений. Действительно, получив значение «*истина*» в последовательности логических сложений или значение «*ложь*» в последовательности логических умножений нет никакой необходимости далее производить вычисления — результат уже определен и известен.

Например, выражение ***A or B or C or D*** не имеет смысла вычислять, если известно, что значение переменной ***A*** есть ***True*** («*истина*»).

Компиляторы строят объектный код вычисления логических выражений таким образом, что вычисление выражения прекращается сразу же, как

только его значение становится предопределенным. Это позволяет ускорить вычисления при выполнении результирующей программы. В сочетании с преобразованиями логических выражений на основе тождеств булевой алгебры и перестановкой операций эффективность данного метода может быть несколько увеличена.

Не всегда такие преобразования инвариантны к смыслу программы. Например, при вычислении выражения ***A or F(B) or G(C)*** функции ***F*** и ***G*** не будут вызваны и выполнены, если значением переменной ***A*** является ***true***. Это не важно, если результатом этих функций является только возвращаемое ими значение, но если они обладают «побочным эффектом», то семантика программы может измениться.

Хорошим стилем считается также принимать во внимание эту особенность вычисления логических выражений. Тогда операнды в логических выражениях следует стремиться располагать таким образом, чтобы в первую очередь вычислялись те из них, которые чаще определяют все значение выражения. Кроме того, значения функций лучше вычислять в конце, а не в начале логического выражения, чтобы избежать лишних обращений к ним.

13.3. Оптимизация передачи параметров в процедуры и функции.

Существуют методы, позволяющие снизить затраты кода и времени выполнения на передачу параметров в процедуры и функции и повысить в результате эффективность результирующей программы:

— передача параметров через регистры процессора;
— подстановка кода функции в вызывающий объектный код.

Метод передачи параметров через регистры процессора позволяет разместить все или часть параметров, передаваемых в процедуру или функцию, непосредственно в регистрах процессора, а не в стеке. Это позволяет ускорить обработку параметров функции, поскольку работа с регистрами процессора всегда выполняется быстрее, чем с ячейками оперативной памяти, где располагается стек. Если все параметры удается разместить в регистрах, то сокращается также и объем кода, поскольку исключаются все операции со стеком при размещении в нем параметров.

Понятно, что регистры процессора, используемые для передачи параметров в процедуру или функцию, не могут быть задействованы напрямую для вычислений внутри этой процедуры или функции. Поскольку программно доступных регистров процессора всегда ограниченное количество, то при выполнении сложных вычислений могут возникнуть проблемы с их распределением. Тогда компилятор должен выбрать: либо использовать регистры для передачи параметров и снизить эффективность вычислений (часть промежуточных результатов, возможно, потребуется размещать в оперативной памяти или в том же стеке), либо использовать свободные регистры для вычислений и снизить эффективность передачи параметров.

Поэтому реализация данного метода зависит от количества программно доступных регистров процессора в целевой вычислительной системе и от используемого в компиляторе алгоритма распределения регистров. В современных процессорах число программно доступных регистров постоянно увеличивается с разработкой все новых и новых их вариантов, поэтому и значение этого метода постоянно возрастает.

Этот метод имеет ряд недостатков. Во-первых, очевидно, он зависит от архитектуры целевой вычислительной системы. Во-вторых, процедуры и функции, оптимизированные таким методом, не могут быть использованы в качестве процедур или функций библиотек подпрограмм ни в каком виде. Это вызвано тем, что методы передачи параметров через регистры процессора не стандартизованы (в отличие от методов передачи параметров через стек) и зависят от реализации компилятора. Наконец, этот метод не может быть использован, если где бы то ни было, в процедуре или функции, требуется выполнить операции с адресами (указателями) на параметры.

Метод подстановки кода функции в вызывающий объектный код (так называемая *inline*-подстановка) основан на том, что объектный код функции непосредственно включается в вызывающий объектный код всякий раз в месте вызова функции.

Для разработчика исходной программы такая функция (называемая *inline*-функцией) ничем не отличается от любой другой функции, но для вызова ее в результирующей программе используется принципиально другой механизм. По сути, вызов функции в результирующем объектном коде вовсе не выполняется: просто все вычисления, производимые функцией, выполняются непосредственно в самом вызывающем коде в месте ее вызова.

Очевидно, что в общем случае такой метод оптимизации ведет не только к увеличению скорости выполнения программы, но и к увеличению объема объектного кода. Скорость увеличивается за счет отказа от всех операций, связанных с вызовом функции: это не только сама команда вызова, но и все действия, связанные с передачей параметров. Вычисления при этом идут непосредственно с фактическими параметрами функции. Объем кода растет, так как приходится всякий раз включать код функции в место ее вызова. Тем не менее, если функция очень проста и включает в себя лишь несколько машинных команд, то можно даже добиться сокращения кода результирующей программы, так как при включении кода самой функции в место ее вызова оттуда исключаются операции, связанные с передачей ей параметров.

13.4. Оптимизации циклов.

Циклом в программе называется любая последовательность участков программы, которая может выполняться повторно.

Циклы обычно содержат в себе один или несколько линейных участков, где производятся вычисления. Поэтому методы оптимизации линейных

участков позволяют повысить также и эффективность выполнения циклов, причем они оказываются тем более эффективными, чем больше кратность выполнения цикла. Но есть методы оптимизации программ, специально ориентированные на оптимизацию циклов.

Для оптимизации циклов используются следующие методы:

—вынесение инвариантных вычислений из циклов;

—замена операций с индуктивными переменными;

—слияние и развертывание циклов.

Вынесение инвариантных вычислений из циклов заключается в вынесении за пределы циклов тех операций, операнды которых не изменяются в процессе выполнения цикла. Очевидно, что такие операции могут быть выполнены один раз до начала цикла, а полученные результаты потом могут использоваться в теле цикла.

Пример 13.8. Вынесение инвариантных вычислений из циклов

Например, цикл

```
for (i=1; i < 1; i++)
   { A[i] = B * C * A[i];
          …
    }
```

может быть заменен на последовательность операций

```
D = B * C;
for (i=1; i < 1; i++)
   { A[i] = D * A[i];
          …
    }
```

если значения ***B*** и ***C*** не изменяются нигде в теле цикла.

При этом операция умножения ***B*C*** будет выполнена только один раз, в то время как в первом варианте она выполнялась ***10*** раз над одними и теми же результатами.

Замена операций с индуктивными переменными заключается в изменении сложных операций с индуктивными переменными в теле цикла на более простые операции. Как правило, выполняется замена умножения на сложение.

Переменная называется *индуктивной* в цикле, если ее значения в процессе выполнения цикла образуют арифметическую прогрессию. Таких переменных в цикле может быть несколько, тогда в теле цикла их иногда можно заменить на одну-единственную переменную, а реальные значения для каждой переменной будут вычисляться с помощью соответствующих коэффициентов соотношения (за пределами цикла значения всем переменным должны быть присвоены, если, конечно, они используются).

Простейшей индуктивной переменной является переменная-счетчик цикла с перечислением значений (цикл типа ***for***, который встречается в

синтаксисе многих современных языков программирования). Более сложные случаи присутствия индуктивных переменных в цикле требуют специального анализа тела цикла. Не всегда выявление таких переменных является тривиальной задачей.

После того как индуктивные переменные выявлены, необходимо проанализировать те операции в теле цикла, где они используются. Часть таких операций может быть упрощена. Как правило, речь идет о замене умножения на сложение.

Пример 13.9. Замена в цикле операции умножения на операцию сложения

Например, цикл

```
S =10;
for (i = 1;i < N; i++) {A[i] = i*S};
```

может быть заменен на последовательность операций

```
S = 10;
T = S; i =1;
while (i < = 10)
      {A[i] = T;
        T =T + 10;
        i =i + 1;
      }
```

Здесь использован синтаксис языка Си, а ***T*** — это некоторая новая вре́менная переменная (использовать для той же цели уже существующую переменную ***S*** не вполне корректно, так как ее значение может быть использовано и после завершения этого цикла). В итоге удалось отказаться от выполнения ***N*** операций умножения, заменив их на ***N*** операций сложения (которые обычно выполняются быстрее). Индуктивной переменной в первом варианте цикла являлась ***i***, а во втором варианте — ***i*** и ***T***.

В современных реальных компиляторах такие преобразования используются достаточно редко, поскольку они требуют достаточно сложного анализа программы, в то время как достигаемый выигрыш невелик: разница в скорости выполнения сложения и умножения, равно как и многих других операций, в современных вычислительных системах не столь существенна. Кроме того, существуют варианты циклов, для которых эффективность указанных методов преобразования является спорной.

Слияние и развертывание циклов предусматривает два различных варианта преобразований: слияния двух вложенных циклов в один и замена цикла на линейную последовательность операций.

Слияние двух циклов можно проиллюстрировать на примере следующих циклов:

Пример 13.10. Слияние двух циклов (инициализация двумерного массива)

```
for (i = 1; i < N; i++)
   {for (j =l; j < M; j++) {A[i][j] = 0;}
   }
```

В этом фрагменте происходит инициализация двумерного массива. Но в объектном коде двумерный массив — это всего лишь область памяти размером ***N * M***, поэтому (с точки зрения объектного кода, но не входного языка!) эту операцию можно эффективнее представить так:

Пример 13.11. Слияние двух циклов (инициализация одномерного массива)

```
K = N * M;
for (i = l; i < K; i++)  {A[i] = 0};
```

Развертывание циклов можно выполнить для циклов, кратность выполнения которых *известна уже на этапе компиляции*. Тогда цикл, кратностью ***N***, можно заменить на линейную последовательность ***N*** операций, содержащих тело цикла.

Пример 13.12. Развертывание цикла, кратность выполнения которого известна заранее

Например, пусть в программе присутствует следующий цикл

```
for (i = 1; i < 3; i++0)   {A[i] = i};
```

Его можно заменить простыми операциями присваивания для инициализации:

```
A[1] = 1;
A[2] = 2;
A[3] = 3;
```

В результате замены программа станет работать эффективнее.

Вопросы для контроля

1. Какие методы подготовки к генерации кода, используемые в настоящее время, вам известны?
2. Какие методы подготовки к генерации кода являются машинно-независимыми и почему?
3. Чем отличаются процессы преобразования алгебраического выражения в трехадресный код с неявно именуемым результатом и в трехадресный код с явно именуемым результатом? Приведите примеры.
4. Чем отличаются префиксная и постфиксная записи алгебраического выражения? Приведите примеры.
5. Какие способы существуют для оптимизации циклов? Приведите примеры.

Глава 14. Исследование способов тестирования и отладки программного обеспечения

Учебные цели:

- дать *представление о работе с динамическими структурами и указателями;*
- *познакомить с видами тестирования и отладки программного обеспечения.*

14.1. Работа с динамическими структурами и указателями.

Динамические структуры данных — это структуры данных, память под которые выделяется и освобождается по мере необходимости в процессе исполнения программы.

Динамические структуры данных в процессе существования в памяти могут изменять не только число элементов, составляющих эти структуры, но и характер связей между элементами структуры. При этом не учитывается изменение содержимого самих элементов данных.

Особенность динамических структур, выражающаяся в непостоянстве их размера и взаимоотношений между элементами, приводит к тому, что на этапе создания машинного кода компилятор не может выделить для всей структуры целиков участок памяти фиксированного размера, а также не может сопоставить конкретные адреса с отдельными компонентами структуры (потому что адреса определяются только в процессе динамического отведения памяти под них).

Для решения проблемы адресации динамических структур данных используется метод, называемый *динамическим распределением памяти*, то есть память под отдельные элементы выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае выделяет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.

Динамическая структура данных характеризуется тем что:

- она не имеет имени;
- количество элементов структуры разрешается не фиксировать;
- в процессе выполнения программы
  - выделяется память для динамической структуры;
  - размерность структуры может меняться;
  - характер взаимосвязи между элементами структуры может меняться.

Для того, чтобы работать с некоторой динамической структурой данных, в программе для неё описывается статическая переменная типа указатель (значением этого указателя является адрес объекта). Как только

динамической структуре данных выделяется память, адрес начала этой памяти записывается в качестве значения соответствующего указателя. В дальнейшем доступ к динамической структуре будем осуществляться посредством обращения к указателю.

Как упоминалось выше, сами динамические величины не требуют описания в программе, так как во время компиляции память под них не выделяется (во время компиляции память выделяется только под статические величины). А вот указатели —это статические величины, поэтому они требуют описания.

В каких же случаях возникает необходимость использовать динамические структуры данных? Обычно потребность в таком распределении памяти возникает, если

- используются переменные, имеющие довольно большой размер (например, массивы большой размерности), необходимые в одних фрагментах программы и совершенно не нужные в других;
- в процессе работы программы требуется массив, список или иная структура, количество элементов, процесс обработки которой непредсказуем;
- размер данных, обрабатываемых в программе, превышает объем сегмента данных, выделенных фрагменту программ.

Динамические структуры, по определению, могут располагаться в физически несмежных участках памяти, запрашивать заранее неизвестный объём памяти (так как заранее неизвестно количество элементов структуры) в процессе обработки.

Ввиду того, что элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Поэтому для того, чтобы установить связь между элементами динамической структуры используются указатели.

Такое представление данных в памяти называется *связным*; оно позволяет обеспечить значительную изменчивость структуры.

Преимущества связного распределения динамической памяти в том, что:

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
- структура обладает большой гибкостью.

Недостатки у связной структуры тоже есть, основные из них следующие:

- указатели для связывания элементов друг с другом требуют дополнительную память;

— доступ к элементам связной структуры может быть менее эффективным по времени.

Последний недостаток — наиболее серьезный, и именно он ограничивает применимость связного представления данных.

При *смежном* представлении данных для вычисления адреса любого элемента достаточно знать номер элемента и информацию, содержащуюся в дескрипторе структуры.

Для *связного* представления адрес элемента нельзя вычислить из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, и далее, следуя по цепочке указателей от элемента к элементу, осуществить поиск требуемого элемента.

Поэтому связное представление практически *никогда не применяется* в задачах, где логическая структура данных представляется вектором или массивом, то есть, если можно осуществить доступ по номеру элемента.

И наоборот, связное динамическое представление часто применяется в задачах, где логическая структура исходной информации требует специального вида (таблицы неизвестного объёма, списки, деревья и тому подобное).

Последовательность работы с динамическими структурами данных такова:

1. создание структуры (отведение места) в динамической памяти;
2. выполнение различных операций над динамическими объектами, используя указатели;
3. удаление (освобождение) занятое структурой места.

Теперь обратимся к практической стороне работы с динамической памятью.

Напомним, что *указатель* – переменная, которая в качестве своего значения содержит адрес байта памяти. Указатель занимает 4 байта.

Как правило, указатель связывается с некоторым типом данных. В таком случае он называется *типизированным*. Для его объявления используется знак «*», который помещается перед соответствующим типом, например:

v* int; b*, c* float;

Указатели сами по себе представляют значения, которые можно хранить в массивах. То есть, оказывается, можно рассмотреть массив указателей.

Массив указателей определяется одним из трех способов:

тип *имя_массива [размер];
тип *имя_массива [] = инициализатор;
тип *имя_массива [размер] = инициализатор;

Приведём примеры:

int a[] = {1, 2, 3, 4}; //описан и инициализирован массив ***a***
int *p1[3]; // массив указателей ***p1*** состоит из трех элементов,
//не инициализирован и является пустым.
int *p2[] = { &a[1], &a[2], &a[0] }; //массивы ***p2*** и ***p3*** в качестве элементов
int *p3[3] = { &a[3], &a[1], &a[2] }; // хранят адреса на элементы массива ***a***

Кроме типизированных указателей в языке Си можно объявлять указатель, не связывая его с конкретным типом данных. Для этого служит *нетипизированный* указатель ***void****, например:

void* ptr;

Поскольку нетипизированные указатели не связаны с конкретным типом, с их помощью удобно динамически размещать данные, структура и *тип* которых меняются в ходе работы программы.

Так как значениями указателей являются адреса переменных памяти, следовало бы ожидать, что значение одного из них можно передавать другому. Однако это не совсем так. Операция передачи значения одного указателя другому может проводиться только среди указателей, связанных с одними и теми же *типами* данных.

Например, пусть объявлены указатели:

p1*, p*int; p3* float; pp* void;

В этом случае присваивание ***p1 = p2;*** допустимо, а вот ***p1 = p3;*** запрещено, так как ***p1*** и ***p3*** указывают на *разные* типы данных. Это ограничение не распространяется на нетипизированные указатели, поэтому можно записать ***pp = p3; p1 = pp;*** и достичь необходимого результата.

Вся динамическая память в современных языках программирования и, в частности, в Си/Си++ представляет собой сплошной массив байтов, называемый кучей. Физически куча располагается за областью памяти, которую занимает тело программы. Начало, конец и текущая граница незанятой динамической памяти кучи хранятся в стандартных переменных.

Память под любую динамическую переменную выделяется процедурой ***new*** в Си++ (или ***malloc, calloc, realloc*** в Си), параметром обращения к которой является *типизированный* указатель. В результате выполнения этой команды указатель принимает значение, соответствующее динамическому адресу, начиная с которого можно разместить данные.

Например, фрагмент программы выделения памяти может выглядеть следующим образом:

```
i, j *int;
R* float;
{new(i);
  new(R);
  new(j); …
```

В результате выполнения первого оператора указатель ***i*** принимает значение, которое перед этим имел указатель начала кучи. Сам указатель начала кучи увеличивает свое значение на ***4***, так как длина внутреннего представления типа ***integer***, связанного с указателем ***i***, составляет ***4*** байта.

Оператор ***new(R)*** вызывает еще одно смещение указателя начала кучи, но уже на ***8*** байтов, потому что такова длина внутреннего представления типа ***float.***

Аналогичная процедура применяется и для переменной любого другого типа. После того как указатель стал определять конкретный физический байт памяти, по этому адресу можно разместить любое значение соответствующего типа, для чего сразу за указателем без каких-либо пробелов ставится значок «*», например:

i* = 4 + 3;
j* = 17;
R* = 2 * pi;

Таким образом, значение, на которое указывает указатель, то есть собственно данные, размещенные в куче, обозначаются значком «*». Значок «*» ставится сразу за указателем. Если после указателя значок «*» отсутствует, то имеется в виду адрес, по которому размещаются данные. Динамически размещенные данные (но не их адрес!) можно использовать для констант и переменных соответствующего типа в любом месте, где это допустимо, например:

R* = sqr(R*)+sin(R*+i*)-2.3;

Но невозможен оператор

R = sqr(R*)+i*;

так как указателю ***R*** нельзя присвоить значение *вещественного* типа.

Точно так же недопустим оператор

R* = sqr(R);

поскольку значением указателя ***R*** является адрес, и его (в отличие от того значения, которое размещено по данному адресу) нельзя возводить в квадрат.

Ошибочным будет и присваивание ***R* = i***, так как вещественным *данным*, на которые указывает ***R****, нельзя давать значение указателя (*адрес*).

Динамическую память можно не только забирать из кучи, но и возвращать обратно. Для этого используется процедура ***delete(p)*** в Си++ (***free(p)*** в Си), где ***p*** — указатель, который не изменяет значение указателя, а лишь возвращает в кучу память, ранее связанную с указателем.

При работе с указателями и динамической памятью необходимо самостоятельно следить за правильностью использования процедур ***new***, ***delete,*** и за работами с адресами и динамическими переменными, так как транслятор эти ошибки не контролирует. Ошибки этого класса могут привести к зависанию компьютера, а то и к более серьезным последствиям!

Другая возможность состоит в освобождении целого фрагмента кучи. С этой целью перед началом выделения динамической памяти текущее значение указателя запоминается в переменной-указателе (обозначим её ***mark***). Тогда можно в любой момент освободить фрагмент кучи, начиная с того адреса, который запомнила переменная-указатель ***mark***, и до конца динамической памяти. Для этого используется процедура ***realloc.***

14.2. Отработка приемов тестирования и отладки программного обеспечения.

После завершения комплексного тестирования приступают к оценочному тестированию, целью которого является тестирование программы на соответствие основным требованиям. Эта стадия тестирования особенно важна для программных продуктов, предназначенных для продажи на рынке.

Оценочное тестирование, которое также называют «*тестированием системы в целом*», включает следующие виды:

- тестирование *удобства использования* — последовательная проверка соответствия программного продукта и документации на него основным положениям технического задания (ТЗ);
- тестирование *на предельных объемах* —проверка работоспособности программы на максимально больших объемах данных, например, объемах текстов, таблиц, большом количестве файлов и тому подобное;
- тестирование *на предельных нагрузках* — проверка выполнения программы на возможность обработки большого объема данных, поступивших в течение короткого времени;
- тестирование *удобства эксплуатации* — анализ психологических факторов, возникающих при работе с программным обеспечением; это тестирование позволяет определить, удобен ли интерфейс, не раздражает ли цветовое или звуковое сопровождение и тому подобное;
- тестирование *защиты* — проверка защиты, например, от несанкционированного доступа к информации;
- тестирование *производительности* — определение пропускной способности при заданной конфигурации и нагрузке;
- тестирование *требований к памяти* — определение реальных потребностей в оперативной и внешней памяти;
- тестирование *конфигурации оборудования* — проверка работоспособности программного обеспечения на разном оборудовании;
- тестирование *совместимости* — проверка преемственности версий: в тех случаях, если очередная версия системы меняет форматы данных, она должна предусматривать специальные конвекторы, обеспечивающие возможность работы с файлами, созданными предыдущей версией системы;
- тестирование *удобства установки* — проверка удобства установки;
- тестирование *надежности* — проверка надежности с использованием соответствующих математических моделей;
- тестирование *восстановления* — проверка восстановления программного обеспечения, например, системы, включающей базу данных, после сбоев оборудования и программы;

— тестирование *удобства обслуживания* — проверка средств обслуживания, включенных в программное обеспечение;
— тестирование *документации* — тщательная проверка документации, например, если документация содержит примеры, то их все необходимо попробовать;
— тестирование *процедуры* — проверка ручных процессов, предполагаемых в системе.

Естественно, целью всех этих проверок является поиск несоответствий техническому заданию. Считают, что только после выполнения всех видов тестирования программный продукт может быть представлен пользователю или к реализации. Однако на практике обычно выполняют не все виды оценочного тестирования, так как это очень дорого и трудоемко. Как правило, для каждого типа программного обеспечения выполняют те виды тестирования, которые являются для него наиболее важными. Так базы данных обязательно тестируют на предельных объемах, а системы реального времени — на предельных нагрузках.

Вопросы для контроля

1. Дайте определение динамической структуры данных. Охарактеризуйте её особенности.

2. В каких же случаях возникает необходимость использования динамических структур данных?

3. Какое представление динамических данных называется связным? Каковы преимущества такого представления данных?

4. Какое представление динамических данных называется несвязным? Каковы преимущества такого представления данных?

5. Дайте определение указателя. Приведите примеры описания указателей.

Глава 15. Моделирование процессов компиляции с помощью обработки различных структур, представляющих выходные потоки этапов компиляции

Учебные цели:

- *дать представление о выполнении операций над строками, структурами или списками;*
- *познакомить с приемами выполнения операций над строками, структурами или списками.*

Процессы (в том числе процесс компиляции) в операционных системах описываются структурными данными, элементами которых могут быть данные самых разных типов:

- строки, которые могут представлять уникальные идентификаторы и имена, например, имена (дескрипторы) файлов;
- базовые переменные в цифровой и булевой форме, представляющие уникальные идентификаторы, флаги, например, время создания и завершения процесса, маркеры доступа;
- указатели, например, указатель на список потоков процесса;
- массивы, представляющие, например, таблицы (к примеру, таблица дескрипторов процесса);
- структуры, представляющие блоки данных, например, блок переменных окружения процесса;
- списки разных типов, необходимые для хранения данных, например, двунаправленный список потоков процесса, список потоков процесса;
- и другие сложные типы данных.

Для моделирования даже простого процесса компиляции для простой входной строки потребуются знания и методы работы с разными структурами. Так в качестве исходной программы, которую будет обрабатывать компилятор, достаточно взять строку, представляющую конструкции простой формальной грамматики.

В процессе выполнения фазы анализа (лексического, синтаксического, семантического анализов) потребуется

- использовать списки, структуры, флаги для построения вспомогательных таблиц, например, (таблицы идентификаторов, таблицы символов),
- использовать методы динамического распределения памяти (при построении вспомогательных стеков) для построения или оптимизации конструкций промежуточного представления (например, дерева) в фазах анализа и синтеза,
- выполнить построения конструкций (тетрад, триад, польской записи) для подготовки к фазе генерации команд.

В предыдущих главах рассматривались описания, способы и особенности использования многих из указанных в приведенном выше списке структурных типов данных. В этой главе рассмотрим некоторые особенности работы с такими структурными данными, как строки и структуры. Особенности работы со списками подробно разбирались в главе 11.

15.1. Выполнение операций над строками, структурами или списками

Операции над структурами

Рассмотрев способы описания структурных переменных, естественно рассмотреть, каким образом с их помощью можно обрабатывать данные в программах пользователя.

К структурным данным в Си/Си++ применимы следующие операции:

Таблица 15.1.

№	Название операции	Знак операции
1.	выбор элемента через *имя* (***селектор***)	**.** «точка»
2.	выбор элемента через *указатель* (***селектор***)	-> (минус и знак больше)
3	***присваивание***	=
4.	***взятие адреса***	&

1. Для *получения доступа* к элементу структурной переменной через ее имя используется символ «точка», обозначающий операцию *выбор поля структуры*. Например, пусть структурные переменные ***pos, pos1*** и массив ***pos2***, описаны как типовая структура ***RECORD:***

```
struct RECORD                 //описание структуры
{  int num;                    /* Табельный номер */
   char name[20];             /* Имя служащего    */
   float wage;                /* Зарплата в месяц */
};
struct RECORD pos, pos1, pos2[5];   /*описание структур pos, pos1,
                                      массива структур pos2*/
```

Применение к структуре ***pos*** операции ***«.»*** записью ***pos.num*** осуществляет доступ к полю ***num*** структурной переменной ***pos***, а ***pos1.num*** — поле с тем же именем и того же типа, но в составе структурной переменной ***pos1***. Эти элементы можно записывать в любых выражениях языка Си, допускающих использование переменных целого типа, так как элемент ***num*** структурного шаблона ***RECORD*** имеет тип ***int***:

```
pos.num = 67; //целочисленному полю num структуры pos присвоено
              //целое число 67
if ((pos.num < 1330) && (pos1.num = = 1340))...{….}
              /* целочисленное поле num структуры pos используется в
                 условии условного оператора if*/
```

Аналогично ***pos.wage*** и ***pos1.wage*** — переменные с плавающей точкой, а ***pos.name*** и ***pos1.name*** есть константы-указатели на символьные строки,

ибо ***name*** — это имя массива (а имя массива — одновременно ссылка на начало области, отведенной под элементы массива).

Для массива структур перед выделением элемента структуры необходимо указать номер элемента массива, для которого выделяется элемент структуры:

pos2[0].num — элемент ***num*** первого элемента (первой структуры) массива структур;

pos2[3].wage — элемент ***wage*** четвертого элемента (четвертой структуры) массива структур.

Обратите внимание, что индекс структуры в массиве структур записывается сразу после имени массива, а не в конце выражения, то есть запись ***pos2.num[0]*** — неверна.

В случае вложенных структур операция выделения элемента применяется необходимое число раз. Например, для описаний структур ***NAME*** и ***RECORD***:

```
struct NAME
{ char fam[10];   //фамилия
  char im[10];    //имя
};
struct RECORD
{ int num;
  struct NAME name;
  float wage;
};
struct RECORD pos, pos1, pos2[5];
```

выражение ***pos.name.fam*** адресует элемент ***fam*** структурной переменной ***name (***описанной по типу структуры ***NAME)***, которая входит в структурную переменную ***pos***, (описанную по типу структуры ***RECORD***)

2. *Присваивание*. Можно использовать *имена структурных переменных в качестве операндов операции* присваивания. При этом обе структурные переменные (слева и справа от знака операции присваивания «=») должны быть описаны с помощью одного и того же структурного шаблона. Так, для структурных переменных ***pos, pos1*** и массива ***pos2***, описанных выше, допустимы следующие операторы:

```
pos = pos1;
pos1 = pos2[1];
pos2[2] = pos;
```

Выполнение подобных присваиваний заключается в копировании значений всех элементов (в том числе и массивов) *правой* структурной переменной в соответствующие элементы *левой* структурной переменной, то есть мы обращаемся к структурной переменной как *к единой совокупности значений*.

3. К структурным переменным, точно так же, как и к обычным переменным, применима операция *взятия адреса*, обозначаемая знаком

«***&***». В качестве примера использования операции взятия адреса рассмотрим все тот же структурный шаблон ***RECORD*** и описание структурной переменной ***w***:

```
struct RECORD
{ int num;
  char name[20];
  float wage;
};
struct RECORD w;
```

После того как описан структурный шаблон, то есть определен тип ***struct RECORD***, можно описать *указатель* на этот тип. Описание указателя производится аналогично описанию указателей на другие типы данных:

```
struct RECORD *ptrw;
```

Итак, описаны структурная переменная ***w*** типа ***struct RECORD*** и указатель ***ptrw*** на тип ***struct RECORD***. Теперь к структурной переменной ***w*** можно применить операцию взятия адреса и присвоить полученный адрес указателю ***ptrw***:

```
ptrw = &w;
```

Заметим, что тот же результат можно получить и по-другому. При описании указатель может быть инициализирован адресом любой структурной переменной соответствующего типа, например, так:

```
struct RECORD *ptrw=&w;
```

Теперь ***ptrw*** ссылается на структурную переменную ***w***. Каким же образом можно использовать указатель ***ptrw*** для получения значения элемента структурной переменной ***w***? Для ответа на этот вопрос рассмотрим операцию *выделения элемента через указатель*.

4. *Выделение элемента структурной переменной через указатель* можно осуществить двумя способами.

Первый, наиболее часто применяемый способ, использует знак операции «->» (операция «->» записывается как минус, за которым следует символ отношения больше). Действие операции «->» заключается в следующем:

Если ***P*** есть указатель на структурный тип и ***P*** содержит адрес переменной ***S*** такого же структурного типа, тогда обращение ***P-><элемент_структуры>*** относится к конкретному элементу структурной переменной ***S***.

В условиях предыдущего описания данных и инициализации указателя получим обращения:

ptrw->num к содержимому элемента ***num*** структурной переменной ***w***;

ptrw->name к содержимому элемента ***name*** структурной переменной ***w***;

ptrw->wage к содержимому элемента ***wage*** структурной переменной ***w***.

Обратим внимание на то, что в данном случае для обращения к элементу нельзя записать, например, ***ptrw.num***, так как ***ptrw*** не является

именем структурной переменной (это указатель, то есть содержит адрес, по которому находится структурная переменная).

Второй способ обращения к элементу структурной переменной через ее указатель заключается в применении уже известной операции *косвенной адресации*.

Суть состоит в следующем: поскольку ***ptrw*** является указателем на структурную переменную **w**, то, по определению операции косвенной адресации, использование выражения ****ptrw*** равносильно использованию переменной ***w***. И то, и другое реализует обращение к одному и тому же полю памяти. Следовательно, ***(*ptrw).num*** представляет собой обращение к элементу ***num*** структурной переменной ***w***.

Замечание. Скобки здесь обязательны, так как операция «.» (точка) имеет приоритет выше, чем операция «*» (звездочка). Таким образом, обращения к полям переменной ***w*** можно переписать так:

(*ptrw).num к содержимому элемента ***num*** структурной переменной ***w;***
(*ptrw).name к содержимому элемента ***name*** структурной переменной ***w;***
(*ptrw).wage к содержимому элемента ***wage*** структурной переменной ***w.***

Заметим, что первый способ более распространен, так более лаконичен и -> выполняет роль селектора при указателе на структурную переменную.

5. Обращаться к полям структурной переменной можно и при помощи *полей-функций*. При этом функции имеют *прямой доступ* к полям структурной переменной, операцию селектор использовать не надо. Ниже приведен фрагмент программы, в которой поля–функции используются для печати значений переменной (функция ***print_rec()***) и для изменения значения поля ***wage*** (функция ***wage_raise()***).

```
struct RECORD
   { int num;              // Табельный номер
     char name[20];        // Имя служащего
     float wage;           // Зарплата в месяц
   }
    void print_rec()
      {printf("num=%d\n name=%s\nwage = %f \n\n", num, name,wage); }
    void init(int num1, char *name1, float wage1)
            {  num = num1; strcpy(name, name1); wage = wage1;  }
     float wage_raise(int d)
           { return wage = wage*d;  }
//объявление и инициализация структурной переменной
RECORD r = {1328, "Ivanov Ivan",2};
//печать значений структурной переменной
printf(" %d \n %s \n %f \n\n", r.num, r.name, r.wage);
//инициализация структурной переменной
```

r.init(1329, "Petrov Petr", 20);
//печать значений структурной переменной
r.print_rec();
//изменение значения поля структурной переменной
r.wage = r.wage * 5;
//изменение значения поля структурной переменной
r.wage_raise(5);
wage_raise(5); //ошибка, обращаться к полям–функциям можно только //через структурную переменную

Описания строк и особенности работы со строками.

Строки в Си/Си++могут определяться следующим образом:

- как строковые константы;
- как массивы символов;
- через указатель на символьный тип;
- как массивы строк.

Для строки (хранения строки), как и для массива, должно быть предусмотрено выделение памяти.

Любая последовательность символов, заключенная в двойные кавычки «“ ”», рассматривается как строковая константа.

Любая строка должна заканчиваться *нуль-символом* «***'\0'***», целочисленное значение которого равно **0** (это важно для корректного вывода). При объявлении строковой константы *нуль-символ* добавляется к ней автоматически. То есть, если объявлена пустая строка, то в ней содержится только *нуль-символ***.**

Под хранение строки выделяются последовательно идущие ячейки оперативной памяти. Для хранения кода каждого символа строки отводится 1 байт. Строка представляет собой массив символов, к которому добавлен в конце нуль символ. Таким образом, при вводе последовательности символов, представляющей строковую константу, в оперативной памяти компьютера будут размещены символы этой последовательности в последовательных байтах, включая последний нулевой байт. Для помещения в строковую константу некоторых служебных символов используются символьные комбинации. Так, если необходимо включить в строку символ двойной кавычки, ему должен предшествовать символ «обратный слеш»: «\”».

Строковые константы размещаются в статической памяти. Начальный адрес последовательности символов в двойных кавычках трактуется как адрес строки. При определении массива символов необходимо сообщить компилятору требуемый размер памяти.

char m[82];

Компилятор самостоятельно определяет размер массива символов, если при объявлении строки производится инициализация строковой константой:

char m2[] = "Умом Россию не понять.";
char m3[] = {'У','м','о','м',' ','Р','о','с','с','и','ю',' ','н','е',' ','п','о','н','я','т','ь','\0'};

В этом случае имена ***m2*** и ***m3*** являются указателями на первые элементы массивов:

- ***m2*** эквивалентно ***&m2[0]***
- ***m2[0]*** эквивалентно ***'У'***
- ***m2[1]*** эквивалентно ***'м'***
- ***m3*** эквивалентно ***&m3[0]***
- ***m3[2]*** эквивалентно ***'о'***

При объявлении массива символов и инициализации его строковой константой можно явно указать размер массива, но указанный размер массива должен быть больше, чем размер инициализирующей строковой константы, по крайней мере, на 1 байт, например

char m2[40] = "Умом Россию не понять.";

Для задания строки можно использовать указатель на символьный тип.

char *m4;

При таком объявлении массива переменной ***m4*** может быть присвоен адрес массива ***m3***:

m4 = m3;
****m4*** эквивалентно ***m3[0]='У'***
****(m4+1)*** эквивалентно ***m3[1]='м'***

После операции присваивания ***m3*** является константой-указателем, и уже нельзя изменить ***m3***, так как это означало бы изменение положения (адреса) массива в памяти, в отличие от ***m4,*** который не связан с какой-либо структурой.

Для указателя можно использовать операцию увеличения (перемещения на следующий символ):

m4++;

Массивы символьных строк

Нередко возникает необходимость описание *массива символьных строк*. В этом случае можно использовать индекс строки для доступа к нескольким разным строкам.

char *poet[4] = {"Погиб поэт!", "— невольник чести —", " Пал, ", "оклеветанный молвой…"};

П	о	г	и	б		п	о	э	т	!	\0									
—		н	е	в	о	л	ь	н	и	к		ч	е	с	т	и		—	\0	
	П	а	л	,		\0														
о	к	л	е	в	е	т	а	н	н	ы	й	м	о	л	в	о	й	…	\0	

Рис. 15.1. Размещение символьных строк в случае массива из четырёх указателей с непосредственной инициализацией символьными константами

В этом объявлении ***poet*** является массивом, состоящим из четырех указателей на символьные строки (рис. 15.1). Каждая строка символов

представляет собой символьный массив, поэтому имеется четыре указателя на массивы. Указатель ***poet[0]*** ссылается на первую строку:

****poet[0]*** эквивалентно **'П'**,

****poet[1]*** эквивалентно **'—'**.

Инициализация выполняется по правилам, определенным для массивов. Тексты в кавычках эквивалентны инициализации каждой строки в массиве. Запятая разделяет соседние последовательности.

Кроме того, можно явно задавать размер строк символов, используя описание, подобное следующему:

char poet[4][23];

Разница заключается в том, что такая форма задает «прямоугольный» массив, в котором все строки имеют одинаковую длину (15.2).

П	*о*	*г*	*и*	*б*		*п*	*о*	*э*	*т*	*!*	*\0*										
—		*н*	*е*	*в*	*о*	*л*	*ь*	*н*	*и*	*к*		*ч*	*е*	*с*	*т*	*и*		*—*	*\0*			
	П	*а*	*л*	*,*		*\0*																
о	*к*	*л*	*е*	*в*	*е*	*т*	*а*	*н*	*н*	*ы*	*й*	*м*	*о*	*л*	*в*	*о*	*й*	*…*	*\0*			

Рис. 15.2. Размещение символьных строк в случае массива из четырёх указателей с одинаковым ограничением памяти для строк, адреса которых они содержат

Кроме объявлений, приведенных выше, существует понятие *свободного массива*. *Свободный массив* — это массив, в котором длина каждой строки определяется тем указателем, который эту строку инициализирует. Таким образом, свободный массив тратит память весьма экономно. Описание

char *poet[4];

определяет свободный массив,

Операции со строками

Большинство операций языка Си, имеющих дело со строками, работает с указателями. Для размещения в оперативной памяти строки символов необходимо:

— выделить блок оперативной памяти под массив;

— проинициализировать строку.

Для выделения памяти под хранение строки могут использоваться функции динамического выделения памяти. При этом необходимо учитывать требуемый размер строки:

```
char *name;       //объявлен указатель на символьную строку
name = (char*)malloc(10);  /*для строки выделена память в 10 символов и
                             адрес этой памяти записан в указатель name*/
scanf("%9s", name);       //осуществляется ввод строки с консоли
```

Для ввода строки использована функция ***scanf()***, в формате которой вводимая строка ограничена девятью (9) символами. Последним символом строки автоматически будет вставлен нуль-символ «**'\0'**».

Функции ввода и вывода строк в Си/Си++ подробно рассматривать не будем, этому достаточно много времени уделяется в процессе изучения дисциплины «Программирование»: ***scanf(0), printf(), gets(), gets_s()***, ***puts()*** для ввода-вывода строк и ***getchar(), putchar()*** — для ввода-вывода символов. Отметим только, что очень полезно серьезнее отнестись к изучению функций ввода-вывода языка Си, так как это позволит глубже понять взаимодействие типов данных и размещение их в памяти.

По тем же причинам не будем рассматривать особенности применения функций стандартной библиотеки ***string.h.*** Необходимо следить, чтобы эта библиотека была подключена в директивах, иначе функции не будут известны интегрированной среде. Перечислим основные из стандартных функций этой библиотеки: ***strcat(), strncat(), strcpy(), strncpy(), strcmp(), strncmp(), strlen(), strset(), strnset()*** и многие другие, которые подробно изучаются в программировании.

Вопросы для контроля

1. Какие структурные типы данных используются при разработке компиляторов?

2. Опишите, какие возможности существуют для обращения к элементу структурной переменной через ее указатель.

3. Опишите, процесс взятия адреса через указатель. Приведите пример для вложенных структур.

4. Каковы особенности представления строки в оперативной памяти в Си/Си++? Приведите примеры объявления строк, в том числе с использованием указателя,

5. Опишите возможности объявления массивов строк. Покажите размещение строк в памяти для разных вариантов.

Глава 16. Принципы функционирования систем программирования.

Учебные цели:

— *дать представление об интегрированных средах разработки;*
— *познакомить со структурой и функциональными средствами систем программирования;*
— *дать представление об основах кроссплатформенного программирования.*

16.1. Интегрированные среды разработки.

Интегрированная среда разработки (**IDE** – *Integrated Development Environment*) — специальная программа, предоставляющая возможность удобной совместной работы с различными компонентами системы программирования.

В предыдущих главах рассматривались компоненты, входящие в систему программирования. Это и *редакторы кода*, и *компиляторы*, и *сборщики* (*компоновщики*), и *отладчики*, и многие другие.

При первом же знакомстве со всеми этими программами становится понятно, что каждая из них может работать с разными начальными установками. Например, можно настроить множество параметров для редактора кода: цвет фона, цвет шрифта, шрифт, размер символа табуляции и еще сотню разных характеристик. Для компилятора можно указать, как оптимизировать код: по скорости, по размеру, никак не оптимизировать, а также есть возможность управления многими другими параметрами. Аналогично обстоит дело практически со всеми составляющими системы программирования.

Теперь представьте себе процесс работы над программой, когда по очереди запускаются все эти программы с огромным количеством разных параметров. То есть, сначала запускается редактор кода и пишется в нем программа. После этого подготовленная программа сохраняется, а затем закрывается редактор. Далее запускается компилятор со всеми указанными ему необходимыми настройками и именем файла с текстом программы в качестве параметра. Например, компилятор отработал и нашел 4 ошибки. Снова запускается текстовый редактор и в результате титанических усилий находятся эти строки и ошибки в них. После этого снова сохраняется программа, закрывается редактор и опять запускается компилятор. В результате, когда избавились от синтаксических ошибок, компилятор создает *объектный код* (уже без синтаксических ошибок) и закрывается. Теперь запускается *сборщик* (редактор связей, компоновщик, линкер), с указанным ему набором параметров и тем самым объектным файлом, полученным на этапе компиляции. Если ошибок нет, то наконец, получится *исполняемый файл*,

Рапускаете этот исполняемый файл на исполнение, программа запустилась и … «повисла». Или не повисла, но появилось сообщение, что уравнение не имеет корней, хотя точно известно, что решение есть. Это значит, что теперь с семантикой что-то не то. Иначе говоря, программа работает неправильно: либо неправильно запрограммирован алгоритм, либо ошибка в алгоритме. Эту программу необходимо отлаживать, искать ошибки.

Для отладки программ существует программное средство — *отладчик*. И весь процесс продолжается, начиная с редактора текста.

Для устранения неудобств и повышения эффективности процесса разработки программ, программных продуктов создатели систем программирования стали строить их в виде так называемых «*интегрированных сред разработки*». Термин «*интегрированная*» в названии среды означает, что она включает в себя в качестве элементов все необходимые инструменты для выполнения полного цикла работ над программой: написания, компиляции, построения исполняемого модуля, запуска, отладки. Кроме того, интегрированные среды позволяют выполнять следующие операции:

- —визуально (в диалоге) производить быструю настройку параметров каждого из компонентов системы программирования;
- —сохранять разные системы настроек и загружать их по мере необходимости;
- —нажатием нескольких клавиш или выбором соответствующих пунктов меню осуществлять запуск одного или сразу нескольких компонентов системы программирования, автоматизируя процесс передачи им необходимых параметров.

В любой интегрированной среде исполняемый модуль из исходного текста программы можно получить нажатием пары кнопок на клавиатуре. Единственный минус таких сред является прямым следствием их главного плюса: собрав «под одной крышей» большой набор инструментов, интегрированная среда сама становится весьма сложной программой. Однако время, потраченное на ее изучение, окупается в дальнейшем. И, наконец, еще один положительный момент — устройство большинства сред одинаково в концептуальном плане, различия наблюдаются лишь в комбинациях клавиш для того или иного действия и/или в названиях пунктов меню.

Одно из последних достижений в области разработки программного обеспечения — *визуальные среды программирования* (самые известные – ***Borland® Delphi™*** с базовым языком ***Object Pascal*** и многоязыковая среда ***Microsoft® Visual Studio.NET***). Их появление связано с двумя важными факторами. Во-первых, это стремление человека максимально

автоматизировать собственный труд. Второй фактор связан с тем, что современный пользователь в большинстве своем не станет работать с программой, у которой неудобен или непонятен интерфейс. Сейчас при создании программ их внешнему виду уделяется не меньшее значение, чем внутреннему содержанию. Поэтому, естественно, появились и стали актуальны визуальные среды.

Совсем недавно визуальные среды были примитивны и имели ограниченный функционал, в отличии от интегрированных, основанных на работе с языками. Но цифровые технологии не стоят на месте и уже сегодня доступны мощные инструменты, способные заменить код на удобный интерфейс (манипуляции с объектами).

В основе визуальных сред, тоже лежит язык программирования, но в отличие от интегрированных сред, которые предназначены для специалистов-программистов, в визуальных средах не всегда предполагается знание языка программирования (не нужно прописывать его самостоятельно), поэтому визуальные среды востребованы в большой мере для специалистов предметных областей. Процесс создания программы заключается в манипуляции блоками: их обозначением и соединением в последовательную цепь. Универсальные возможности визуальных сред, позволяют воплощать самые необычные проекты и в разы ускоряют процесс создания продукта.

Приведем примеры некоторых современных визуальных сред разного назначения.

Scratch — одна из лучших обучающих программ для создания программ на русском языке. Разработана для практического изучения программирования. Имеет очень простой интерфейс, визуальный редактор, но довольно узкий функционал.

Visual Studio 2013 — профессиональная среда разработки программного обеспечения (на английском языке). Включает в себя огромный набор возможностей и инструментов, необходимых для создания программ любой сложности.

Microsoft Visual Studio 2019 — целый комплекс утилит с интерфейсом на русском языке, предназначенных для разработки консольных приложений с графической оболочкой, вэб-программ, сайтов и служб, которые поддерживаются платформами компании *Microsoft*.

HiAsm — визуальный конструктор на русском языке, позволяющий новичкам создавать программы без знания специальных языков.

Алгоритм — удобный конструктор на русском языке для новичков, позволяющий без использования языков программирования создать свою полноценную программу. Обладает гибким функционалом и понятным интерфейсом, простой в освоении.

16.2. Структура и функциональные средства систем программирования.

Системой программирования будем называть весь комплекс программных средств, предназначенных для кодирования, тестирования и отладки программного обеспечения. Нередко системы программирования взаимосвязаны и с другими техническими средствами, служащими целям создания программного обеспечения на более ранних этапах жизненного цикла (от формулировки требований и анализа до проектирования). Однако рассмотрение таких систем выходит за рамки данного учебного пособия.

Системы программирования в современном мире доминируют на рынке средств разработки. Практически все фирмы-разработчики компиляторов поставляют свои продукты в составе соответствующей системы программирования в комплексе всех прочих технических средств. Отдельные компиляторы являются редкостью и, как правило, служат только узко специализированным целям.

Термин «ресурсы» не слишком удачный, так как этим словом обозначаются очень многие понятия, связанные с вычислительными системами (например, ресурсы вычислительного процесса). Однако этот термин применяется при работе со средствами разработки, поэтому придется смириться с таким его использованием.

С точки зрения терминологии термин «компилятор ресурсов», присутствующий на рисунке 16.1 правильнее было бы назвать «транслятор ресурсов», так как в результате своей работы они обычно порождают не объектный файл, а некий промежуточный код ресурсов. Однако термин «компилятор ресурсов» стал уже общепринятым.

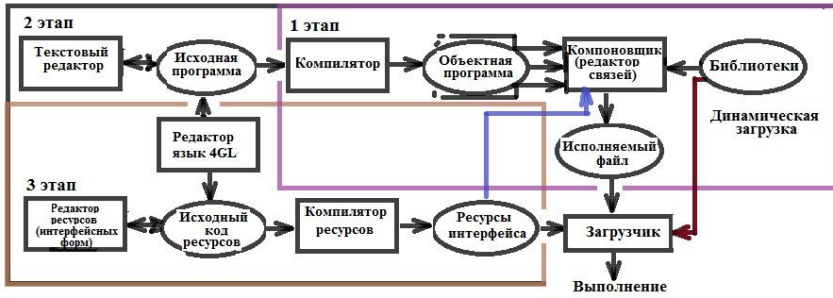


Рис.16.1 Общая структура и этапы развития систем программирования

На рисунке 16.1 приведена общая структура современной системы программирования. На ней выделены все основные составляющие современной системы программирования и их взаимосвязь. Отдельные составляющие разбиты по группам в соответствии с этапами развития средств разработки. Эти группы отражают все этапы развития от отдельных программных компонентов до цельной системы программирования.

Из этой схемы (рис. 16.1) видно, что современная система программирования — это достаточно сложный комплекс различных программно-технических средств. Все они служат цели создания прикладного и системного программного обеспечения. Тенденция такова, что всё развитие систем программирования идет в направлении неуклонного повышения их дружественности и сервисных возможностей. Это связано с тем, что на рынке, в первую очередь, лидируют те системы программирования, которые позволяют существенно снизить трудозатраты, требуемые для создания программного обеспечения на этапах жизненного цикла, связанных с кодированием, тестированием и отладкой программ. Показатель *снижения трудозатрат* в настоящее время считается *более существенным*, чем показатели, определяющие эффективность результирующей программы, построенной с помощью системы программирования.

В качестве основных тенденций в развитии современных систем программирования следует указать внедрение в них средств разработки на основе так называемых «языков четвертого поколения» — ***4GL*** (*four generation languages*), — а также поддержка систем «быстрой разработки программного обеспечения» —***RAD*** (*rapid application development*).

Языки четвертого поколения — ***4GL***— представляют собой широкий набор средств, ориентированных на проектирование и разработку программного обеспечения. Они строятся на основе оперирования *не синтаксическими структурами языка и описаниями элементов интерфейса*, а представляющими их графическими образами. На таком уровне проектировать и разрабатывать прикладное программное обеспечение может пользователь, не являющийся квалифицированным программистом, зато имеющий представление о предметной области, на работу в которой ориентирована прикладная программа. Языки четвертого поколения являются следующим (четвертым по счету) этапом в развитии систем программирования.

Описание программы, построенное на основе языков ***4GL***, транслируется затем в исходный текст и файл описания ресурсов интерфейса, представляющие собой обычный текст на соответствующем входном языке высокого уровня. С этим текстом уже может работать профессиональный программист-разработчик — он может корректировать и дополнять его необходимыми функциями. Дальнейший ход создания программного обеспечения идет уже традиционным путем, как это показано на рисунке 16.1.

Такой подход позволяет разделить работу на независимые части:

— работу проектировщика, ответственного за общую концепцию всего проекта создаваемой системы,
— дизайнера, отвечающего за внешний вид интерфейса пользователя,
— и профессионального программиста, отвечающего непосредственно за создание исходного кода создаваемого программного обеспечения.

В целом языки четвертого поколения решают уже более широкий класс задач, чем традиционные системы программирования. Они составляют часть средств автоматизированного проектирования и разработки программного обеспечения, поддерживающих все этапы жизненного цикла, а именно, ***CASE***-систем.

16.3. Основы кроссплатформенного программирования. Трансляция адресов

Прежде всего, дадим определение *кроссплатформенности* или межплатформенности и связанных с ним понятий. Кроссплатформенность — это способность программного обеспечения работать с двумя и более аппаратными платформами и (или) операционными системами.

Кроссплатформенная программа — это программа, у которой есть версии для разных операционных систем.

Обеспечивается кроссплатформенность благодаря использованию следующих важных составляющих:

— *высокоуровневых* языков программирования,

— *сред разработки и выполнения*, поддерживающих *условную компиляцию*,

— *компоновку и выполнение кода*

для *различных* платформ. Типичным примером является программное обеспечение, предназначенное для работы в операционных системах ***Linux*** и ***Windows*** одновременно.

Анализируя приведенные выше составляющие кроссплатформенности, нужно отметить, что большинство современных высокоуровневых программирования являются кроссплатформенными. Это, например, ***Cи, Cи++, Free Pascal, FreeBASIC, PureBasic*** — кроссплатформенные языки *на уровне компиляции.* Это означает, что для этих языков есть компиляторы под *различные платформы,* что позволяет (при надлежащем качестве кода) не переписывать основной движок[26] программы, а менять только особые системозависимые части.

Не менее важны для кроссплатформенности стандартизованные библиотеки среды выполнения (вторая составляющая кроссплатформенности). В частности, стандартом стала библиотека языка Си (***POSIX***), примерами других крупных кроссплатформенных библиотек являются ***Qt, GTK+, FLTK, STL, Boost, OpenGL, SDL, OpenAL, OpenCL.***

И, наконец, несколько слов о *кроссплатформенном пользовательском интерфейсе.* На разных операционных системах, независимо от того, как

[26] **Движо́к** (жаргонное от *engine* — мотор, двигатель) — выделенная часть программного кода для реализации конкретной прикладной задачи. Это программа, часть программы, библиотека, комплекс программ и приложений, который объединены в общую систему для того, чтобы функционировал программный продукт, и им можно было управлять. Движок несет основную функцию в виде предоставления инструментов для создания контента в зависимости от задачи и реализации. Как правило, прикладная часть выделяется из программы для использования в нескольких проектах и/или раздельной разработки/тестирования.

технически достигнута работа в них, стандартные элементы интерфейса имеют разные размеры. Поэтому простое жёсткое позиционирование элементов интерфейса невозможно: под другой операционной системой необходимо обеспечить их совместимость.

Взаимодействие пользователя с интерфейсом или иным продуктом называют пользовательским опытом. Этот термин придумал Дональд Норман.

UX [юикс] расшифровывается как *user experience* и переводится на русский язык как «пользовательский опыт». Этот «пользовательский опыт» может быть продуманным, лёгким и комфортным, либо тяжёлым, расстраивающим и неудачным. Хороший **UX** — это когда продукт удобен и продуман (например, сайт открывается быстро и не требует 10 кликов, чтобы выдать нужную информацию). Клиентский опыт — это тоже пользовательский опыт, и его тоже можно проектировать так же, как сайты.

UX-дизайнер или проектировщик действует на основе наблюдений за пользователями и строит гипотезы, как должен работать продукт, а затем проверяет их тестами и исследованиями. Такая практика широко применяется в дизайне интерфейсов.

UI [юай] расшифровывается как *user interface* и переводится «пользовательский интерфейс». Так как в последние десятилетия произошёл бум в интерфейсах, под термином **UI** обычно понимают экранные интерфейсы и их внешний вид. **UI** – это про визуальный стиль, графический дизайн на экране. **UI**-дизайнер умеет работать с типографикой, цветами, композицией, анимацией. На основе черновых схематичных набросков (вайрфреймов) **UI**-дизайнер собирает финальный дизайн-макет.

Рынку мобильных приложений второй десяток лет, однако его развитие сегодня не менее бурно развивается. Спрос на создание мобильных приложений со стороны компаний постоянно растёт, и он всё ещё заметно превышает предложение, что приводит к постоянному удорожанию разработки. Одно из решений в удешевлении этого процесса — *кроссплатформенная* разработка, когда один и тот же код используется на всех платформах.

Если разработчики в процессе написания приложения пользуются *принятым для конкретной платформы* языком программирования, касается ли это ***Objective-C***[27] и ***Swift***[28] для ***iOS*** или ***Java или Kotlin***[29] ***для Android***, такое приложение называется *нативным* (от англ. *native* — родной, естественный).

---

[27] ***Objective-C*** — компилируемый объектно-ориентированный язык программирования, используемый корпорацией *Apple*, построенный на основе языка *Си* и парадигм параллельного языка ***Smalltalk***. Язык ***Objective-C*** является надмножеством языка Си, поэтому Си-код полностью понятен компилятору ***Objective-C***.

[28] ***Swift*** — это надёжный и интуитивно понятный язык программирования с открытым кодом от ***Apple***, при помощи которого можно создавать приложения для платформ ***iOS***, ***Mac***, ***Apple TV*** и ***Apple Watch***.

Swift — это быстрый и эффективный язык программирования с откликом в реальном времени, который легко можно вставить в готовый код ***Objective-C***. На языке ***Swift*** можно писать более надёжные и безопасные коды, экономить время и создавать приложения с расширенными возможностями.

[29] ***Kotlin*** (*Кóтлин*) — статически типизированный, объектно-ориентированный язык программирования, работающий поверх ***Java Virtual Machine*** и разрабатываемый компанией ***JetBrains***. Компилируется в ***JavaScript*** и в исполняемый код

Отметим преимущества *нативных приложений*:

- *скорость работы и отклика интерфейса*: приложение реагирует на нажатия объекты мгновенно, практически отсутствуют задержки в анимации, скроллировании, получении и выводе данных;
- *понятный и простой доступ* к функциям и датчикам устройства: для разработчика не представляет проблемы работа с геолокацией, пуш-уведомлениями, съёмкой фото и видео через камеру, звуком, акселерометром и другими датчиками;
- *возможность углублённой работы* с функциями смартфона: как и в предыдущем пункте, такие вещи, как анимации, создание сложных интерфейсов и работа нейросетей прямо на устройствах реализуются, может быть, и не просто, но прогнозируемо;
- *родной для платформы интерфейс*: нативные приложения обычно оперируют «платформенными» элементами интерфейса: меню, навигация, формы и все остальные элементы дизайна берутся от операционной системы и потому привычны и понятны пользователю.

Недостаток один — дороговизна разработки и поддержки, потому что для каждой платформы надо писать свой код. С ростом рынка мобильных приложений разработчики стали не просто дороги, а очень дороги.

Кроссплатформенные приложения пишутся сразу для *нескольких платформ* на *одном языке, отличном от нативного*. Как такой код может работать на разных устройствах? Существуют два подхода.

Первый подход заключается в том, что на этапе подготовки приложения к публикации он превращается в нативный для определённой платформы с помощью транспилера. Фактически один кроссплатформенный язык программирования «переводится» на другой.

Второй подход состоит в том, что к получившемуся коду добавляется определённая обёртка, которая, работая уже на устройстве, на лету транслирует вызовы из *неродного кода* к родным функциям системы.

Предполагается, что большая часть такого кода может переноситься между платформами. Очевидно, что, например, логика совершения покупок, сохранения товара в корзину, просчёта маршрута для такси, написания сообщения в мессенджер не меняется в зависимости от того, платформа ***Android*** у клиента или ***iOS***.

Чтобы на разных платформах приложение работало идентично, нужно лишь доработать ***UI*** и ***UX*** для этих платформ. Но сейчас, в определённых пределах, даже это можно объединить — например, меню-гамбургер активно используется как на ***Android***, так и на ***iOS***. Так что даже внесений исправления

ряда платформ через инфраструктуру ***LLVM***. Язык назван в честь острова Котлин в Финском заливе, на котором расположен город Кронштадт.

в интерфейс для того, чтобы приложение отвечало духу и букве нужной платформы — вопрос желания, необходимой скорости и качества разработки.

Такой подход даёт следующие преимущества:

— *стоимость и скорость разработок:* ввиду того, что размер требуемого кода становится заметно меньше, то и стоимость работ снижается;

— *возможность использовать внутренние ресурсы компании*: кроссплатформенную разработку мобильных приложений зачастую можно осуществить силами программистов компании (сопровождения).

Но есть и недостатки:

— *неродной интерфейс* или, как минимум, необходимость работы с интерфейсом каждой платформы отдельно: у каждой системы свои требования к дизайну элементов и иногда они могут быть взаимоисключающими (при разработке проекта это необходимо учитывать);

— *проблемы в реализации сложных функций* или возможные проблемы работы даже с простыми процедурами в силу ошибок самих фреймворков разработки. Кроссплатформенная среда лишь *транслирует запросы* к системным вызовам и интерфейсам в понимаемый ею или системой формат, и потому на этом этапе возможны, как сложности с пониманием, так и с возникновением ошибок внутри самого фреймворка;

— *скорость работы*: так как кроссплатформенная среда является «надстройкой» над кодом (не всегда, но в определённых ситуациях), в ней возникают свои задержки и паузы в отработке действий пользователя и выводе на экран результатов; это было особенно заметно несколько лет назад на смартфонах, более маломощных относительно сегодняшних, однако сейчас, с ростом производительности мобильных устройств, этим уже можно пренебречь.

Эти два метода практически являются зеркальным отражением друг друга: один — превращение кода в нативный на этапе сборки, второй — добавление определённой обёртки, транслирующей вызовы к системе и от неё; и, соответственно, плюсы нативной разработки являются минусами кроссплатформенной и наоборот.

При разработке современных приложений, в особенности кроссплатформенных, нужно учитывать технологию трансляции сетевых адресов. Этот материал подробно изучается в дисциплине «Операционные системы и сети», поэтому здесь лишь напомним, в чём состоит этот процесс.

Любой сети, желающей подключиться к Интернет, необходим набор ***IP*** адресов, которые может выделить любая имеющая на это право

организация. Существуют три класса IP адресов: класс **A**, внутри которого можно описать до 16777214 хостов, класс **B**, позволяющий описать до 65533 хостов и класс **C**, в котором 254 хоста.

Бум, который переживает Интернет в последние годы, привел к тому, что сети класса **A** и **B** в настоящее время стали недоступны для организаций и отдельных пользователей. Поэтому этим клиентам может быть выделена только сеть класса **C** с 254 адресами. При большем числе хостов потребуются другие сети класса C, что усложняет работу администратора.

Другой способ решения проблемы состоит в использовании трансляции сетевых адресов (NAT).

Вкратце трансляция сетевых адресов — это технология, которая позволяет использовать для внутренней сети любые адреса, возможно даже из класса A; при этом сохраняется одновременный и прозрачный доступ в Интернет для всех хостов.

Механизм функционирования такого процесса достаточно прост: каждый раз, когда хост с зарезервированным адресом пытается получить доступ в Интернет, межсетевой экран контролирует эту попытку и автоматически преобразует его адрес в разрешенный. Когда хост назначения отвечает и посылает данные на разрешенный адрес, межсетевой экран преобразует его обратно в зарезервированный адрес и передает данные внутреннему хосту. При этом ни клиент, ни сервер не знают о существовании этого преобразования.

Кроме уже упомянутых преимуществ, трансляция сетевых адресов позволяет все хосты внутренней сети сделать невидимыми для внешней сети, что увеличивает уровень безопасности.

Вопросы для контроля

1. Дайте определение интегрированной среды разработки. Перечислите составляющие интегрированной среды разработки.
2. Чем характеризуется визуальная среда программирования? Какова цель их разработки?
3. Дайте определение системы программирования.
4. Дайте определение кроссплатформенности. Назовите составляющие, обеспечивающие кроссплатформенность.
5. В чём состоит суть подходов получения кроссплатформенного приложения?

Глава 17. Принципы функционирования систем программирования.

Учебные цели:

- *дать представление о месте компиляторов в составе современных систем программирования;*
- *дать представление о функционировании основных модулей системы программирования.*

17.1. Компиляторы в составе систем программирования

Компиляторы являются, безусловно, основными модулями в составе любой системы программирования. Поэтому не случайно, что они стали одним из главных предметов рассмотрения в данном учебном пособии. Без компилятора никакая система программирования не имеет смысла, а все остальные ее составляющие на самом деле служат лишь целям обеспечения работы компилятора и выполнения им своих функций.

От первых этапов развития систем программирования вплоть до появления интегрированных сред разработки пользователи (разработчики исходных программ) всегда, так или иначе, имели дело с компилятором. Они непосредственно взаимодействовали с ним как с отдельным программным модулем. Сейчас, работая с системой программирования, пользователь, как правило, имеет дело только с ее интерфейсной частью, которую обычно представляет текстовый редактор с расширенными функциями. Запуск модуля компилятора и вся его работа происходят автоматически и скрытно от пользователя: разработчик видит только конечные результаты выполнения компилятора. Хотя многие современные системы программирования сохранили прежнюю возможность непосредственного взаимодействия разработчика с компилятором (это и *Makefile*[30], и, так называемый, «интерфейс командной строки»), но пользуется этими средствами только узкий круг профессионалов. Большинство пользователей систем программирования сейчас редко непосредственно сталкиваются с компиляторами.

На самом деле, кроме самого основного компилятора, выполняющего перевод исходного текста на входном языке в язык машинных команд, большинство систем программирования могут содержать в своем составе целый ряд других компиляторов и трансляторов. Так, большинство систем программирования содержат в своем составе и компилятор с языка ассемблера, и компилятор (транслятор) с входного языка описания ресурсов. Все они редко непосредственно взаимодействуют с пользователем.

---

[30] ***Makefile*** — это файл, содержащий набор инструкций, используемых утилитой ***make*** в инструментарии автоматизации сборки (работы редактора связей).

Тем не менее, работая с любой системой программирования, следует помнить, что основным модулем её всегда является компилятор. Именно технические характеристики компилятора, прежде всего, влияют на эффективность результирующих программ, порождаемых системой программирования.

17.2. Анализ функционирования основных модулей системы программирования.

Сложная система обычно может быть разделена на более простые части — *модули*.

Модульность является важным качеством инженерных процессов и продуктов. Большинство промышленных процессов являются модульными и составлены из комплексов работ, которые комбинируются простыми способами (последовательными или перекрывающимися) для достижения требуемого результата. Главное преимущество модульности заключается в том, что она позволяет применять *принцип разделения* задач на двух этапах:

- при работе с элементами каждого модуля *отдельно* (игнорируя элементы других модулей);
- при работе с общими характеристиками групп модулей и отношениями между ними с целью *объединить* их в конкретный, более крупный и сложный компонент.

Если данные этапы выполняются в последовательности, предусматривающей сначала концентрацию процессов на модулях, а затем — их объединение, то система проектируется *снизу вверх.* Если сначала систему разбивают на модули, а потом работают над их индивидуальным проектированием, то это — проектирование *сверху вниз.*

При структурном построении комплексов программ важное значение имеет размер и сложность компонентов для каждого уровня иерархии и соответственно число иерархических уровней для крупных проектируемых систем. По принципам построения, языку описания, размеру и другим характеристикам компонентов в структуре проектируемой системы можно выделить *иерархические уровни*:

- программных модулей, оформляемых как законченные компоненты текста программ;
- функциональных групп (компонентов) или пакетов программ;
- комплексов программ, оформляемых как законченные программные продукты определенного целевого назначения.

С повышением иерархического уровня увеличивается размер текста программ, реализующих компоненты этого уровня и количество обрабатываемых переменных. Одновременно совокупности команд все более специализируется и снижается возможность повторного применения компонентов в различных комбинациях для решения аналогичных задач.

Программные модули решают относительно небольшие функциональные задачи, и каждый реализуется от десяти до ста операторами языка программирования. Каждый модуль может использовать на входе около десятка типов переменных. Если для решения небольшой функциональной задачи требуется более 100 операторов, то обычно целесообразно проводить декомпозицию задачи на несколько более простых модулей.

Функциональные группы программ (компоненты) формируются на базе нескольких или десятков модулей и решают достаточно сложные автономные задачи. На их реализацию целесообразно использовать до десятка тысяч строк текста программы. Соответственно возрастает число используемых типов переменных и разнообразие выходных данных. При этом быстро растет число типов переменных, обрабатываемых модулями и локализующихся в пределах одного или нескольких модулей.

Комплексы программ — программные продукты — создаются для решения сложных задач управления и обработки информации. В комплексы объединяются несколько или десятки функциональных групп программ для решения общей целевой задачи системы. Размеры проектируемой системы зачастую исчисляются сотнями модулей, десятками и сотнями тысяч операторов. Встречаются проектируемые системы, содержащие до двух-трех десятков структурных иерархических уровней, построенных из модулей.

Проектирование модулей включает в себя разработку

- локальных функций и подробных описаний алгоритмов обработки данных;
- межмодульных интерфейсов;
- внутренних структур данных;
- структурных схем передач управления;
- средств управления в исключительных ситуациях.

С их помощью определяются следующие функции:

- порядок следования отдельных шагов обработки,
- ситуации и типы данных, вызывающие изменения процесса обработки, а также
- повторно используемые функции программы.

Программные модули для их многократного использования должны базироваться на унифицированных правилах структурного построения, оформления спецификаций требований и описаний текстов программ и комментариев. Кроме того, целесообразно для каждого проекта директивно ограничивать размеры модулей по числу строк текста с учетом языка программирования, например, 30-ю или 50-ю операторами.

Основная цель такой унификации:

— облегчить разработку модулей,
— обеспечить их качество и тестирование, а также
— упростить управление их функциями и характеристиками.

Эти правила в значительной степени стандартизированы в современных языках программирования. Однако при их применении целесообразно выделять типовые ассоциации операторов и ограничения их использования, а также вводить правила описания текстов программ, комментариев, данных и заголовков модулей, ограничения их размеров и сложности. Эти правила наиболее полно должны соблюдаться при разработке основной массы функциональных программ, подлежащих повторному использованию в модифицируемых версиях программных продуктов. Компоненты организации вычислительного процесса, контроля функционирования, ввода-вывода и некоторые другие могут иметь отличия в правилах структурного построения модулей.

Для обеспечения *управляемой модификации* и развития конфигураций версий программного продукта важно *стандартизировать* структуру базы данных, в которой накапливается и хранится исходная, промежуточная и результирующая информация в процессе функционирования проектируемой системы. Основными компонентами этой структуры являются информационные модули или пакеты данных. В них также целесообразно использовать типовые структуры, ориентированные на эффективную обработку данных в конкретной проблемной области. Объединение информационных модулей позволяет создавать более сложные структуры данных определенного целевого назначения. Иерархия связей между этими компонентами в некоторой степени соответствует процессу обработки и потокам данных и относительно слабо связана со структурой программных компонентов в проектируемой системы. Структуры информационных модулей целесообразно координировать между собой с учетом цели и места их использования программными компонентами.

Особое значение для качества модулей и компонентов крупных проектируемых систем имеет *стандартизация структуры межмодульных интерфейсов по передачам управления и по информации.* Эти правила формируются на базе описаний языков программирования или оформляются на основе правил структурного построения программ и базы данных конкретных проектов проектируемой системы. В последнем случае соглашения о связях конкретизируются в макрокомандах межмодульного взаимодействия.

Структурное проектирование сложных комплексов программ активно развивается на основе концепции и стандартов *открытых систем*.

Основной *принцип открытых систем* состоит в создании среды, включающей программные и аппаратные средства, службы связи, интерфейсы, форматы данных и протоколы, которая в своей основе имеет

развивающиеся, доступные и общепризнанные стандарты и обеспечивает переносимость, взаимодействие и масштабируемость приложений и данных.

Применение стандартов открытых систем следует начинать при создании *архитектуры исходных модулей* мобильных проектируемых систем (ПС) и баз данных (БД), а далее неукоснительно использовать при всех процессах жизненного цикла (ЖЦ). Во всех случаях создание архитектуры модулей и компонентов современных сложных систем целесообразно вести с использованием *профилей международных стандартов*, значительная часть которых обеспечивает *мобильность* и *возможность повторного использования* готовых программных средств и баз данных.

Основными *целями* создания и применения концепции, методов и *стандартов открытых систем* является

— повышение общей экономической эффективности разработки и функционирования систем, а также
— логической и технической совместимости их компонентов,
— обеспечение мобильности и повторного применения готовых программ и данных.

Они реализуют спецификации на интерфейсы, процессы и форматы данных, достаточные для того, чтобы обеспечить:

—возможность *расширения* проектируемой системы, а также переноса (*мобильность*) программных компонентов и систем, разработанных должным образом, с минимальными изменениями на широкий диапазон аппаратных и операционных платформ;
—*совместную работу* с другими программными продуктами и системами на локальных и удаленных платформах;
—*взаимодействие с пользователями* в стиле, облегчающем последним переход от системы к системе (*мобильность* пользователей).

Для достижения этих целей развиваются и применяются различные проблемно-ориентированные технологии и комплексы средств автоматизации жизненного цикла (ЖЦ) мобильных программ и баз данных, основанные на повторном использовании готовых апробированных модулей, программных компонентов и данных, их эффективном переносе на различные аппаратные и операционные платформы и согласованном взаимодействии в распределенных информационных системах.

Профессионалы в области открытых систем акцентируют усилия на поиске и создании гибкой, способной к наращиванию среды, что базируется на трех направлениях стандартизации в области систем:

— аппаратных и операционных платформ;
— методов и технологии обеспечения жизненного цикла прикладных программных средств и баз данных;
— интерфейсов компонентов и модулей между собой, с операционной и внешней средой.

Методы открытых систем обеспечивают эффективные по трудоемкости и качеству структурное проектирование, расширение и перенос готовых программных средств обработки информации и баз данных на различные аппаратные и операционные платформы. Эти методы можно разделить на три части:

— общая концепция и методы непосредственного обеспечения мобильности компонентов программных средств и баз данных в процессе разработки систем за счет унификации интерфейсов с операционной и аппаратной средой;
— методы, поддерживающие мобильность компонентов и комплексов программ и данных в распределенных системах и совместимость их взаимодействия с внешней средой;
— методы создания текстов программных средств, баз данных и их компонентов на стандартизированных языках программирования высокого уровня, обеспечивающие потенциальную возможность их переноса на различные аппаратные платформы.

Первая группа методов создавалась с ограниченной и определенной целью *локализовать* и *унифицировать интерфейсы* программных компонентов между собой, с заранее выделенными операционными системами и с внешней средой. Интерфейсные стандарты являются базой для обеспечения структурного проектирования, свободного перемещения и расширения программных продуктов и компонентов в различные по архитектуре и функциям операционные окружения и аппаратные платформы.

Эти методы позволяют разделить функциональную часть комплексов программ и их связи с организационным окружением, обеспечивая технологическую, архитектурную и языковую независимость функциональной части программ и данных. Тем самым они являются базой для реализации концепции открытости в системах и обеспечивают свободу разработчикам при выборе

— методов структурного проектирования и
— языков программирования

для создания компонентов программ и описаний данных. Стандарты этой группы включают группу стандартов ***POSIX***, в которой определены концепция и функции интерфейсов переносимых операционных систем, команды управления и сервисные программы, а также расширение для переносимых операционных систем.

Цель второй группы методов — поддержать мобильность программных продуктов в открытых системах путем унификации их интерфейсов с внешней средой. Они обеспечивают

— *совместимость* обмена данными в различных файловых системах и базах данных,

— *унификацию* административного управления и взаимодействия с пользователями, а также
— *унификацию* методов обеспечения безопасности и защиты информации.

Таким образом, создается стандартизированная по интерфейсам внешняя среда на различных гетерогенных аппаратных платформах, в которую могут эффективно погружаться различные программы, согласованные по интерфейсам с этими стандартами. Стандарты этой группы регламентируют функции и процессы, поддерживающие взаимодействие с внешней средой:

— концепции и архитектуру *визуализации* информации для взаимодействия с пользователями и ее представления в базовых системах графического отображения;
— архитектуру, интерфейсы, базовые процессы и языки управления файловыми системами и базами данных, обеспечивающими их *совместимость* и *унификацию* в сложных системах;
— *административное управление* локальными и распределенными компонентами систем в процессе решения функциональных задач обработки информации.

Третья группа методов создавалась в значительной степени независимо от первой и второй. Эти методы преследуют цель *унифицировать тексты* программных модулей, компонентов и описания данных, создаваемых для различных аппаратных платформ при любой их архитектуре, независимо от операционной и внешней среды. Первоначальное огромное число языков программирования (свыше 200), каждый из которых имел несколько диалектов, в результате сократилось до 6—10 массовых языков, ограниченных стандартами, не допускающими диалектов.

Создание современных модулей и компонентов проектируемой системы (ПС) и базы данных (БД) поддерживается

— методами и инструментальными средствами технологий,
— методами тестирования и аттестации программ и их интерфейсов, а также
— стандартизированным составом и содержанием документации на программы и базы данных.

Эти методы и средства обеспечивают необходимое качество модулей и компонентов сложных проектируемых системах (ПС) и базах данных (БД). В результате обеспечивается мобильность функциональной части текстов программ, однако они могут требовать доработок интерфейсов для сопряжения с новой средой аппаратного и операционного окружения систем.

Вопросы для контроля

1. В чём преимущество модульности программного продукта? Какие задачи должен решать программный модуль?

2. Какие задач ставятся перед разработчиком при проектировании программного модуля?

3. В чем состоит принцип открытых систем?

4. Что должны обеспечивать программные продукты, реализуемые с учетом стандартов открытых систем?

5. В чём состоим суть методов открытых систем для проектирования программных продуктов?

Глава 18. Принципы распараллеливания в современных многопроцессорных вычислительных системах.

Учебные цели:

- *дать представление о классификации суперкомпьютерных вычислительных систем;*
- *дать представление о принципах распараллеливания в современных системах программирования;*
- *дать представление о типах параллелизма.*

18.1. Изучение классификации суперкомпьютерных вычислительных систем.

Под архитектурой вычислительной системы понимаются абстрактное представление ЭВМ с точки зрения программиста. Полное описание архитектуры системы включает в себя:

- основные форматы представления данных;
- способы адресации данных в программе;
- состав аппаратных средств вычислительной машины, характеристики этих средств, принципы организации вычислительного процесса.

В этой главе рассмотрим современные аспекты архитектуры вычислительной системы (суперкомпьютерной вычислительной системы).

Определим структуру вычислительной системы как совокупность аппаратных средств ЭВМ с указанием основных связей между ними.

На сегодняшний день существует много различных классификаций вычислительных систем. Рассмотрим наиболее важные и, можно считать, классические классификации.

Классификация Флинна.

Наибольшее распространение получила классификация вычислительных систем, предложенная в 1966 г. профессором Стенфордского университета М. Д. Флинном (M.J.Flynn), — классификация Флинна. Эта классификация охватывает только два классификационных признака: тип *потока команд* и тип *потока данных* (см. рисунок 18.1).

В одиночном потоке команд в один момент времени может выполняться только одна команда. В этом случае эта единственная команда определяет в данный момент времени работу всех или, по крайней мере, многих устройств вычислительной системы (на всех, или многих, системах выполняется одна и та же, именно эта, единственная команда).

Рис. 18.1. Классификации Флинна. Классификация по типу потока команд.

Во множественном потоке команд в один момент времени может выполняться много команд. В этом случае каждая из таких команд определяет в данный момент времени работу только одного или лишь нескольких (но не всех) устройств вычислительной системы.

Таким образом, в одиночном потоке последовательно выполняются отдельные команды, во множественном потоке — группы команд.

Рис. 18.2. К классификации Флинна. Классификация по типу потока данных.

Одиночный поток данных обязательно предполагает наличие в вычислительной системе только *одного устройства оперативной памяти и одного процессора*. Однако при этом процессор может быть как угодно сложным, так что процесс последовательной обработки каждой единицы информации в единицу времени в потоке может требовать выполнения разных (и многих) команд.

Множественный поток данных состоит из многих зависимых или независимых одиночных потоков данных.

В соответствии со сказанным, все вычислительные системы делятся на четыре типа:

- ***SISD (ОКОД);***
- ***MISD (МКОД);***
- ***SIMD (ОКМД);***
- ***MIMD (МКМД).***

Вычислительная система ***SISD*** представляет собой классическую однопроцессорную ЭВМ фон-неймановскую архитектуру.

На вычислительную систему ***MISD*** существуют различные точки зрения. По одно них — за всю историю развития вычислительной техники системы ***MISD*** не были созданы. По другой точке зрения (менее распространенной, чем первая) к ***MISD***-системам относятся векторно-конвейерные вычислительные системы. Примем для определенности первую точку зрения.

Вычислительная система ***SIMD*** содержит много процессоров, которые *синхронно* (как правило) выполняют одну и ту же команду над разными данными. В свою очередь, системы ***SIMD*** делятся на два больших класса:

— векторно-конвейерные вычислительные системы;
— векторно-параллельные вычислительные системы или матричные вычислительные системы.

Вычислительная система ***MIMD*** содержит много процессоров, которые (как правило, *асинхронно*) выполняют разные команды над разными данными. Подавляющее большинство современных супер-ЭВМ имеют архитектуру ***MIMD*** (по крайней мере, на верхнем уровне иерархии). Системы ***MIMD*** часто называют многопроцессорными системами.

Рассмотренная классификации Флинна позволяет по принадлежности компьютера к классу ***SIMD*** или ***MIMD*** сделать сразу понятным базовый принцип его работы. Часто этого бывает достаточно. Недостатком классификации Флинна является "переполненность" класса ***MIMD***, так как он содержит всевозможные многопроцессорные системы

Классификация по типу строения оперативной памяти.

По типу строения оперативной памяти системы разделяются на

— системы с *общей* (*разделяемой*) памятью,
— системы с *распределенной* памятью и
— системы с физически распределенной, а логически общедоступной памятью (*гибридные* системы).

В вычислительных системах с общей памятью (*Common Memory Systems* или *Shared Memory Systems*) значение, записанное в память одним из процессоров, напрямую доступно для другого процессора. *Общая* память обычно имеет высокую *пропускную способность* памяти (*bandwidth*) и низкую *латентность* памяти (*latency*) при передаче информации между процессорами, но при условии, что *не происходит* одновременного обращения нескольких процессоров к одному и тому же элементу памяти.

Доступ к *общей* памяти разных процессоров системы осуществляется, как правило, *за одинаковое время*. Поэтому такая память называется еще ***UMA***-памятью (*Unified Memory Access*) — памятью с одинаковым временем доступа. Система с такой памятью носит название вычислительной системы с *одинаковым временем доступа к памяти*. Системы с общей памятью называются также *сильно связанными* вычислительными системами.

В вычислительных системах с распределенной памятью (*Distributed Memory Systems*) каждый процессор имеет свою локальную память с локальным адресным пространством. Для систем с распределенной памятью характерно наличие большого числа быстрых каналов, которые связывают отдельные части этой памяти с отдельными процессорами.

Обмен информацией между частями распределенной памяти осуществляется обычно относительно медленно. Системы с распределенной памятью называются также *слабо связанными* вычислительными системами.

Вычислительные системы с *гибридной* памятью — (*Non-Uniform Memory Access Systems*) имеют память, которая физически распределена по различным частям системы, но логически разделяема (образует единое адресное пространство). Такая память называется еще *логически общей* (*разделяемой*) памятью (*logically shared memory*). В отличие от ***UMA***-систем, в ***NUMA***-системах время доступа к различным частям оперативной памяти различно.

Заметим, что память современных параллельных систем является многоуровневой, иерархической, что порождает проблему ее когерентности.

Классификация по типу коммуникационной сети.

Классификация параллельных вычислительных систем по типу коммуникационной сети процессоров очень похожа на классификацию связей в коммуникационной сети. Заметим лишь, что по количеству уровней иерархии коммуникационной среды процессоров различают системы с одноуровневой коммутационной сетью (один уровень коммутации) и системы с иерархической коммутационной сетью процессоров (когда группы процессоров объединены с помощью одной системы коммутации, а внутри каждой группы используется другая).

Классификация по степени однородности

По степени однородности различают однородные (*гомогенные*) и неоднородные (*гетерогенные*) вычислительные системы. Обычно при этом имеется в виду тип используемых процессоров.

В однородных вычислительных системах (гомогенных вычислительных системах) используются *одинаковые процессоры*, в неоднородных вычислительных системах (гетерогенных вычислительных системах) – процессоры *различных типов*. Вычислительная система, содержащая какой-либо специализированный вычислитель (например, *Фурье*-процессор), относится к классу неоднородных вычислительных систем.

В настоящее время большинство высокопроизводительных систем относятся к классу *однородных* систем с *общей памятью* или к классу *однородных* систем с *распределенной* памятью.

Рассмотренные классификационные признаки параллельных вычислительных систем не исчерпывают всех возможных их характеристик. Существует, например, еще разделение систем

- по *степени согласованности режимов работы* (синхронные и асинхронные вычислительные системы),
- по *способу обработки* (с пословной обработкой и ассоциативные вычислительные системы),

— по *жесткости структуры* (системы с фиксированной структурой и системы с перестраиваемой структурой),
— по *управляющему потоку* (системы потока команд (*instruction flow*) и системы потока данных (*data flow*) и тому подобное.

Современные высокопроизводительные системы имеют, как правило, иерархическую структуру. Например, на верхнем уровне иерархии система относится к классу ***MIMD***, каждый процессор которой представляет собой систему ***MIMD*** или систему ***SIMD***.

Отметим также тенденцию к построению *распределенных систем с программируемой структурой*. В таких системах нет общего ресурса, отказ которого приводил бы к отказу системы в целом — средства управления, обработки и хранения информации распределены по составным частям системы. Такие системы обладают способностью автоматически реконфигурироваться в случае выхода из строя отдельных их частей. Средства реконфигурирования позволяют также программно перестроить систему с целью повышения эффективности решения на этой системе данной задачи или класса задач.

18.2. Понятие о конвейере, матричном процессоре, структуре мультипроцессорных и мультикомпьютерных вычислительных систем.

Векторно-конвейерные системы и векторно-параллельные (SIMD-системы)

Векторно-конвейерные вычислительные системы относятся к классу ***SIMD***-систем. Основные принципы, заложенные в архитектуру векторно-конвейерных систем:

— конвейерная организация обработки потока команд;
— введение в систему команд набора векторных операций, которые позволяют оперировать с целыми массивами данных.

Длина обрабатываемых векторов в современных векторно-конвейерных системах составляет, как правило, 128 или 256 элементов. Основное назначение векторных операций состоит в распараллеливании выполнения операторов цикла, в которых обычно сосредоточена большая часть вычислительной работы.

Первый векторно-конвейерный компьютер ***Cray-1*** появился в 1976 году. Архитектура этого компьютера оказалась настолько удачной, что он положил начало целому семейству компьютеров.

Современные векторно-конвейерные системы имеют иерархическую структуру:

— на нижнем уровне иерархии расположены конвейеры операций (например, конвейер (*pipeline*) сложения вещественных чисел, конвейер умножения таких же чисел и тому подобное);

— некоторая совокупность конвейеров операций объединяется в конвейерное функциональное устройство;
— векторно-конвейерный процессор содержит ряд конвейерных функциональных устройств;
— несколько векторно-конвейерных процессоров (от 2 до 16), объединенных общей памятью, образуют вычислительный узел;
— несколько таких узлов объединяются с помощью коммутаторов, образуя либо ***NUMA***-систему либо ***MPP***-систему.

Типичными представителями такой архитектуры являются классические компьютеры ***CRAY J90/T90, CRAY SV1, NEC SX-4/SX-5***. Уровень развития микроэлектронных технологий не позволяет в настоящее время производить однокристальные векторно-конвейерные процессоры, поэтому эти системы довольно громоздки и чрезвычайно дороги.

Каждая часть конвейера операций называется ступенью конвейера операций, а общее число ступеней — длиной конвейера операций.

Пример 18.1. Конвейер операций сложения вещественных чисел

Рассмотрим следующий 4-х ступенчатый конвейер операций сложения вещественных чисел $x = e2^p$, $y = f2^q$ (таблица 18.1):

Таблица 18.1

Номер ступени	Наименование
1	Вычитание порядков ***p - q***
2	Сдвиг одной из мантисс ***e, f***
3	Сложение мантисс
4	Нормализация

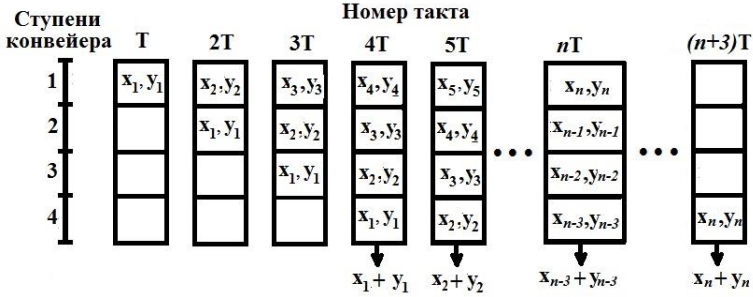


Рис. 18.3. К примеру 18.1. Временная диаграмма сложения ***(n*1)***-векторов вещественных чисел ***x, y*** на 4-х ступенчатом конвейере операции сложения.

Положим, что выполняется сложение двух ***(n*1)***-векторов вещественных чисел $X = (x_1, x_2, \ldots, x_n)$, $Y = (y_1, y_2, \ldots, y_n)$. Диаграмма сложения этих векторов приведена на рисунке 18.3.

В векторно-конвейерных системах в рамках одного конвейерного функционального устройства широко используется (то есть аппаратно поддерживается) зацепление конвейеров операций.

Покажем суть этой процедуры на примере.

Пример 18.2. Зацепление конвейеров

Положим, что в некоторой прикладной программе, исполняемой на векторно-конвейерной системе, необходимо вычислить

$$E = \frac{(A + B) * C}{D} \qquad (18.1)$$

где ***A, B, C, D*** — (***n*1***)-векторы вещественных чисел. Под произведением и делением векторов понимается их покомпонентное умножение и деление, соответственно. Иными словами, операции, указанные в выражении (1), понимаются в смысле

$$e_i = \frac{(a_i + b_i) * c_i}{d_i}, \quad i \in [1, n] \qquad (18.2)$$

Положим также, что конвейерное функциональное устройство данной векторно-конвейерной системы имеет следующие конвейеры операций:

— конвейер сложения вещественных чисел;
— конвейер умножения вещественных чисел;
— конвейер деления вещественных чисел

Тогда для повышения скорости вычисления компонент вектора ***E*** целесообразно использовать *зацепление* указанных конвейеров (рис. 18.2). В результате, можно сказать, получается новый конвейер, который выполняет сложную операцию (рис. 18.4):

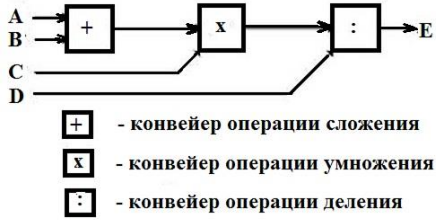


Рис. 18.4. К примеру 18.2. К зацеплению конвейеров.

Конвейер операций не следует путать с *конвейером команд*, в котором при исполнении одной команды готовится к исполнению несколько следующих более мелких команд. Также, как в конвейере операций, каждая часть конвейера команд называется ступенью конвейера команд, а общее число ступеней — длиной конвейера команд. Конвейеры команд широко используются в современных процессорах. Так, процессор ***Intel 486*** уже имел пяти ступенчатый конвейер выполнения целочисленных команд, ступенями которого являлись следующие операции:

— предвыборка (команда извлекается из кэш-памяти и размещается в одном из двух 16-байтовых буферах);
— декодирование;
— генерация адреса;
— исполнение в арифметико-логическом устройстве (АЛУ);
— запись результата в КЭШ-память.

Процессор ***Pentium 2*** (*суперскалярная* архитектура) имел уже два 8-ми ступенчатых конвейера целочисленных команд.

Кроме конвейеров в векторно-конвейерных системах для ускорения работы используют различные механизмы адресации, операции с автоинкрементом (автодекрементом) адреса, механизмы ускоренной выборки и записи (многопортовая память, память с расслоением и так далее), отдельное адресное обрабатывающее устройство, отдельное скалярное устройство для выполнения скалярных операций и прочее.

Недостатком векторно-конвейерных систем является невысокая загрузка процессорных элементов. Высокая производительность достигается только на операциях с длинными векторами. На скалярных операциях и при обработке векторов и матриц невысокой размерности значительная часть устройств может простаивать. В целом, векторно-конвейерные системы характеризуются высокой производительностью *при полной загрузке их вычислительных устройств*, которая имеет место только при решении определенного, достаточно узкого, круга задач.

В качестве классического примера векторно-конвейерной системы приведем легендарную супер-ЭВМ ***CYBER-205*** фирмы ***CDC. CYBER-205*** имел следующие конвейерные функциональные устройства:

— одно конвейерное функциональное устройство «скалярных» операций с конвейерами
  - сложения (5-ти ступенчатый);
  - умножения (5-ти ступенчатый);
  - логических операций сложения (3-х ступенчатый);
  - цикла;
  - извлечения корня;
  - деления;
— 1, 2 или 4 конвейерных функциональных устройства «векторных» операций с конвейерами
  - сложения;
  - умножения;
  - сдвига;
  - логических операций;
  - задержки.

В качестве примера современной супер-ЭВМ, использующей векторно-конвейерные процессоры, приведем японскую систему ***Fujitsu-VPP5000***. На верхнем уровне ***Fujitsu-VPP5000*** имеет ***MPP*** архитектуру.

Производительность одного процессора системы составляет 9.6 Гфлопс, пиковая производительность системы может достигать 1249 Гфлопс, максимальная емкость памяти — 8 Тб. Система масштабируется до 512 узлов.

Векторно-параллельные системы.

Как и векторно-конвейерные системы, векторно-параллельная вычислительная система обычно имеет иерархическую структуру. На нижнем уровне иерархии находятся векторно-параллельные процессоры, представляющие собой совокупность скалярных процессоров (процессорных элементов), которые объединены некоторой коммуникационной сетью и в каждом такте синхронно выполняют одну и ту же команду над разными данными. На верхнем уровне иерархии векторно-параллельные процессоры объединяются общей памятью или коммуникационной сетью, образуя ***NUMA***-систему либо ***MPP*** систему.

Векторно-параллельные процессоры имеют в своих системах команд специальные векторные (*матричные*) операции, такие, как векторное и матричное сложение, умножение вектора на матрицу, умножение матрицы на константу, вычисление скалярного произведения, свертки и так далее.

При выполнении векторных операций различные компоненты векторов и матриц обрабатываются *параллельно* на различных процессорных элементах.

Основными компонентами векторно-параллельного процессора являются

- совокупность скалярных процессоров (P);
- совокупность модулей оперативной памяти (M);
- коммуникационная среда;
- устройство общего управления.

Выделим две группы векторно-параллельных процессоров:

- процессоры с одинаковым числом скалярных процессоров и модулей памяти;
- векторные процессоры с различным количеством скалярных процессоров и модулей памяти.

В векторно-параллельном процессоре с одинаковым числом скалярных процессоров и модулей памяти каждый скалярный процессор подключается к своему модулю памяти (рис. 18.5). Команда, выдаваемая устройством управления, содержит одинаковый адрес для всех скалярных процессоров. С помощью специального «флага» можно запретить выполнение команды на данном скалярном процессоре – «маскирование команды».

Рис. 18.5. Структура векторно-параллельного процессора с одинаковым числом скалярных процессоров и модулей памяти.

В векторно-параллельном процессоре с различным количество скалярных процессоров и модулей памяти (рис.18.6) основной проблемой является *проблема исключения конфликтов* при обращении к памяти (поскольку к одному модулю памяти могут одновременно обращаться в переделе все скалярные процессоры). Для преодоления этой проблемы в системах этого класса используют изощренные схемы хранения массивов данных.

Рис. 18.6. Структура векторно-параллельного процессора с различным количеством скалярных процессоров и модулей памяти.

В историческом плане наиболее известной векторно-параллельной системой является ***ILLIAC-IV*** (*Burroughs*). ЭВМ имела 64 процессора, объединенных в 8*8 плоскую решетку. Пиковая производительность равнялась 100 Мфлопс.

Недостатком векторно-параллельных систем, как и векторно-конвейерных систем, является низкая, как правило, *загрузка процессорных элементов*. Высокая производительность векторно-параллельных систем достигается только на векторных операциях, в то время как на скалярных операциях, а также при обработке векторов и матриц небольшой размерности, значительная часть процессорных элементов может простаивать.

Программирование векторно-параллельных систем сложнее, чем векторно-конвейерных систем. В целом, векторно-параллельные системы, как и векторно-конвейерные системы, характеризуются высокой производительностью только при полной загрузке их вычислительных устройств, которая достигается при решении достаточно узкого класса задач.

В качестве современных векторно-параллельных систем приведем примеры суперкомпьютерных вычислительных систем из рейтинга TOP500. Первое место в этом рейтинге в 2020 году занимал японский суперкомпьютер *Supercomputer Fugaku A64FX48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan.* Общее количество вычислительных ядер — *7 299 072.* Пиковая производительность Rpeek (Tflop/s) — 513 854,7.

Второе место в 2020 занимал американский суперкомпьютер ***Summit*** (*IBM Power System AC922, IBM POWER9 22C 3.07*GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband). Общее количество вычислительных ядер — 2 414 592. Пиковая производительность Rpeek (Tflop/s) — 200 794,88.

Третье место занимал американский суперкомпьютер ***Sierra*** с похожей архитектурой (*IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband*). Общее количество вычислительных ядер — 1 572 480. Пиковая производительность Rpeek (Tflop/s) — 125 712.

На четвертое место сместился китайский суперкомпьютер ***Sunway TaihuLight*** (*Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway*), который был запущен на два года раньше американских «конкурентов» и довольно долго лидировал в рейтинге. Общее количество вычислительных ядер — 10 649 600. Пиковая производительность Rpeek (Tflop/s) — 125 435,9.

Эксперты назвали этот суперкомпьютер Fugaku лучшим не в одной, а сразу в четырех категориях:

- самых мощных;
- использование в промышленности;
- использование в сфере искусственного интеллекта;
- в области анализа больших цифр.

Примечательно, что до этого ни один компьютер в мире не занимал первое место сразу в четырех номинациях из шести. Суперкомпьютер из Японии стал первым в рейтинге впервые за девять лет после своего предшественника компьютера K.

18.3. Представление о типах параллелизма.

Известно громадное количество последовательных (традиционных) методов и алгоритмов решения задач. «Приспособление» этих методов алгоритмов для их реализации на параллельных вычислительных системах носит название «*распараллеливание*» (*paralleling*). Отметим особую сложность этой задачи для вычислительных систем с распределенной памятью.

Однако чаще необходимо специально разрабатывать и обосновывать параллельные алгоритмы. Решение этих задач является предметом новой и бурно развивающейся отрасли вычислительной математики — параллельной вычислительной математики. Результатами параллельной вычислительной математики являются как эффективные параллельные алгоритмы для решения конкретных задач, так и целые классы принципиально новых алгоритмов, ярким представителем которых являются нейросетевые алгоритмы. Одним из важных результатов параллельной вычислительной математики является установление того факта, что многие традиционные последовательные алгоритмы не имеют эффективных параллельных аналогов.

Для последовательных вычислений лучший вычислительный алгоритм тот, который имеет минимум арифметических операций, особенно трудоемких (при прочих равных условиях). Однако для параллельных вычислений важным является также учет сложности межоператорных связей по данным и управлению. Поэтому актуальной задачей является разработка параллельных алгоритмов, которые имеют эффективное отображение на архитектуры параллельных вычислительных систем.

Определим более строго *распараллеливание* как *нахождение параллельного алгоритма решения задачи и реализация этого алгоритма на параллельной вычислительной системе*. Отметим, что распараллеливание не обязательно предполагает получение параллельного алгоритма из последовательного алгоритма. В таком случае, будем понимать термин «распараллеливание» как синоним термина «*синтез параллельного алгоритма*».

В таком случае, с точки зрения уровней абстракции, кроме распараллеливания на уровнях метода и алгоритма, можно говорить о распараллеливании на уровне математической модели задачи, а также на уровне программы, реализующей используемый алгоритм.

Таким образом, распараллеливание решения любой задачи многовариантно: *для одной и той же математической модели можно построить различные параллельные алгоритмы* и для каждого из этих алгоритмов — различные параллельные программы. Многовариантность оказывается настолько важной характеристикой, что одной из

центральных проблем в области параллельных вычислений является проблема *оценки эффективности алгоритмов* для данной параллельной вычислительной системы или для класса таких систем.

Всякий алгоритм имеет две характеристики, по которым можно судить о его качестве:

— *сложность* алгоритма (требуемое время на вычисления);
— *численная устойчивость* (малая чувствительность к ошибкам в исходных данных и ошибкам округления).

Приведем определение понятий сложности алгоритма и численной устойчивости, принятое в теории алгоритмов.

Вычисли́тельная сло́жность — понятие, обозначающее *функцию зависимости объёма работы, которая выполняется некоторым алгоритмом, от размера и структуры входных данных*; зависит от сложности операторов, составляющих алгоритм, объёма входных данных и их, например, упорядоченности, и влияет на время выполнения алгоритма.

Алгоритм называется *численно устойчивым, если ошибки округления не превосходят ошибок в исходных данных.*

Проблема обеспечения устойчивости параллельных алгоритмов к погрешностям в исходных данных и вычислительным погрешностям при построении параллельных алгоритмов является более сложной и актуальной, чем в традиционных последовательных алгоритмах. В целом, устойчивость параллельных алгоритмов к ошибкам округления хуже, чем устойчивость соответствующих последовательных алгоритмов.

Введём понятие *зернистости* или *гранулированности* параллельной программы, как меры степени ее параллелизма. *Зернистость* параллельной программы определяется объемом элементарных процессов этой программы, где под элементарным процессом понимается один или совокупность последовательно выполняемых операторов в виде блоков, подпрограмм, процедур, функций, задач и прочее. Мелкозернистая параллельная программа содержит элементарные процессы преимущественно небольшого объема, крупнозернистая параллельная программа — процессы преимущественно большого объема.

Наряду с зернистостью программы, говорят о зернистости алгоритма или степени грануляции алгоритма, как мере степени параллелизма алгоритма. Мелкозернистый алгоритм позволяет выделить в нем параллельные ветви только малого объема. Напротив, крупнозернистый алгоритм, адекватный ***MIMD***-системам, состоит из независимых или слабо связанных ветвей значительного объема, которые могут обрабатываться параллельно. Крупнозернистый алгоритм часто называют крупноблочным алгоритмом.

Задача, для которой известны только мелкозернистые алгоритмы ее решения, называется *сильно связанной* задачей. Напротив, задача, для

которой известен хотя бы один крупно крупнозернистый алгоритм ее решения, называется *слабо связанной* задачей.

Параллельные алгоритмы, ориентированные на ***SIMD***-вычислительные системы и ***MIMD***-вычислительные системы, существенно различны.

В данном параграфе рассматриваются методы и алгоритмы, ориентированные на ***MIMD***-системы с общей памятью или на ***MIMD***-системы с распределенной памятью.

В задачах можно выделить следующие типы параллелизма:

- параллелизм данных;
- функциональный параллелизм;
- геометрический параллелизм;
- алгоритмический параллелизм;
- конвейерный параллелизм;
- «беспорядочный» параллелизм.

Параллелизм данных

Параллелизмом данных обладают задачи, которые включают в себя неоднократное выполнение одного и того же алгоритма с различными исходными данными. Такие вычисления могут, очевидно, производиться параллельно.

Если задача обладает параллелизмом данных, то соответствующую параллельную программу (***SPMD***-программу) целесообразно организовать в виде совокупности одинаковых программ, каждая из которых выполняется на своем подчиненном процессоре, и из основной программы, которая выполняется на мастер-процессоре (рис. 18.7). Такая программа является, как правило, крупнозернистой.

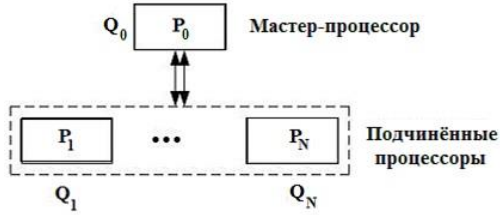


Рис. 18.7. К распараллеливанию на основе параллелизма данных. Q_0 — основная программа, $Q_i, i \in [1,N]$ — одинаковые подчиненные программы.

Распараллеливание на основе параллелизма данных называется *декомпозицией по данным*.

Рассмотрим простейший пример декомпозиции по данным.

Пусть на ***N*** процессорной ***MIMD***-вычислительной системе необходимо выполнить сложение двух (***n*1***)- векторов ***A, B*** и получить (***n*1***)-вектор ***C***. Тогда декомпозиция по данным состоит в распределении ***n/N*** элементов векторов ***A, B***, по процессам $Q_1, Q_2, \ldots, Q_N$, выполняющимся на

процессорах $P_1, P_2, \ldots, P_N$ и вычислении соответствующих элементов результирующего вектора C.

Заметим, что возможны задачи, обладающие параллелизмом данных, когда данные не распределяются мастер-процессом по подчиненным процессам, а генерируются самими подчиненными процессами (например, используя генераторы случайных чисел) или вводятся ими из внешних устройств.

Декомпозиция по данным может быть *статической* и *динамической*. При статической декомпозиции по данным каждому процессу заранее назначается его доля данных. При динамической декомпозиции по данным мастер-процесс распределяет блоки данных по процессам в ходе решения задачи по мере появления данных и освобождения соответствующих процессоров.

Функциональный параллелизм

Функциональный параллелизм — это параллелизм групп операций, объединенных по функциональному признаку. Распараллеливание на основе функционального параллелизма называется функциональной декомпозицией. Тривиальным примером функциональной декомпозиции является декомпозиция задачи на три следующих подзадачи: ввод исходных данных, обработка, вывод результатов, визуализация результатов. Параллелизм при этом достигается параллельным выполнением трех указанных подзадач и созданием «конвейера» (последовательного или последовательно параллельного) между ними. Заметим, что каждая из указанных подзадач может обладать параллелизмом данных. Приведем еще один (менее очевидный) пример функциональной декомпозиции. Задачу математического моделирования самолета можно разбить на подзадачи моделирования атмосферы, фюзеляжа самолета, его левого крыла, правого крыла и так далее.

Важно следующее обстоятельство: при функциональной декомпозиции задачи число используемых процессоров определяется числом подзадач. Увеличить число процессоров с целью увеличения скорости решения задачи при таком подходе затруднительно. Фактически, программа, использующая функциональный параллелизм, не является масштабируемой.

Заметим также, что функциональная декомпозиция задачи естественным образом используется при построении отказоустойчивых систем.

Наибольший интерес представляют масштабируемые алгоритмы, которые преимущественно и рассматриваются в данном параграфе.

Геометрический параллелизм

Геометрическим параллелизмом обладают, например, многие физические задачи, которые описываются дифференциальными уравнениями в частных производных (задачи механики сплошной среды, теории поля и тому подобное.). Такие задачи обычно решаются конечно-разностными методами или методами конечных элементов. Дискретные аналоги этих задач имеют локально-ограниченные взаимодействия между

узлами сетки, покрывающей область решения, что позволяет разбить область решения задачи на подобласти и вычисления в каждой из подобластей поручить отдельному процессору (рис. 18.8).

Распараллеливание на основе геометрического параллелизма называется декомпозиция области решения. Декомпозиция области решения – один из основных способов получения крупнозернистых алгоритмов и программ.

Рис. 18.8. К распараллеливанию на основе декомпозиции области решения Ω на подобласти Ω_i, $i \in [1,N]$.

Отличие геометрического параллелизма от параллелизма по данным состоит в том, что подзадачи обработки каждой из подобластей $\boldsymbol{\Omega_i}$, $\boldsymbol{i \in [1, N]}$ взаимосвязаны (требуется обмен данными между этими подзадачами).

Декомпозиция области решения эффективна при условии, что вычислительная сложность каждой из программ $\boldsymbol{Q_i}$, $\boldsymbol{i \in [1, N]}$ примерно одинакова (для однородной вычислительной системы). Кроме того, для эффективности декомпозиции области решения программа, выполняемая процессором $\boldsymbol{P_i}$, должна использовать лишь небольшой объем данных, расположенных на других процессорах. Желательно, что бы эти не локальные данные были расположены на небольшом количестве соседних процессоров. Такое свойство называется локальностью алгоритма.

Можно выделить статическую декомпозицию области решения и динамическую декомпозицию области решения. Если вычислительные сложности подзадачи обработки каждой из подобластей $\boldsymbol{\Omega_i}$, $\boldsymbol{i \in [1, N]}$ меняются в процессе вычислений, то статическая декомпозиция области решения может оказаться малоэффективной. В этом случае может быть целесообразной динамическая декомпозиция области решения, при которой границы между подобластями $\boldsymbol{\Omega_i}$, $\boldsymbol{i \in [1, N]}$ меняются в процессе вычислений. Такая ситуация может иметь место, например, при решении задач механики сплошной среды на адаптивных сетках.

Алгоритмический параллелизм

Алгоритмическим параллелизмом называется параллелизм, который выявляется путем выделения в данном алгоритме тех фрагментов, которые могут выполняться параллельно (рис. 18.9). Алгоритмический параллелизм редко порождает крупнозернистые параллельные алгоритмы и программы.

Синтез параллельных алгоритмов (распараллеливание) на основе алгоритмического параллелизма называется *алгоритмической декомпозицией*. При использовании алгоритмической декомпозиции следует стремиться к разбиению задачи на крупные и редко взаимодействующие ветви. Должно быть обеспечено, по возможности, однородное распределение массивов по ветвям параллельного алгоритма.

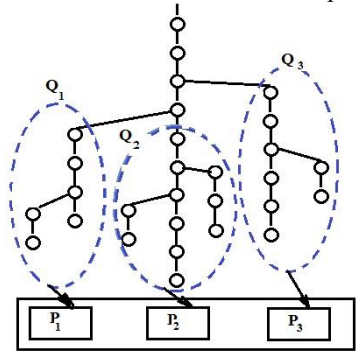


Рис. 18.9. К использованию алгоритмического параллелизма. ***Q1, Q2, Q3*** — ветви алгоритма. Кружки обозначают операторы алгоритма.

Отличие алгоритмического параллелизма от функционального состоит в том, что функциональный параллелизм предполагает объединение только функционально близких операторов алгоритма, в то время как алгоритмический параллелизм функциональную близость операторов не учитывает.

Конвейерный параллелизм

Типичный пример *конвейерного* параллелизма представляет собой параллелизм систем обработки видеоинформации. В таких системах множество изображений должны проходить несколько этапов обработки: например, ввод данных, обработка данных и вывод результатов. В данном случае естественно использовать конвейерную декомпозицию задачи, когда каждый этап обработки выполняется на отдельном процессоре.

Конвейерная декомпозиция имеет весьма ограниченное применение, поскольку весьма редко удается организовать достаточно длинный конвейер и обеспечить равномерную загрузку всех процессоров.

Отметим значительную общность конвейерного и функционального параллелизма.

«Беспорядочный» параллелизм

В некоторых классах алгоритмов возможное количество параллельных ветвей и их вычислительная сложность априори неизвестны, а определяются особенностями конкретной задачи. Примерами алгоритмов, обладающих «беспорядочным» параллелизмом, является алгоритм «ветвей и границ»», рекурсивные алгоритмы типа «разделяй и властвуй» и прочее.

В этом случае новые ветви по мере их возникновения передаются управляющим процессором обрабатывающим процессорам в соответствии с их текущей загрузкой. При каждом таком акте необходимо в той или иной форме решать задачу балансировки загрузки процессоров, что влечет за собой значительные накладные расходы.

Реструктуризация данных

Широко известным частным случаем *алгоритмической декомпозиции* является реструктуризация данных.

Рассмотрим задачу вычисления произведения $n=8$ вещественных чисел $a_1, a_2, \ldots, a_8$. В простейшем случае параллельное вычисление этого произведения можно организовать в соответствии со следующей ярусно параллельной формой (ЯПФ). Такие вычисления можно выполнить и на одном процессоре последовательно. При этом высота (количество тактов — $h_{ЯПФ}$) и ширина (максимальное количество процессоров, участвующих в вычислениях — $w_{ЯПФ}$) ярусно параллельной формы (ЯПФ) составляют $h_{ЯПФ}=7$, $w_{ЯПФ}=1$.

Ярус 1. Вычисление произведения a_1a_2;

Ярус 2. Вычисление произведения $(a_1a_2)a_3$;

…

Ярус 7. Вычисление произведения $(a_1a_2a_3a_4a_5a_6a_7)a_8$.

Если для вычислений используется $N > n$ процессоров, то при использовании такого алгоритма в каждый момент времени простаивают все процессоры, кроме одного.

Более эффективным алгоритмом решения данной задачи является *алгоритм сдваивания*, применимый к любой ассоциативной операции ($h_{ЯПФ}=3$, $w_{ЯПФ}=4$):

Ярус 1. Вычисление произведений a_1a_2; a_3a_4; a_5a_6; a_7a_8;

Ярус 2. Вычисление произведений $(a_1a_2)a_3$; $(a_1a_2)(a_3a_4)$; $(a_5a_6)a_7$; $(a_5a_6)(a_7a_8)$;

Ярус 3. Вычисление произведения $(a_1a_2a_3a_4)a_5$; $(a_1a_2a_3a_4)(a_5a_6)$; $(a_1a_2a_3a_4)(a_5a_6a_7)$; $(a_1a_2a_3a_4)(a_5a_6a_7a_8)$

Данный алгоритм обеспечивает полную загрузку каждого из 4-х процессоров на всех шагах вычислений. Синим цветом выделены лишние числа для вычисления произведения Хотя в некоторых случаях эти промежуточные произведения бывают очень полезны.

Известно утверждение: *алгоритм сдваивания для вычисления произведения* (или любой другой ассоциативной операции) $M=2^m$ *элементов равен* $a=\prod_{i=1}^{M} a_i$ и обеспечивает решение задачи за $O(\log M)$ на энергонезависимой памяти PRAM с $N=\frac{M}{\log M}$ процессорами.

Замечание. В силу ассоциативности операции умножения, рассмотренные алгоритмы дают одинаковые результаты в условиях точных вычислений. Однако, в реальных условиях вычислений с погрешностями округления эти алгоритмы, вообще говоря, дают разные результаты.

Вопросы для контроля

1. Дайте характеристику каждому классу вычислительных систем по классификации Флинна.
2. Что представляет собой конвейерное функциональное устройство? Приведите пример.
3. По каким характеристикам сегодня судят о качестве алгоритма? Объясните кратко в чём суть этих характеристик.
4. Назовите и кратко охарактеризуйте типы параллелизма.
5. Объясните, в чём суть алгоритма сдваивания.

Глава 19. Способы передачи данных в параллельных системах с различной организацией памяти.

Учебные цели:

- *дать представление о приоритетах, прерываниях и резидентных программах;*
- *дать представление о принципах синхронного и асинхронного выполнения параллельных подпроцессов;*
- *дать представление о способах передачи данных между параллельными подпроцессами.*

Процесс — это некоторая часть работы операционной системы (ОС), обладающая уникальным идентификационным номером — ***id***, и её адресное пространство. Адресное пространство — некоторый список адресов в памяти, с которыми происходит работа этого процесса. С другими адресами процессу приходится работать через системный вызов. Одна программа может включать как несколько процессов, так и один, причем последнее используется наиболее часто (в нашем случае динамически исполняемая программа пользователя, представляющая задачу или задачи).

Разбиение на процессы позволяет распараллелить задачи, благодаря чему ускорить работу, но в большинстве случаев для этого проще и выгоднее использовать потоки, которые намного быстрее взаимодействуют друг с другом и обладают рядом других положительных моментов, — это приводит к меньшей используемости процессов.

Поток — это часть уже самого процесса, выполняющая определенный список действий. У каждого процесса есть как минимум один поток, увеличение количества потоков обеспечивает распараллеливание процесса.

Объясним, в чём выгода увеличения потоков внутри процесса по сравнению с увеличением количества самих процессов. Каждый поток, как часть процесса, имеет доступ ко всему адресному пространству процесса, ко всем его устройствам и переменным. Поэтому взаимодействие двух отдельных потоков реализуется очень просто и не требует обращения к системе, а для взаимодействия двух процессов требуется обращение через системный вызов. Поэтому параллельное использование потоков более распространено, чем процессов.

Сам поток представляет собой стек команд со счетчиком, обладающий несколькими важными свойствами, такими как состояние и приоритет. Состояний потока всего три:

- состояние *активности*, то есть поток выполняется на данный момент,
- состояние *неактивности*, когда поток ожидает выделения процессора для перехода в состояние активности,
- состояние *блокировки*, когда потоку не выделяется время (соответственно он не занимает место в очереди, освобождая ресурсы) вне зависимости от его приоритета.

19.1. Понятие о приоритетах, прерываниях и резидентных программах.

Для определения порядка выполнения потоков диспетчер системы использует систему приоритетов.

Класс процессов (задач), имеющих самые высокие значения приоритета называется *критическим*. Этот класс предназначается для процессов, которые обычно являются процессами реального времени, то есть для них должен быть обязательно предоставлен определенный минимум процессорного времени.

Следующий класс процессов называется *приоритетным*. К этому классу относят процессы, которые выполняют по отношению к остальным задачам роль сервера. Приоритет таких процессов должен быть выше обычных задач: это будет гарантировать, что запрос на некоторую функцию от обычных процессов выполнится сразу, а не будет стоять в очереди и дожидаться, пока до него дойдет очередь на выполнение среди других пользовательских приложений.

Большинство процессов относят к обычному классу, его еще называют *регулярным, стандартным* или *пользовательским*. По умолчанию система программирования порождает процесс, относящийся именно к этому классу.

Кроме перечисленных, существует еще один класс процессов, который называется *фоновым*. К фоновым задачам относятся программы, которые получают процессорное время только тогда, когда нет задач из других классов, которым сейчас нужен процессор (к примеру, это счётные программы).

При выполнении программ, реализующих какие-либо задачи контроля и управления (это характерно, прежде всего, для систем реального времени), может случиться такая ситуация, когда одна или несколько задач не могут быть реализованы (выполнены) в течение длительного промежутка времени из-за возросшей нагрузки в вычислительной системе. Потери, связанные с невыполнением таких задач, могут оказаться больше, чем потери от невыполнения программ с более высоким приоритетом. При этом оказывается целесообразным временно изменить приоритет «аварийных» задач (для которых истекает отпущенное для них время обработки). После выполнения «аварийных» задач их приоритет восстанавливается. Поэтому почти в любой операционной системе реального времени (ОСРВ) имеются средства для изменения приоритета программ. Есть такие средства и во многих операционных системах, которые не относятся к классу ОСРВ. Введение механизмов динамического изменения приоритетов позволяет реализовать более быструю реакцию системы на *короткие* запросы пользователей — это очень важно при *интерактивной* работе, — но при этом гарантировать выполнение любых запросов в обычном режиме.

В *Windows* каждый поток (тред, сред, нить — ***thread***[31]) имеет базовый уровень приоритета, который лежит в диапазоне от двух уровней ниже базового приоритета процесса, его породившего, до двух уровней выше этого приоритета, как показано на рисунке 19.1.

Рис. 19.1. Схема динамического изменения приоритетов в *Windows*

Базовый приоритет процесса определяет, насколько сильно могут различаться приоритеты потоков процесса и как они соотносятся с приоритетами потоков других процессов. Поток наследует этот базовый приоритет и может изменять его так, чтобы он стал немного больше или немного меньше. В результате получается приоритет планирования, с которым потоки начинает исполняться. В процессе исполнения потока его приоритет может отклоняться от базового).

На рисунке 19.1 показан динамический приоритет потока, нижней границей которого является базовый приоритет потока, а верхняя — зависит от вида работ, исполняемых потоком. Например, если поток обрабатывает «пользовательский ввод», то диспетчер задач *Windows* поднимает его динамический приоритет; если же он выполняет вычисления, то диспетчер постепенно снижает его приоритет до базового. Снижая приоритет одного процесса и поднимая приоритет другого, подсистемы могут управлять относительной приоритетностью потоков внутри процесса.

Диспетчер использует систему приоритетов для эффективного управления ресурсами вычислительной системы. Если становится готовым к выполнению другой поток или процесс с более высоким приоритетом, операционная система прекращает исполнение или вытесняет текущий поток или процесс,

Существует группа очередей: по одной для каждого приоритета. *Windows* поддерживает 32 уровня приоритетов; потоки делятся на два класса:

— реального времени и
— переменного приоритета.

Потоки реального времени, имеющие приоритеты от 16 до 31 — это высокоприоритетные потоки, используемыми программами с

[31] ***Thread*** – это *виртуальный процессор*, имеющий свой собственный набор регистров, аналогичных регистрам настоящего центрального процессора. Один из наиважнейших регистров у виртуального процессора, как и у реального — это индивидуальный указатель на текущую инструкцию.

критическим временем выполнения, то есть требующие немедленного внимания системы (по терминологии *Microsoft*).

Диспетчер задач просматривает очереди, начиная с самой приоритетной. При этом, если очередь пустая, то есть нет готовых к выполнению задач с таким приоритетом, осуществляется переход к следующей очереди. Если есть задачи, требующие процессор немедленно, они будут обслужены в первую очередь. Для собственно *системных модулей*, функционирующих в статусе задач, зарезервирована очередь с номером 0.

Перехват прерываний и резидентные программы

Большая часть всех функциональных возможностей операционной системы заключается в обработке разнообразных аппаратных и особенно программных прерываний. Обращение к многочисленным системным функциям операционной системы (ОС) выполняется с помощью вызова специального программного прерывания.

Операционная система предоставляет программе пользователя возможность перехватить любое прерывание, то есть установить свой обработчик этого прерывания. Фактически для этого достаточно записать адрес нового обработчика по соответствующему адресу памяти: имеются системные функции для запоминания адреса прежнего обработчика и установки нового.

Перехват аппаратных прерываний позволяет программе оперативно реагировать на различные события. Особенно часто перехватываются прерывания от таймера, что позволяет выполнять некоторое действие регулярно, через заданный интервал времени, (скажем, отображать текущее время) или же выполнить его один раз в заранее запланированный момент, а также прерывания от клавиатуры, позволяющие выполнить действие при нажатии определенной комбинации клавиш. Например, одно время были популярны резидентные калькуляторы, которые появлялись на экране при нажатии заданных клавиш. Еще один пример такого рода — программы, переключающие русский/латинский регистры клавиатуры.

Перехват программных прерываний позволяет программе модифицировать выполнение любой функции операционной системы. Примерами перехвата прерываний могут служить прерывания для определения реакции программы на нажатие клавиш или на критические ошибки, либо вызов системной программы, которая обеспечивает корректное разделение файлов между процессами. Эта системная программа перехватывает основные файловые функции операционной системы, чтобы отследить все открытия и закрытия файлов и установку/снятие блокировок. На основании этой информации модифицированные функции открытия, чтения и записи файла определяют, разрешена ли запрошенная операция.

Программы, использующие перехват прерываний, можно разбить на два класса.

1. ***Нерезидентные программы***, которые после завершения своей работы возвращают управление и всю занимаемую память системе. Такие программы перехватывают прерывания только на время своей работы и должны обязательно восстановить стандартную обработку прерываний при своем завершении. Требование восстановления стандартной обработки прерываний касается не только нормального завершения, но и завершения по нажатию клавиш и по критической ошибке. В противном случае при последующем возникновении прерывания управление будет передано по адресу уже не существующего в памяти обработчика, а это приведет к серьёзной ошибке операционной системы.
2. ***Резидентные программы*** представляют собой обработчики прерываний, которые постоянно находятся в памяти даже после завершения загрузившего их процесса, вплоть до перезагрузки системы. Таким образом, резидентные программы могут оказывать влияние на работу операционной системы и всех запускаемых программ.

19.2. Изучение принципов синхронного и асинхронного выполнения параллельных подпроцессов.

Операционная система многопроцессорной вычислительной системы должна, прежде всего, выполнять функции обычной операционной системы:

- обрабатывать вызовы; управлять памятью;
- поддерживать файловую систему;
- управлять устройствами ввода-вывода.

Кроме того, многопроцессорная операционная система должны выполнять ряд специфических функций. Выделим из них четыре следующие функции:

- функция *синхронизации* параллельных процессов;
- функция *коммуникации* параллельных процессов;
- функция *управления распределенной памятью* (в системах с распределенной памятью);
- функция *планирования* параллельных процессов.

Отметим, что три первые функции находят непосредственное отражение в языках программирования высокого уровня. Четвертая функция в значительной мере определяет производительность многопроцессорной системы.

Кроме очевидных требований надежности и производительности к операционной системы (ОС) многопроцессорной ЭВМ предъявляются следующие требования.

Прозрачность операционной системы: пользователь не должен знать, где расположены те или иные ресурсы; пользователи должны разделять ресурсы автоматически (средствами ОС).

Масштабируемость операционной системы: выход из строя одного из процессоров системы или увеличение количества процессоров в ней не должны приводить к отказу операционной системы, то есть операционная система должна работать *не хуже*.

Для обеспечения масштабируемости системы

- ни один из процессоров не должен иметь полной информации о состоянии системы,
- процессоры должны принимать решения на основе только локальной информации,
- не должны использоваться глобальные часы.

Центральным понятием операционной системы для многопроцессорных вычислительных систем является понятие процесса. В параграфе 18 даны были определения процесса и потока. Рассмотрим их подробнее.

Процессы.

Единицы работы, между которыми операционная система разделяет процессоры и другие ресурсы вычислительной системы, называется *процессом*. Любая работа вычислительной системы состоит в выполнении некоторой программы. Поэтому можно сказать, что процесс — это выполнение вычислительной системой некоторой системной или прикладной программы, а также их фрагмента.

Каждому процессу в операционной системе соответствует *контекст* процесса. Этот контекст включает в себя:

- *пользовательский контекст* (соответствующий программный код, данные, размер виртуальной памяти, дескрипторы открытых файлов и прочее);
- *аппаратный контекст* (содержимое регистра счетчика команд, регистра состояния процессора, регистр указателя стека, а также содержимое регистров общего назначения);
- *системный контекст* (состояние процесса, идентификатор соответствующего пользователя, идентификатор процесса и прочее).

Важно, что из-за большого объема данных *контекста процесса*, переключение процессора системы с выполнения одного процесса на выполнение другого процесса (*смена контекста процесса*) является относительно дорогостоящей операцией.

Для уменьшения времени смены контекста процесса в современных операционных системах (например, в ОС ***UNIX***) наряду с понятием процесса широко используется понятие легковесного процесса «***light-weight process***» или понятие потока, нити «***thread***». Легковесный процесс (или нить) можно определить, как подпроцесс некоторого процесса, выполняемый в контексте этого процесса (рис. 19.2). Контекст процесса содержит общую для всех его легковесных процессов информацию: виртуальная память, дескрипторы открытых файлов и так

далее. Остальная информация из контекста процесса переходит в контексты его легковесных процессов.

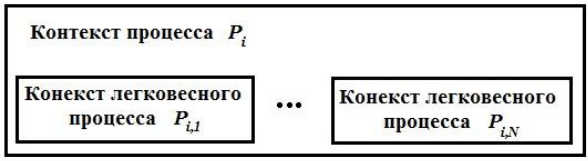


Рис. 19.2. К определению легковесного процесса

Как уже упоминалось в параграфе 18, простейшим процессом является процесс, состоящий из одного легковесного процесса (потока, нити).

Важно отметить принципиальное обстоятельство, а именно, что нити одного процесса выполняются в *общей виртуальной памяти*, то есть имеют *равные права доступа* к любым частям виртуальной памяти процесса. Основной ресурс вычислительной системы — *процессорное время* — выделяется операционной системой не процессу, а легковесному процессу.

Теперь, учитывая выше сказанное, можно определить процесс, как некоторый контекст, включающий виртуальную память и другие системные ресурсы, в котором выполняется, по крайней мере, один легковесный процесс, обладающий своим собственным (более простым) контекстом.

Операционная система «знает» о существовании двух указанных уровней контекстов и способна сравнительно быстро изменять контекст легковесного процесса, не изменяя общего контекста процесса. Заметим, что для синхронизации легковесных процессов, работающих в общем контексте процесса, можно использовать более дешевые средства, чем для синхронизации процессов.

Понятие легковесного процесса направлено на организацию вычислений в многопроцессорной вычислительной системе в случае, когда приложение, выполняемое в рамках одного процесса, обладает внутренним параллелизмом.

Разумеется, параллельное выполнение приложения можно организовать и на пользовательском уровне: путем создания для одного приложения нескольких процессов для каждой из параллельных работ. Однако, при этом не учитывается тот факт, что эти процессы решают общую задачу, а, значит, могут иметь много общего:

— общие данные,
— программные коды,
— права доступа к ресурсам системы и прочее.

Кроме того, как отмечалось выше, каждый процесс требует значительных системных ресурсов, которые при такой организации параллельных вычислений неоправданно дублируются.

Средства создания и завершения процессов.

Рассмотрим основные средства создания и завершения процессов на примере операционной системы ***UNIX*.**

Для создания нового процесса используется системный вызов ***fork***. Все процессы операционной системы ***UNIX***, кроме начального, запускаемого при «раскрутке» системы, образуются при помощи системного вызова ***fork***. После создания процесса-потомка процесс-предок и процесс-потомок начинают «жить» своей собственной жизнью, произвольным образом изменяя свой контекст. Например, и процесс-предок, и процесс-потомок могут выполнить системный вызов ***exec*** (см. ниже), приводящий к полному изменению контекста процесса.

Системный вызов ***wait*** используется для синхронизации процесса-предка и процессов-потомков. Выполнение этого системного вызова приводит к приостановке выполнения процесса-предка до тех пор, пока не завершится выполнение какого-либо из процессов, являющегося его потомком.

Сигнал — это способ информирования процесса со стороны ядра операционной системы о происшествии некоторого события (***event***) в системе, например:

- *исключительная ситуация* (выход за допустимые границы виртуальной памяти, попытка записи в область виртуальной памяти, которая доступна только для чтения и т.д.);
- *ошибка в системном вызове* (несуществующий системный вызов, ошибки в параметрах системного вызова и т.д.);
- *прием сообщения* от другого процесса;
- *нажатия пользователем определенных клавиш* на клавиатуре терминала, связанного с процессом.

Все возможные в системе сигналы имеют уникальные номера и идентификаторы.

С помощью системного вызова ***signal*** пользовательская программа может осуществить «перехват» указанного в вызове сигнала — вызвать соответствующую функцию, которая выполнит обработку этого сигнала. Например, вызов ***signal(SIGFPE, error)*** вызовет выполнение функции ***error*** при переполнении или делении на ноль во время выполнения операции с плавающей запятой.

ОС ***Unix*** предоставляет возможность пользовательским процессам направлять сигналы другим процессам. Например, системный вызов ***kill(PID, signum)*** посылает процессу с идентификатором ***PID*** сигнал с номером ***signum.***

При выполнении системного вызова ***exec(filename,…),*** где ***filename*** – имя выполняемого файла, операционная система производит реорганизацию виртуальной памяти вызывающего процесса, уничтожая в ней сегменты старого программного кода и образуя новые сегменты, в которые загружаются программный код из файла ***filename***.

19.3. Изучение способов передачи данных между параллельными подпроцессами в системах с общей памятью и в распределенных системах.

Процессы, выполняемые на разных процессорах мультикомпьютера, обычно общаются между собой с использованием модели обмена данными посредством *передачи сообщений*. Имеется два варианта организации связи между процессами посредством передачи сообщений:

- *простое рандеву:* операционная система мультикомпьютера предоставляет пользовательским программам средства отправки и получения сообщений в *явном виде* (в виде системных вызовов типа ***send*** и ***receive***);
- *вызов удаленной процедуры:* операционная система *скрывает* от пользователя передачу и прием сообщений под видом механизма вызова удаленной процедуры.

Системные вызовы типа ***send*** *и* ***receive.***

Различают *блокирующие* вызовы (*синхронные* вызовы) и *неблокирующие* вызовы (*асинхронные* вызовы).

Обмен сообщениями между процессами с использование *блокирующих* вызовов происходит по следующей схеме:

- вызов ***send*** посылает сообщение указанному в вызове процессу и блокирует вызывающий процесс до завершения отправки сообщения (рис.19.3.а);
- вызов ***receive*** вызывает блокировку вызывающего процесса вплоть до завершения получения сообщения.

Если вызов ***send*** является *неблокирующим*, то он возвращает управление вызывающему процессу до завершения отправки сообщения (рис.19.3.б).

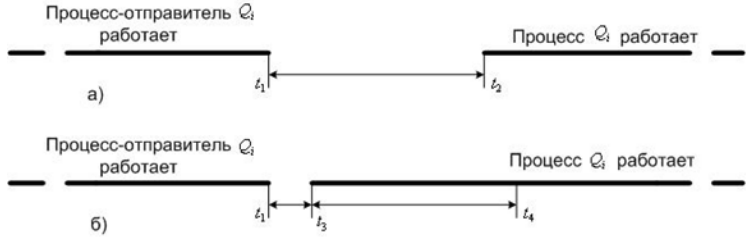


Рис. 19.3.а) Схема блокирующего вызова ***send***. В течение периода времени $[t_1, t_2]$ процесс Q_1 заблокирован, происходит отправка сообщения.

б) Схема неблокирующего вызова ***send***. В течение периода времени $[t_1, t_3]$ процесс Q_1 заблокирован, ОС обрабатывает прерывание, вызванное вызовом. В течение периода времени $[t_3, t_4]$ происходит отправка сообщения.

Аналогично, неблокирующий вызов ***receive*** возвращает управление вызывающему процессу до завершения получения сообщения. Преимущество неблокирующих вызовов состоит в том, что процессы, осуществляющие эти вызовы, могут работать параллельно с обменом сообщениями.

Реализации неблокирующих вызовов ***send*** значительно сложнее реализации блокирующих вызовов. Дело в том, что в изложенной схеме процесс-отправитель сообщения не знает, когда отправка сообщения завершена и когда можно повторно использовать область памяти, в котором хранилось отправляемое сообщение.

Возможно несколько решений этой проблемы.

1. *Копирование* сообщения в буфер ядра ОС. Недостатком данного решения является необходимость наличия дополнительных буферов для вызовов, а также необходимость лишнего копирования, что может существенно снизить производительность системы.
2. *Прерывание* процесса-отправителя в тот момент, когда отправка сообщения завершена. Это решение значительно усложняет программу пользователя, поскольку требует обработки в ней указанных прерываний
3. *Копирование* сообщения в буфер ядра ОС только в том случае, когда процесс пытается повторно использовать область памяти, в которой хранится сообщение. Данное решение может потребовать неоднократного копирования указанной области памяти (если процесс формирует очередные вызовы ***send*** до завершения отправки предыдущих сообщений).

Аналогично, реализации неблокирующих вызовов ***receive*** вызывает серьезные проблемы. Процесс-получатель может информироваться о завершении приема сообщения с помощью прерывания. Однако, опять же, эти прерывания программе пользователя необходимо обрабатывать.

Другим решением является создание для каждого процесса своего «почтового ящика». ОС записывает поступающие сообщения в этот почтовый ящик, а программа пользователя периодически опрашивает его на предмет наличия непрочитанных сообщений. Используются и другие решения.

Функции, которые реализуют процесс передачи сообщений с буферами:

- ***msgget()*** — создает новую очередь сообщений, элементы которой представляют собой буферы сообщений;
- ***msgsnd()*** — используется для посылки сообщения (точнее, для постановки его в указанную очередь сообщений);
- ***msgrcv()*** — используется для приема сообщения (вернее, для выборки сообщения из очереди сообщений);
- ***msgclt()*** — служит для управления очередью сообщений (изменения прав доступа к очереди, уничтожения указанной очереди и прочее.

Замечание. Для процессов, которые взаимодействуют в компьютерной сети в соответствии с семейством протоколов ***TCP/IP***, операционная система ***Unix*** использует программные гнезда (***sockets***). Взаимодействие процессов на основе программных гнезд основано на модели "клиент-

сервер". Операционная система поддерживает внутренние соединения и маршрутизацию данных от клиента к серверу.

Вызов удаленной процедуры.

Часто обмен данными между процессами носит ярко выраженный *асимметричный* характер: один из процессов (процесс-клиент) запрашивает у другого процесса (процесс-сервер) некоторую услугу (сервис) и не продолжает свое выполнение до тех пор, пока эта услуга не будет выполнена и пока процесс-клиент не получит соответствующие результаты. Семантически такой режим взаимодействия процессов эквивалентен вызову процедуры. Поэтому естественно желание оформить его должным образом и синтаксически.

С точки зрения прикладной программы вызов удаленной процедуры (***remote procedure call***) происходит по следующей схеме:

- процесс Q_i, выполняемый на процессоре P_i, вызывает процедуру, выполняемую на процессоре P_j;
- процесс Q_i приостанавливается;
- вызванная процедура выполняется на процессоре P_j;
- процесс Q_i, продолжает выполняться.

Информация между процессом Q_i и вызываемой процедурой передается через параметры процедуры (как в традиционных языках программирования).

Реализация вызова удаленной процедуры:

- передача процессом-клиентом Q_i параметров процедуры процессу-серверу Q_j (рис. 19.4);
- выполнение процесса-сервера Q_j;
- передача процессом-сервером Q_j результатов процессу-клиенту Q_i (рис. 19.5).

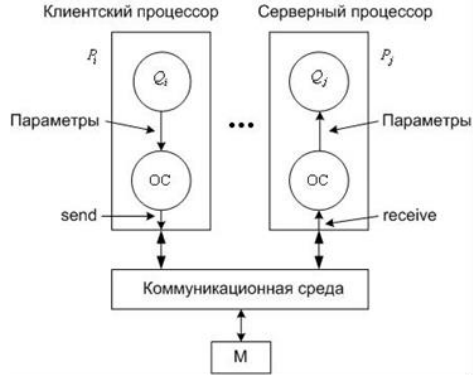


Рис. 19.4. Передача процессом-клиентом Q_i параметров процедуры процессу-серверу Q_j.

Операционная система клиентского процессора упаковывает параметры в сообщение и с помощью системного вызова ***send*** отправляет сообщение серверному процессору. ОС серверного процессора с помощью системного вызова ***receive*** принимает параметры, распаковывает их и передает процессу-серверу.

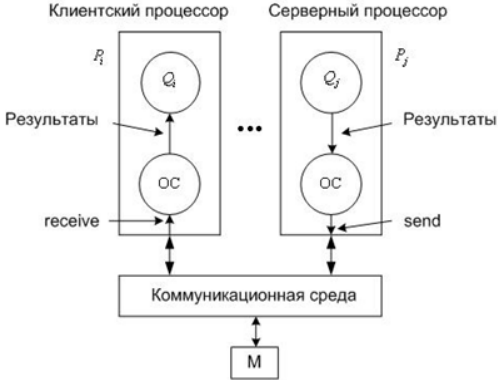


Рис. 19.5. Передача процессом-сервером Q_j результатов выполнения процедуры процессу-клиенту Q_i.

Аналогично, при передаче процессом-сервером $\boldsymbol{Q_j}$ результатов выполнения процедуры процессу-клиенту $\boldsymbol{Q_j}$ (рис. 19.5). ОС серверного процессора упаковывает результаты в сообщение и с помощью системного вызова ***send*** отправляет сообщение клиентскому процессору. ОС клиентского процессора с помощью системного вызова ***receive*** принимает результаты, распаковывает их и передает процессу-клиенту.

Поскольку в различных компьютерах мультикомпьютера данных могут представляться по-разному (например, часто по-разному представляются числа с плавающей точкой), то одной из основных функций данного механизма обмена сообщениями является автоматическое преобразование форматов данных при взаимодействии процессов, выполняющихся на разнородных компьютерах.

Хотя идея изложенной схемы проста, реализация этой схемы сталкивается с рядом проблем.

Во-первых, поскольку процесс-клиент и процесс-сервер находятся в разных адресных пространствах, невозможно использование указателей в качестве параметров. В некоторых случаях эту проблему можно обойти, передавая процессу-серверу не указатель, а сами данные, на которые ссылается этот указатель. Если данные имеют сложную структуру, этот метод практически неработоспособен. Аналогичная ситуация имеет место также и при использовании системных вызовов типа ***send*** и ***receive***.

Во-вторых, если язык программирования пользовательского процесса не является сильно типизированным, то не всегда операционная система процесса-клиента может передать процессу-серверу все необходимые данные. Такая ситуация, например, может иметь место, если количество элементов массива определяется не его описанием, а ограничивается каким-либо специальным символом.

В-третьих, локальная процедура наряду с параметрами, передаваемыми через ее заголовок, может использовать глобальные параметры. Удаленная процедура из-за разных адресных пространств процесса-клиента и процесса-сервера не может использовать глобальные переменные процесса-клиента.

Вызов удаленной процедуры может быть реализован с помощью системных вызовов или библиотечных функций.

Вопросы для контроля

1. Какие классы процессов по отношению к приоритету Вы можете определить? Опишите их особенности.

2. Дайте определение приоритета, прерывания, резидентной программы.

3. Чем отличаются контексты процесса и потока?

4. Опишите особенности передачи данных в параллельных системах с общей памятью.

5. Опишите особенности передачи данных в распределенных параллельных системах.

Глава 20. Принципы параллельного программирования

Учебные цели:

- *дать представление о методах преобразования последовательных программ в параллельные;*
- *дать представление о методах построения параллельного алгоритма;*
- *дать представление об особенностях выполнения параллельных вычислений: мобильность, переносимость, интероперабельность.*

20.1. Методы преобразования последовательных программ в параллельные.

Задача распараллеливания.

В любом полном рассмотрении вопросов параллельного программирования должны быть представлены способы получения параллельной программы из обычной алгоритмической записи, которая несет в себе след чисто последовательной логики рассуждений, так как любые вычисления, проводимые вручную и автоматически, выполнялись последовательно. Однако оказывается, что любой реальный процесс при детальном его рассмотрении выполняется параллельно. И чем внимательнее и детальнее мы постараемся его описать, тем больше это описание будет приближаться к параллельной форме. Следовательно, задача состоит в том, чтобы с помощью каких-то понятий научиться достаточно просто и полно описывать естественные параллельные процессы.

Сложность задачи распараллеливания во многом определяется языком, на котором записан исходный алгоритм. Наиболее просто это делать, если язык не содержит циклических структур. Примером таких языков можно назвать языки «геометрического описания деталей со сложной объемной конфигурацией». Это язык, в котором предложение — последовательность директив.

Как правило, основное время выполнения программ на обычных языках связано с реализацией циклов. Поэтому важной задачей является распараллеливание циклов.

Некоторые языки программирования имеют средства для описания параллелизма. Однако все параллельные фрагменты программы должны быть указаны явным образом. Более того, должна быть указана последовательность их выполнения, организована передача необходимых управлений и так далее.

При использовании любого языка программирования программист должен:

- выделить и пометить независимые фрагменты программы;
- обеспечить синхронизацию при выполнении фрагментов;
- осуществить взаимное управление фрагментами.

Следовательно, все трудности по организации параллельной обработки ложатся на программиста, так как он, по существу, должен предварительно написать весь "сценарий" прохождения задачи через компьютер в параллельном режиме.

На разработку математического и программного обеспечения, согласно различным литературным источникам, тратится от 70 до 90 % всех затрат на создание программного проекта.

Это одна из основных причин, почему необходимо эффективно использовать накопившееся обеспечение для однопроцессорных ЭВМ при переходе к многопроцессорным вычислительным системам. Математическое и программное обеспечение следует адаптировать для новых вычислительных систем.

Таким образом, рождается актуальная проблема — *преобразование последовательных программ в последовательно-параллельные*. Но так как эта работа необычайно трудоемка, ее следует автоматизировать. Решение указанной проблемы позволяет,

— во-первых, высвободить большие трудовые ресурсы,
— во-вторых, использовать накопленный за долгие годы развития ЭВМ информационно-программный багаж без ручного перепрограммирования и,
— в-третьих, осуществить переход на многопроцессорные системы с меньшими затратами труда и времени.

Распараллеливание программ можно осуществлять как на уровне отдельных задач, так и на уровне отдельных процедур, операторов, операций и микроопераций. Целесообразность преобразования на указанных уровнях должна решаться в каждом отдельном случае в зависимости от

— структуры вычислительной системы,
— типа программы и
— цели, которая ставится при ее решении.

Рассмотрим общую схему преобразования последовательных программ в последовательно-параллельные. Организовать параллельные вычисления можно с помощью некоторой формальной процедуры, выполняемой автоматически над каждой программой, состоящей из последовательности операторов обычного языка программирования.

Эта процедура позволяет избавить программиста от анализа собственной программы и помогает выявить ее внутренний параллелизм. Выявление параллелизма заключается в разбиении всех операторов программ на два класса:

— тех, которые могут выполняться параллельно, и
— тех, которые должны быть завершены прежде, чем следующие последовательности операторов начнут выполняться.

Такой подход к распараллеливанию основывается на представлении программы в виде ориентированного графа G, множество вершин которого $V = \{v_1, v_2, ..., v_n\}$ соответствует либо отдельным операторам программы, либо совокупности этих операторов языка высокого уровня. Множество направленных дуг $U = \{u_1, u_2, ..., u_m\}$ соответствует возможным переходам между операторами программы.

Из теории графов известно, что ориентированный граф полностью описывается соответствующей матрицей смежности.

Подграфу, представляющему собой последовательность различных вершин, каждая из которых имеет единственную входную и единственную выходную вершины, будем ставить в соответствие линейный участок в программе.

Итак, ставится задача — преобразовать алгоритм решения задачи, заданный последовательной программой, для параллельной его реализации на многопроцессорной вычислительной системе.

Для преобразования программы из одной (*последовательной*) формы в другую (*параллельную*) традиционно используем ее интерпретацию в виде орграфа. Вообще говоря, идея использовать некоторый «формализм» в качестве посредника для преобразования программ встречалась и раньше в связи с другими задачами, это позволяет обнаруживать и анализировать информационные связи в целях глобальной экономии памяти.

Построим схему алгоритма распараллеливания программ, используя некоторые сведения из теории графов.

Схема алгоритма.

1. Представление исходной программы в виде графа.
2. Сведение циклического графа к ациклическому.
3. Наложение информационных связей, заданных между операторами программы, на связи возможных переходов.
4. Распределение вершин графа по уравнениям готовности.
5. Формирование ветвей решения.

В результате указанных действий каждый уровень разбивается на несколько подуровней и производится их согласование. Операторы исходной программы, окончательно распределенные по уровням готовности, объединяются в ветви решения, что и является окончательным результатом рассматриваемой процедуры распараллеливания.

20.2. Построение параллельного алгоритма.

Алгоритм распараллеливания ациклических участков программы состоит из четырех этапов:

- —построение графа зависимостей по данным между операторами программы;
- —построение ярусно-параллельной формы (ЯПФ) программы;
- —составление по ЯПФ параллельной программы;

—отображение полученной программы на архитектуру используемой параллельной вычислительной системы.

Построение графа зависимостей по данным.

Граф зависимостей по данным ***G*** между операторами программы (*Data Dependence Graph, DDG*) строится на основе *информационной, логической и конкуренционной* зависимости между операторами программы. Заметим, что о зависимости по данным можно говорить на разных уровнях — от отдельных инструкций, до более крупных программных блоков.

Информационная зависимость между операторами программы. Оператор ***B*** зависит от оператора ***A*** информационно, если оператор ***A*** «вырабатывает» некоторую переменную ***x,*** которую использует оператор ***B***. Другими словами, оператор ***B*** зависит от оператора ***A*** *информационно*, если

1) существует путь от входного оператора программы, проходящий через операторы ***A, B*** и
2) оператор ***A*** является последним перед ***B*** оператором этого пути, "вырабатывающий" значение переменной ***x***, которое используется оператором ***B***.

Формально информационную зависимость оператора ***B*** от оператора ***A*** можно записать в следующем виде:

$$In(B) \cap Out(A) \neq \emptyset$$

где ***In (B)*** — совокупность входных переменных оператора ***B***,

Out (A) — совокупность выходных переменных оператора ***A***,

Ø – пустое множество.

Логическая зависимость между операторами программы. Оператор ***B*** зависит от оператора ***A*** *логически*, если

1) существует путь от входного оператора программы до оператора A и
2) оператор ***A*** является "распознавателем", решающим будет или нет выполняться оператор ***B***.

Конкуренционная зависимость между операторами программы. Операторы ***A, B*** зависят друг от друга *конкуренционно*, если

1) существуют пути от входного оператора программы, как до оператора ***A***, так и до оператора ***B***, и
2) операторы ***A*** и ***B*** "вырабатывают" одну и ту же переменную ***x***.

Формально конкуренционную зависимость операторов ***A, B*** можно записать в следующем виде:

$$Out(A) \cap Out(B) \neq \emptyset$$

Пример 20.1.Построение ориентированного графа фрагмента программы.

Пользователь вводит относительные адреса ***x, y*** начала и конца массива. Известно, что переменная, имеющее наибольшее значение, соответствует концу массива, а переменная, имеющая наименьшее значение, – началу

массива (массив отсортирован). Требуется распечатать массив, а также нижнюю и верхнюю его границы.

Рассмотрим также следующий фрагмент программы, которая решает поставленную задачу (слева даны обозначения операторов программы; ***c*** — адрес, база массива):

A ввод (x, y);
B l := x;
C h := y;
D v := c+x;
E z := c+y;
P если x>y то F иначе G;
F h := x; l := y;
G печать от min (z, v) до max (z, v);
H печать (l, h);

Граф зависимостей этой программы изображен на рисункне 20.1.

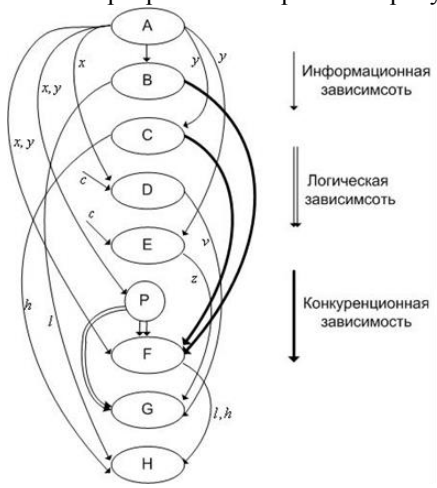


Рис. 20.1. Граф зависимостей программы.

Конкуренционная зависимость в программе имеет место, поскольку ***Out (B) ∩ Out (F) ≠ Ø, Out (C) ∩ Out (F) ≠ Ø***

Большой размер графа ***G*** усложняет последующие этапы распараллеливания программы. Для уменьшения размера графа ***G*** целесообразно произвести сжатие линейных участков программы в обобщенные операторы, то есть выделить в исходном графе ***G*** линейные подграфы и поставить в соответствие каждому из них одну обобщенную вершину графа. Сохраним за преобразованным графом обозначение ***G***. Таким образом, вершинами графа ***G*** являются операторы исходной программы, а также фрагменты исходной программы в виде линейных участков.

20.3. Понятие ярусно параллельной формы (ЯПФ), построения ЯПФ и минимальной ЯПФ алгоритма.

Построение ярусно-параллельной формы программы.

Алгоритм построения ярусно-параллельной формы программы состоит в следующем рекурсивном построении:

1. На первый ярус ярусно-параллельной формы (ЯПФ) заносятся все операторы, в которые не идут стрелки графа зависимостей ***G***;
2. Если построено ***k*** ярусов, то в ***(k+1)***-й ярус заносятся все те операторы, которые имеют входящие стрелки только от первых ***k*** ярусов.

Пример 20.2. Построение графа зависимостей для фрагмента программы (пример 20.1)

Построим ярусно-параллельную форму (ЯПФ) для программы, граф зависимостей которой приведен на рисунке 20.1.

Входящие стрелки отсутствуют только у оператора ***A***. Поэтому на первый ярус ярусно-параллельной формы (ЯПФ) помещаем только этот оператор (рис. 20.2).

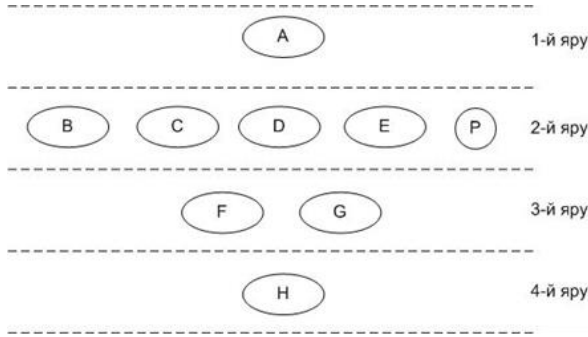


Рис. 20.2. Ярусно-параллельная форма для программы, граф зависимостей которой приведен на рисунке 20.1.

Входящие стрелки только от операторов 1-го яруса имеют операторы ***B, C***, ***D***, ***E***, ***P***. Поэтому помещаем их на второй ярус.

Входящие стрелки только от операторов 1-го и 2-го ярусов имеют операторы ***F, G***. Помещаем их на третий ярус.

Наконец, входящие стрелки только от операторов 1-го, 2-го и 3-го ярусов имеет только оператор ***H***. Помещаем этот оператор на четвертый ярус (рис. 20.2).

Из алгоритма построения ЯПФ следует, что все операторы, находящиеся на одном уровне этой формы, могут выполняться параллельно. Ярус ЯПФ иногда называется уровнем готовности операторов.

Ярусы ярусно-параллельной формы (ЯПФ) устанавливают между операторами отношение предшествования: к моменту начала вычислений на ***(k+1)***-м ярусе должны быть закончены вычисления на ***k***-м ярусе.

С ярусно-параллельной формой (ЯПФ) связан ряд важных понятий (о высоте и ширине ЯПФ говорилось еще в параграфе 18).

Высотой ЯПФ называется количество ее ярусов.

Шириной яруса ЯПФ называется количество операторов на этом ярусе. Шириной ЯПФ называется максимальная из ширин ярусов данной ЯПФ. Ярусно-параллельная форма данного алгоритма, имеющая минимальную высоту, называется *минимальной* ЯПФ или максимально-параллельной ЯПФ. Ширина минимальной ЯПФ называется *шириной алгоритма*, а ее высота минимальной ЯПФ — *высотой алгоритма*. Ширина алгоритма и высота алгоритма представляют собой примеры *мер параллелизма* алгоритма.

Составление по ЯПФ параллельной программы.

Рассмотрим использование ярусно-параллельной формы для разбиения последовательной программы на легковесные процессы (нити) с целью выполнения программы на мультикомпьютере. В этом случае накладные расходы на коммуникацию минимальны, поскольку все нити разделяют *одно адресное пространство*, а синхронизационные операции для нитей (потоков) выполняются *проще и быстрее*, чем для обычных процессов.

Очевидно, что разбиение программы на нити многовариантно. В качестве критерия оптимальности разбиения естественно использовать время выполнения программы. Для определения этого времени в качестве веса узлов графа зависимостей по данным используем время выполнения этих узлов в последовательной программе.

Время выполнения узла может быть найдено с помощью профилирования программы. Для получения более точных результатов можно воспользоваться аппаратными таймерами, имеющимися на большинстве современных мультипроцессоров.

Поскольку целью разбиения является получение выигрыша по времени, количество нитей, на которые производится разбиение, обычно принимается равным количеству процессоров в системе. По этой же причине при разбиении следует стремиться к тому, чтобы *времена выполнения всех нитей были достаточно близки друг к другу* (задача балансировки загрузки).

Разбиения выполняются последовательно по ярусам ЯПФ: к моменту начала разбиения ***(k+1)***-го яруса должно быть закончено разбиение ***k***-го яруса. При разбиении на ***k***-ом ярусе для каждого узла производится *локально оптимальный* выбор нити, в которую он должен быть включен: делается попытка включить его в каждой из имеющихся нитей, после чего выбирается лучший вариант. Очевидно, что при этом каждый раз производится пересчет времени выполнения каждой из нитей. Оптимизация должна производиться

с учетом потерь времени на синхронизацию нитей, если они обмениваются между собой данными. Организация синхронизации нитей также является задачей рассматриваемого этапа распараллеливания.

Эффективность мультипроцессорной программы в значительной мере зависит от эффективности использования ею кэш-памяти, поскольку близко расположенные в памяти данные преимущественно размещаются в более быстрой кэш-памяти. При разбиении программы на нити, это обстоятельство также следует учитывать (путем эмуляции кэш-памяти процессора, на котором будет выполняться данная нить, и соответствующей корректировки времени выполнения узла).

Отображение полученной программы на архитектуру используемой параллельной вычислительной системы.

Положим, что исходная последовательная программа разбита на нити так, как описано выше. Поскольку количество нитей равно количеству процессоров в мультипроцессоре, данный этап не представляет сложности.

Для определения этого времени в качестве веса узлов графа зависимостей по данным используем время выполнения этих узлов в последовательной программе.

Время выполнения узла может быть найдено с помощью профилирования программы. Для получения более точных результатов можно воспользоваться аппаратными таймерами, имеющимися на большинстве современных мультипроцессоров.

20.4. Особенности выполнения параллельных вычислений: мобильность, переносимость, интероперабельность.

Мобильность

Под *мобильностью* (***portability***) понимается возможность переноса программы (в том числе и ОС) на другую аппаратную платформу, то есть на другой тип процессора и другую архитектуру компьютера. Здесь имеется в виду перенос с умеренными трудозатратами, не требующий полной переработки системы.

Свойство мобильности не столь однозначно положительно, как может показаться. Чтобы программа была мобильна, при ее разработке следует отказаться от глубокого использования особенностей конкретной архитектуры (таких, как количество и функциональные возможности регистров процессора, нестандартные команды и т.п.). Мобильная программа должна быть написана на языке достаточно высокого уровня (часто используется язык Си), который можно реализовать на компьютерах любой архитектуры. Платой за мобильность всегда является некоторая потеря эффективности, поэтому немобильные системы распространены достаточно широко.

Переносимость.

С другой стороны, история системного программирования усеяна останками замечательных, эффективных и удобных, но немобильных операционных систем, которые вымерли вместе с процессорами, для которых они предназначались. В то же время мобильная система ***UNIX*** продолжает процветать четвертый десяток лет, намного пережив те компьютеры, для которых она первоначально создавалась. Примерно 5-10% исходных текстов ***UNIX*** написаны на языке ассемблера и должны переписываться заново при переносе на новую архитектуру. Остальная часть системы написана на Си и практически не требует изменений при переносе.

Некоторым компромиссом являются *многоплатформенные* ОС (например, *WindowsNT*), изначально спроектированные для использования на нескольких аппаратных платформах, но не гарантирующие возможность переноса на новые, не предусмотренные заранее архитектуры.

Написать корректную и эффективную программу трудно. Поэтому если программа уже работает в одной среде, то, скорее всего, никому не захочется повторять пройденный путь ее создания при переходе на другой компилятор, процессор или операционную систему. В идеале программа не должна требовать внесения никаких изменений.

Этот идеал называется *переносимостью* (***portability***). На практике термин «*переносимость*» нередко относят к более слабому свойству: программу проще видоизменить при переносе в другую среду исполнения, чем написать заново. Чем меньше изменений надо внести, тем выше переносимость программы.

Зачем нужно свойство переносимости, если программа будет исполняться в какой-то конкретной среде, зачем тратить время на то, чтобы обеспечивать ей более широкую область применения?

Во-первых, каждая хорошая программа почти по определению начинает с какого-то момента применяться в непредсказуемых местах. Создание продукта с функциональностью более общей, чем изначальная спецификация, приведет к тому, что меньше сил придется тратить на поддержку продукта на протяжении его жизненного цикла.

Во-вторых, среда исполнения меняется. При замене компилятора, операционной системы, железа меняются многие параметры окружения программы. Чем меньше программа зависит от специальных возможностей, тем меньше вероятность возникновения сбоев при изменении условий и тем проще программа адаптируется к этим условиям.

И наконец, последнее и самое важное обстоятельство: *переносимая программа всегда лучше написана*. Усилия, затраченные на обеспечение переносимости программы, сказываются на всех ее аспектах; она оказывается лучше спроектированной, аккуратнее написанной и тщательнее протестированной. Технологии программирования переносимых программ непосредственно привязаны к технологиям хорошего программирования вообще.

Естественно, степень переносимости должна определяться, исходя из реальных условий. Не существует абсолютно переносимой программы: просто программа, опробована не во всех средах. Однако, можно сделать переносимость одной из своих главных целей, стараясь создать программу, которая бы выполнялась без изменений практически везде. Даже если эта цель не будет достигнута в полном объеме, время, потраченное на ее достижение, с лихвой окупится, если программу придется переделывать её под другие условия.

Несколько советов разработчику переносимых программ.

1. Старайтесь писать программы, которые бы работали при всех комбинациях различных стандартов, интерфейсов и сред, в принципе подходящих для ее исполнения.
2. Не старайтесь исправить каждую ошибку переносимости, добавляя дополнительный код, наоборот, адаптируйте программу для работы при новых ограничениях.
3. Используйте абстракцию и инкапсуляцию, чтобы ограничить и контролировать те непереносимые фрагменты кода, без которых не обойтись.
4. Если ваш код сможет работать при всех ограничениях, а системные различия в нем будут локализованы, он станет еще и более понятным.

Интероперабельность.

В настоящее время всё большую роль в области информационных технологий играет так называемая «проблема интероперабельности». Под *интероперабельностью* будем понимать способность систем и компонентов к взаимодействию, основанному на использовании информационно-коммуникационных технологий (ИКТ).

Причина того, что в настоящее время проблема интероперабельности приобретает всё большее значение, лежит, в первую очередь, в том, что сегодня практически ни одна сфера жизни не обходится без использования ИКТ, а их развитие и насыщение разнородными средствами вычислительной техники привело к созданию гетерогенной среды, в которой разнородные компоненты должны взаимодействовать друг с другом. причем уровень гетерогенности постоянно увеличивается. Основным инструментом решения проблемы интероперабельности или «прозрачности» гетерогенной среды выступает последовательное применение

— принципов открытых систем и
— методологии функциональной стандартизации.

Свойство *интероперабельности*, наряду со свойством *переносимости*, составляет одно из важнейших свойств открытых систем, и в настоящее время всё большее внимание уделяется именно вопросам обеспечения интероперабельности для информационных систем (ИС) различного

масштаба (от наносистем до ***GRID***[32]-систем и сверхсложных систем) и информационных систем (ИС) различного назначения. Особенно много работ появляется в связи с решением задач формирования и развития информационного общества, включая такие сферы, как наука, образование, здравоохранение, государственное управление, библиотечное и музейное дело. Можно констатировать, что обеспечение интероперабельности является одной из главных основ формирования и развития информационного общества.

Следует отметить, что долгое время решалась проблема так называемой «технической» интероперабельности, связанная с выбором стандартов на протоколы связи. Эту проблему можно считать к настоящему времени в основном решенной.

В настоящее время термин интероперабельность получил расширенное значение. Это связано, главным образом, с тем, что информационно-коммуникационные технологии (ИКТ) в своем развитии стали основой для формирования и развития социо-технических систем.

В таких системах существует необходимость не просто в обмене данными, но и в эффективном использовании информации, где понятие «информация» может получать различные смысловые наполнения в зависимости от рассматриваемой отрасли человеческой деятельности. Укрупнённо, можно говорить о переходе к «семантической» интероперабельности.

Вопросы для контроля

1. Дайте определение интероперабельности, переносимости программы.

2. Какие этапы проходит процесс построения параллельного алгоритма из последовательного?

3. Может ли быть несколько минимальных форм ярусно-параллельных форм алгоритма?

4. Какие типы зависимостей исследуют при построении ярусно-параллельной формы алгоритма?

5. Какие принципы лежат в основе решения проблемы интероперабельности или «прозрачности» гетерогенной среды?

---

[32] ***Грид*** (англ. ***grid*** — решётка, сеть) — это система, которая, во-первых, распределяет ресурсы, не находящиеся под единым центром управления, во-вторых, использует общие протоколы и интерфейсы, в-третьих, обеспечивает нужный уровень обслуживания. **Грид** — это система, которая координирует распределенные ресурсы посредством стандартных, открытых, универсальных протоколов и интерфейсов для обеспечения нетривиального качества обслуживания.

Глава 21. Перспективы развития современного системного программного обеспечения.

Учебные цели:

- *дать представление о перспективах развития системного программного обеспечения;*
- *дать представление об особенностях структуры и функционирования многопроцессорных вычислительных систем.*

21.1. Перспективы развития системного программного обеспечения.

Существующая система

Вероятно, многие программисты сталкивались с ситуацией, когда почти невозможно использовать для построения новой системы уже готовые наработки от предыдущей, поэтому новую систему строят как перенастроенную предыдущую. В этом случае сама разработка реально сводится к программированию в заданных рамках этого единого монолитного фреймворка. Подобный вариант обычно дает не очень полезные результаты, хотя они и рассчитаны на гарантированный успех прикладной разработки.

Общий принцип построения вычислительных систем почти всегда или в подавляющем большинстве случаев может быть сведен к разбиению с двух точек зрения: является ли данный код

- библиотечным или специфичным для приложения и
- интерфейсным, кодом логики обработки или кодом хранения данных.

Разделение разработки программного обеспечения на две самостоятельные задачи позволяет,

- сохраняя код специфики хранения данных, модернизировать логику приложения;
- оставляя неизменной логику приложения, модернизировать и даже полностью заменить интерфейс, создав его аналог с использованием другой технологии.

Первый принцип предполагает одновременное существование нескольких приложений, имеющих самостоятельную логику обработки данных, но использующих одни утилиты их обработки. Второй — использование одновременно несколькими приложениями работу одних и тех же утилит хранения данных.

Отсутствие подобных возможностей сделало бы модернизацию и наращивание программного обеспечения очень трудозатратным процессом. Кроме того, такой блок модулей, как «утилиты интерфейса» обычно строится так, чтобы его можно было применять в любом существующем или разрабатываемом приложении.

В существующей системе модернизации или полного изменения существующей прикладной разработки все достаточно ясно и понятно:

— как ввести в систему еще один интерфейс пользователя,
— к какой группе и в каких соглашениях отнести новый программный код,
— где искать проблемы во время отладки и,
— как, вообще, модернизировать существующую систему.

Однако, направления и темпы развития современного программного обеспечения, бурное развитие новейших приложений наводят на мысли о том, что недалекое будущее программного обеспечения может оказаться совсем не таким, как представляется сейчас.

Распараллеливание выполнения

Выясним, когда параллельность обслуживания запросов или выполнения процессов помогает работе, а когда она может сильно помешать.

Параллельное выполнение процессов на деле означает равноправие любого из них и значительно ухудшает характеристики быстродействия системы. При параллельном обслуживании запросов система ставит все задания на исполнение, и периодически останавливает обслуживание какого-либо процесса, с сохранением его контекста, передает управление другому процессу с восстановлением его контекста исполнения. Цикл повторяется внутри очереди заданий до полного выполнения каждого из них. В итоге каждое задание тратит определенное время на выполнение, но, возможно, еще большее количество времени уходит на ожидание в очереди на исполнение.

При последовательном выполнении задач система ставит запросы в ожидание исполнения. После чего монопольно выполняет первый попавшийся, затем берет следующий — и все повторяется до полного обслуживания всего списка запросов.

Однако, смена однопроцессорных или многопроцессорных систем на системы, имеющие большое количество ядер или процессоров, то есть, физически параллельно выполняющих операции аппаратных модулей, многое меняет. Но приобретение подобной мощной техники отнюдь не гарантирует реального повышения производительности компьютерных систем, так как далеко не все современные программные средства умеют использовать такое количество ядер или процессоров.

Поэтому, в качестве одного из направлений развития софтвера, возможна разработка стратегий выполнения алгоритмов и обработки данных, ориентированных на *физическое распараллеливание*.

Структуры и алгоритмы быстрой памяти

Раньше наиболее ценным, но весьма ограниченным, ресурсом была *оперативная память*. Все алгоритмы и структуры разрабатывавшихся программ были в значительной степени нацелены на минимизацию ее

использования. Сейчас стандартная, коммерчески доступная компьютерная система, имеет в своем распоряжении оперативную память в несколько гигабайт, что превосходит весь объем жестких дисков, применявшихся ранее. Поэтому следующим направлением перспектив представляется *разработка алгоритмов и структур данных, занимающих большие объемы, размещаемые в оперативной памяти*. В недалеком прошлом эти методы эксплуатировались, в частности, в таблицах типа ***SUPERCALK***[33]***, WORKS***[34]***,*** для предварительного вычисления текущих значений и в табличных вычислениях. Представляется неплохим и весьма перспективным *улучшение различных алгоритмов и процедур обработки данных, применяющихся для сокращения объемов вычислений, требующих больших объемов ОЗУ*. Например, обработка изображений с помощью сплайнов, методов триангуляций, факторизация неотрицательных матриц, методов обработки с помощью нейронных сетей, вычислений различного рода таблиц, — все эти методы и те, что будут разработаны, актуальны в области решения задач преобразований, кодирования и декодирования данных: СУБД, видео, звук, графика, задачи поиска и так далее.

Время доступа к жестким дискам, повсеместно применявшимся ранее и применяющимся сейчас в качестве основных накопителей, гораздо больше времени доступа к оперативной памяти. Как уже было сказано, ранее использовавшиеся стратегии и алгоритмы в разработке баз данных были направлены на нивелирование этого фактора. Например, методы блочной организации файлов данных в СУБД, а также кэширования оперативной памяти компьютера. На сегодняшний день мы видим настоящий бум для твердотельных накопителей SSD, которые отличаются тем, что имеют чрезвычайно большую скорость доступа к данным, и соотношение времени доступа оперативной памяти и к дискам существенно изменилось. Логически отсюда следует, что направление ближайших исследований в области организации СУБД будет меняться, и ведущие разработчики баз данных, возможно поменяют сам основополагающий принцип, на котором работают большинство существующих разработок и заменят структуру индексация с B*-дерева, например, на более многообещающий RB-дерево, или даже будут разработаны новые механизмы, на сегодняшний день неизвестные.

---

[33] ***SuperCalc*** — электронный табличный процессор, выпущенный *Sorcim* в 1981 году и изначально поставлявшаяся вместе с *WordStar* в составе набора программ CP/M для портативного компьютера *Osborne 1*. В 1984 году *Sorcim* была куплена *Computer Associates*, из-за чего последние версии *SuperCalc* носили название *CA-SuperCalc*. В отличие от *VisiCalc*, *SuperCalc* является одной из первых электронных таблиц, способных итеративно разрешать кольцевые ссылки (ячейки, значения которых зависят друг от друга).

Версии *SuperCalc* были выпущены для компьютеров *Apple II*, IBM PC с операционной системой *DOS*, а также для *MS Windows*, мейнфреймов IBM (S/360) и VAX/VMS.

[34] ***Microsoft Works*** — простой офисный пакет приложений, предназначенный для домашнего пользования. В состав этого пакета входит программное обеспечение, предоставляющее возможности обработки текста, работы с таблицами, управления базами данных и календарного планирования. В сравнении с *Microsoft Office* обладает меньшими возможностями: к примеру, электронные таблицы содержат лишь один лист, и тому подобное.

Проблема портирования (переноса программного обеспечения с одной операционной системы на другую)

Как выше говорилось, эта проблема достаточно серьезна. На сегодняшний день абсолютно переносимых языков программирования не существует. Принято считать, что наиболее переносимый язык программирования — это язык Си и его клоны. Однако несовместимость компиляторов тем не менее существует. На текущий момент сделано много попыток разработать *портируемый* компилятор. Но факт остается фактом: язык Си стандартизирован, и достаточно давно, но 90% программного обеспечения, компилируемого, например, в ***Linux***, не сможет быть скомпилирован, скажем в ***Windows XP*** без изменения исходного кода. В недалеком будущем, думается, усилия ведущих разработчиков компиляторов будут направлены и на решение этой проблемы. Как следствие, к перспективам рынка программного обеспечения относится возможность *портирования софта*. Тут можно использовать различные подходы — использовать средства виртуализации, интерпретирующие системы, кросскомпиляцию и так далее.

Линейные алгоритмы и структуры

Процессоры компьютерных систем, применявшиеся ранее в качестве ядра, почти не имели внутренних кэшей. К ним программисты относились, можно с уверенностью утверждать, просто никак. Средства встроенных или дополнительных кэшей процессоров могли быть хоть как-то использованы только при выборе аппаратной части компьютера. На все остальное, то есть работу программистов, планирование алгоритмов и структур данных — это совершенно не влияло. На сегодняшний день ситуация кардинально изменилась. Размеры кэшей стали настолько велики, что не учитывать их при разработке СУБД стало попросту невозможно. Существенно возросла сложность структуры кэшэй. И теперь невозможно себе представить СУБД, где бы с максимальной эффективностью не использовались алгоритмы и структуры кэширования данных. *Использование кэша становится правилом хорошего тона и является показателем степени качества работы программиста*.

Удивительно, но уже есть примеры того, как *простая перестановка местами полей в базе данных, неожиданно дает выигрыш в производительности сразу на несколько десятков процентов.*

И если ранее разработчики предпочитали, разрабатывать не линейные, а сложные структуры, что существенно повышало скорость поиска записи, то в настоящее время *линейные структуры оказываются более эффективным, так как вторичное обращение к следующему элементу отрабатывается намного быстрее, чем к элементу, не находящемуся в кэше процессора*. Еще одно направление в линейности представлено исследованиями в области алгоритмов, выполняющихся линейно, или содержащих минимизированное

число переходов. Выполнение подобного алгоритма значительно улучшается именно из-за кэширования последовательности исполняемых кодов, а также и из-за встроенных в процессоры и достаточно давно использующихся средств конвейеризации выполнения. Одним из таких направлений, в частности, и для примера, является *метод развертывания циклов*.

Направления поиска перспектив

Другие направления перспектив могут выглядеть как незначимыми, так и наоборот, гораздо более значимыми в каких-то областях. Каким образом определить, какое направление в ближайшем будущем окажется перспективным? Общим правилом, которым могут руководствоваться программисты, может быть появление новых средств обмена информацией, новых носителей или принципов и изменение ключевых факторов, вызывавших применение определенных методов. Часть из них, конечно, уйдет в прошлое практически сразу, другая часть будет применяться еще долго. Если проанализировать процесс эволюции способов передачи данных, то можно заметить, что некоторые изобретения ознаменовывались бурной технической, а иногда и политической революцией. Достаточно вспомнить изобретение радио, телевидения, сотовой связи, мобильных приложений.

Одном из факторов возникновения беспорядков на Британских островах стала качественная криптографическая защита смартфонов *BlackBerry*. Полиция попросту не могла понять содержание сообщений, передаваемым организаторами беспорядков своим сообщникам.

Феномен "Арабской весны" стал возможен из-за широко распространенных социальных сетей, с помощью которых организаторы беспорядков легко распространяло и получало информацию координирующего характера.

Характерным примером того, что не все новые способы переноса и обработки информации жизнеспособны (можно назвать устройство для хранения информации на ZIP-drive носителе на 100Мб), которые буквально за год-два после появления их на ранке были вытеснены более удобными перезаписываемыми CD на 650/700 Мб.

21.2. Общие сведения об особенностях структуры и функционирования многопроцессорных вычислительных систем и современные средах и системах динамического контроля и отладки параллельных программ

В настоящее время сфера применения многопроцессорных вычислительных систем непрерывно расширяется, охватывая все новые области в самых различных отраслях науки, бизнеса и производства.

Если традиционно многопроцессорные вычислительные системы применялись в основном в научной сфере для решения вычислительных задач, требующих мощных вычислительных ресурсов, то сейчас, из-за

бурного развития бизнеса резко возросло количество компаний, отводящих использованию компьютерных технологий и электронного документооборота главную роль. Непрерывно растет потребность в построении централизованных вычислительных систем для критически важных приложений, связанных с обработкой транзакций, управлением базами данных и обслуживанием телекоммуникаций.

Можно выделить две основные сферы применения описываемых систем: обработка транзакций в режиме реального времени (ОСРВ, или OLTP, ***on-line transaction processing***) и создание хранилищ данных для организации систем поддержки принятия решений (***Data Mining, Data Warehousing, Decision Support System***). Система для глобальных корпоративных вычислений — это, прежде всего, централизованная система, с которой работают практически все пользователи в корпорации, и, соответственно, она должна все время находиться в рабочем состоянии. Как правило, решения подобного уровня устанавливают в компаниях и корпорациях, где любые, даже самые кратковременные, простои сети могут привести к громадным убыткам. Поэтому для организации такой системы не подойдет обыкновенный сервер со стандартной архитектурой, вполне пригодный там, где не стоит жестких требований к производительности и времени простоя. Высокопроизводительные системы для глобальных корпоративных вычислений должны отличаться такими характеристиками, как *повышенная производительность, масштабируемость, минимально допустимое время простоя*.

В настоящее время выделен круг фундаментальных и прикладных проблем, объединенный понятием *«Grand challenges»*, эффективное решение которых возможно только с использованием сверхмощной вычислительных ресурсов. Этот круг включает следующие задачи:

- предсказания погоды, климата и глобальных изменений в атмосфере;
- науки о материалах;
- построение полупроводниковых приборов;
- сверхпроводимость;
- структурная биология;
- разработка фармацевтических препаратов;
- генетика;
- квантовая хромодинамика;
- астрономия;
- транспортные задачи;
- гидро - и газодинамика;
- управляемый термоядерный синтез;
- эффективность систем сгорания топлива;
- геоинформационные системы;
- разведка недр;

— наука о мировом океане;
— распознавание и синтез речи;
— распознавание изображений.

Главной отличительной особенностью многопроцессорной вычислительной системы является ее *производительность*, то есть количество операций, производимых системой за единицу времени. Различают *пиковую* и *реальную производительность*. Под пиковой производительностью многопроцессорной системы понимают величину, равную произведению пиковой производительности одного процессора на число таких процессоров в данной машине, причем, предполагается, что все устройства компьютера работают в максимально производительном режиме.

Пиковая производительность компьютера вычисляется однозначно, и эта характеристика является базовой, по которой производят сравнение высокопроизводительных вычислительных систем. Пиковая производительность есть величина теоретическая и, вообще говоря, не достижимая при запуске конкретного приложения. Реальная же производительность, достигаемая на данном приложении, зависит от взаимодействия программной модели, в которой реализовано приложение, с архитектурными особенностями машины, на которой приложение запускается.

Существуют два способа оценки пиковой производительности компьютера. Один из них опирается на число команд, выполняемых компьютером в единицу времени. Единицей измерения, как правило, является ***MIPS*** (*Million Instructions Per Second*). Производительность, выраженная в ***MIPS***, говорит о скорости выполнения компьютером своих же инструкций. Но, во-первых, заранее не ясно, в какое количество инструкций отобразится конкретная программа, а, во-вторых, каждая программа обладает своей спецификой, и число команд от программы к программе может меняться очень сильно. В связи с этим данная характеристика дает лишь самое общее представление о производительности компьютера.

Другой способ измерения производительности заключается в определении числа вещественных операций, выполняемых компьютером в единицу времени. Единицей измерения является ***Flops*** (***Floating point operations per second***) — число операций с плавающей точкой, производимых компьютером за одну секунду. Такой способ является более приемлемым для пользователя, поскольку последний знает вычислительную сложность своей программы и, пользуясь этой характеристикой, может получить нижнюю оценку времени ее выполнения.

Однако пиковая производительность получается при работе компьютера в идеальных условиях, то есть при отсутствии конфликтов при обращении к памяти при равномерной загрузке всех устройств. В реальных условиях на выполнение конкретной программы влияют такие аппаратно-программные особенности данного компьютера, как:

— особенности структуры процессора,
— системы команд,
— состав функциональных устройств,
— реализация ввода/вывода,
— эффективность работы компиляторов.

Определяющим фактором многопроцессорной вычислительной системы является также *время взаимодействия с памятью*, которое определяется ее

— строением,
— объемом и
— архитектурой подсистем доступа в память.

В большинстве современных компьютеров организации наиболее эффективного доступа к памяти используется так называемая *многоуровневая иерархическая память*. В качестве уровней используются

— регистры и регистровая память,
— основная оперативная память,
— кэш-память,
— виртуальные и жесткие диски,
— ленточные роботы.

При этом выдерживается следующий принцип формирования иерархии: при повышении уровня памяти скорость обработки данных должна увеличиваться, а объем уровня памяти — уменьшаться. Эффективность использования такого рода иерархии достигается за счет хранения наиболее часто используемых данных в памяти верхнего уровня, время доступа к которой минимально. А поскольку такая память обходится достаточно дорого, ее объем не может быть большим. Иерархия памяти относится к тем особенностям архитектуры компьютеров, которые оказывают огромное значение для повышения их производительности.

Для того, чтобы *оценить эффективность работы* вычислительной системы на реальных задачах, был разработан *фиксированный набор тестов*. Наиболее известным из них является ***LINPACK*** — программа, предназначенная для решения системы линейных алгебраических уравнений с плотной матрицей с выбором главного элемента по строке. ***LINPACK*** используется для формирования списка ***Top500*** — пятисот самых мощных компьютеров мира.

В настоящее время большое распространение получили *тестовые программы*, взятые из разных предметных областей и представляющие собой либо модельные, либо *реальные промышленные приложения*. Такие тесты позволяют оценить производительность компьютера действительно на реальных задачах и получить наиболее полное представление об эффективности работы компьютера с конкретным приложением.

Ещё одной из отличительных особенностей многопроцессорной вычислительной системы является *сеть обмена*, с помощью которой процессоры соединяются друг с другом или с памятью. Модель обмена настолько важна для многопроцессорной системы, что многие характеристики производительности и другие оценки выражаются отношением времени обработки к времени обмена, соответствующим решаемым задачам. Существуют две основные модели межпроцессорного обмена: одна основана на *передаче сообщений,* другая — на использовании *общей памяти*. Обе эти модели подробно рассмотрены в параграфе 18.

Рис. 21.1. Мультипроцессорная система с общей памятью.

Первую модель представляют вычислительные системы с общей (разделяемой) основной памятью, которые объединяют до нескольких десятков (обычно менее 32) процессоров. Сравнительно небольшое количество процессоров в таких машинах позволяет иметь одну централизованную общую память и объединить процессоры и память с помощью одной шины. Такой способ организации со сравнительно небольшой разделяемой памятью в настоящее время является наиболее популярным. Структура подобной системы представлена на рис. 21.1.

Вторую модель — крупномасштабные системы с распределенной памятью.

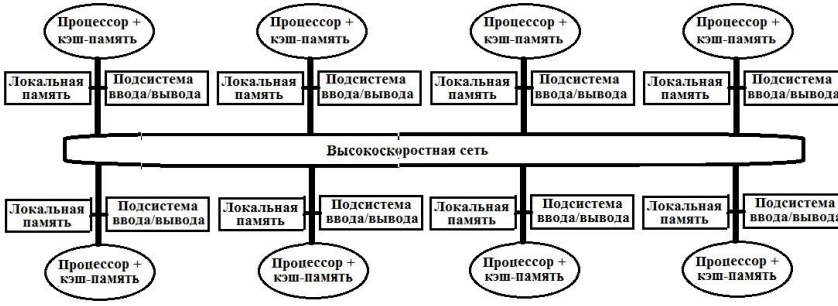


Рис. 21.2. Мультипроцессорная система с распределенной памятью

Для того чтобы поддерживать большое количество процессоров приходится распределять основную память между ними, в противном случае полосы пропускания памяти просто может не хватить для

удовлетворения запросов, поступающих от очень большого числа процессоров. Естественно, при таком подходе также требуется реализовать связь процессоров между собой. Структура такой связи представлена на рисунке 21.2.

Архитектура многопроцессорной вычислительной системы (способ соединения процессоров друг с другом) в бóльшей степени определяет ее производительность, чем тип используемых в ней процессоров. Критическим параметром, влияющим на величину производительности такой системы, является расстояние между процессорами. Так, соединив вместе 10 персональных компьютеров, мы получим систему для проведения высокопроизводительных вычислений, проблема, однако, будет состоять в нахождении наиболее эффективного *способа соединения* стандартных средств друг с другом, поскольку при увеличении производительности каждого процессора в 10 раз производительность системы в целом в 10 раз не увеличится. В этом состоит ещё одна важная особенность архитектуры высокопроизводительных вычислительных систем. Приведём здесь лишь самые употребительные топологии связи процессоров:

1. Топология «решетка» (или «сетка»): пример 4x4 (16 процессоров)

При таком типе топологии соединения *максимальное расстояние* между процессорами окажется равным ***6*** (количество связей, соединяющих процессоры, наиболее отдаленные друг от друга). Теоретические исследования показывает, что при максимальном расстоянии между процессорами больше 4, система не может работать эффективно. Поэтому, при соединении 16 процессоров друг с другом плоская схема «решетка» является неэффективной.

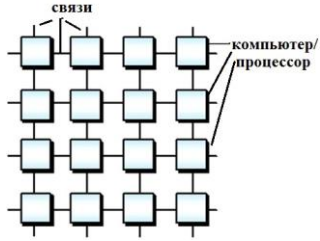


Рис. 21.3.Топология связи «решетка» или «сетка»

Для получения более компактной конфигурации необходимо решить задачу о нахождении фигуры, которая имеет максимальный объем при минимальной площади поверхности.

В трехмерном пространстве таким свойством обладает шар. Но поскольку необходимо построить *узловую систему*, то вместо шара приходится использовать куб (если число процессоров равно ***8*** — рисунок 21.4) или гиперкуб, если число процессоров больше ***8*** (например, для 16 процессоров — рис. 21.5).

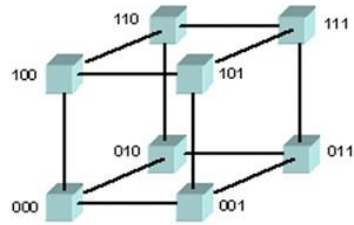


Рис. 21.4. Топология трехмерный куб (8 процессоров)

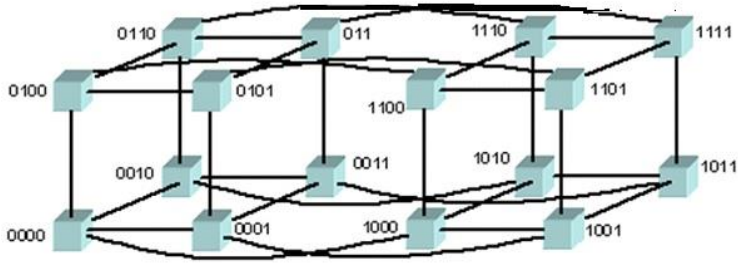


Рис. 21.5. Топология четырехмерный куб (16 процессоров)

Размерность гиперкуба будет определяться в зависимости от числа процессоров, которые необходимо соединить. Так, для соединения 16 процессоров потребуется четырехмерный гиперкуб. Для его построения следует взять обычный трехмерный куб, сдвинуть в еще одном направлении и, соединив вершины, получить гиперкуб размером 4.

Архитектура гиперкуба является второй по эффективности, но самой наглядной. Используются и другие топологии сетей связи: трехмерный тор, "кольцо", "звезда" и другие.

Средства отладки параллельных программ

Для анализа качества параллельных программ существуют специальные средства, использующие интерактивную визуализацию, в частности, анализаторы, которые предназначены для решения следующих основных задач:

- отладки параллельных программ;
- профилирования параллельных программ;
- моделирования параллельных программ.

Отладка параллельной программы (как и последовательной программы) включает в себя функциональную отладку (достижение правильности результатов) и отладку производительности программы.

При разработке параллельных программ основной целью является увеличение быстродействия. Исследования эффективности параллельной программы с целью выявления и устранения «узких мест», а, в конечном счете, с целью уменьшения времени выполнения программы, называется *профилированием* параллельной программы.

Моделированием параллельной программы называется выполнение программы на рабочей станции в режиме эмуляции параллельного выполнения. Целью моделирования является проверка *корректности средств параллелизма* в параллельной программе.

Отладка параллельных программ.

Отладка параллельной программы является процессом существенно более трудоемким, чем отладка последовательной программы. Причинами этого являются

— сложность параллельной программы и
— недетерминированность ее поведения.

Вероятность появления ошибок при написании параллельных программ возрастает, потому что, кроме правильного *функционирования* каждой из параллельных подзадач необходимо обеспечить их правильную *синхронизацию, взаимные коммуникации и исключение тупиков* ***(deadlocks)***.

Отладка параллельной программы требует наличия средств анализа правильности потока управления и потоков данных, а также средств анализа правильности обмена информацией между процессами.

Средства *функциональной отладки* предоставляют пользователю примерно тот же набор примитивов, что и обычные последовательные отладчики:

— инициализация выполнения программы;
— завершение программы;
— приостановка и продолжение выполнения программы;
— задание состояния программы;
— отображение состояния программы;
— модификация состояния программы;
— пошаговое исполнение.

Задание состояния программы состоит в поиске и идентификации пользователем интересующих его данных и программных событий и, в частности, включает в себя трассировку программы.

Отображение состояния программы предполагает предоставление пользователю по его запросам значений переменных и элементов массивов, а также состояний потоков управления и данных.

Модификация состояния программы заключается в изменении состояния указанного пользователем процесса и/или его данных.

Современный технологический цикл отладки параллельной программы включает в себя три следующих этапа:

— *отладка программы на рабочей станции* как последовательной программы (с использованием обычных методов и средств отладки);
— *выполнение программы на той же рабочей станции в режиме эмуляции* параллельного выполнения для проверки корректности средств распараллеливания (моделирование параллельной программы);

— *выполнение программы на целевой параллельной машине* в специальном режиме, при котором промежуточные результаты параллельного выполнения сравниваются с эталонными результатами (например, результатами последовательного выполнения).

21.3. Разнообразие языков параллельного программирования.

Перспективные языки параллельного программирования.

Разработка параллельной программы представляет собой, как правило, сложный, высококвалифицированный и непроизводительный труд. Параллельные программы с большими затратами переносятся между различными архитектурами параллельных вычислительных систем. Эти обстоятельства являются мощным стимулом для создания средств автоматического распараллеливания последовательных программ.

Программирование на последовательном языке программирования и последующее автоматическое распараллеливание выглядит весьма привлекательным, поскольку позволяет:

— применять опыт программирования, накопленный в течение многолетней эксплуатации традиционных последовательных ЭВМ;
— разрабатывать программы, не зависящие от особенностей архитектуры параллельной вычислительной системы;
— использовать программные продукты, накопленные за многие годы создания программного обеспечения для последовательных ЭВМ;
— не тратить силы и средства на параллельную организацию программ, полагаясь что система автоматического распараллеливания решит все вопросы по эффективному переносу программ на любую параллельную систему.

Однако в настоящее время последовательное программирование и последующее автоматическое распараллеливание редко используются в промышленном программировании, поскольку редко обеспечивают достижение приемлемого уровня эффективности параллельных программ.

Существует несколько разных подходов к программированию параллельных вычислительных систем:

— использование *специализированных языков параллельного программирования* и *параллельных расширений* последовательных языков (параллельные реализации ***Fortran*** и ***Cи***);
— использование *средств автоматического и полуавтоматического распараллеливания* последовательных программ (***PGI, BERT 77, FORGE, KAP*** и другие);
— программирование на последовательных языках программирования с *использованием коммуникационных библиотек и интерфейсов* для организации межпроцессорного взаимодействия (***MPI, PVM, OpenMP***);

— программирование на последовательных языках с *использованием параллельных библиотечных процедур* (***ScaLAPACK, HP Mathematical Library, PETSc*** и другие).

Существует также большое количество инструментальных средств, которые *упрощают проектирование параллельных программ* (***CODE, TRAPPER*** и другие).

Преимущественно используется первый из указанных подходов к программированию параллельных вычислительных систем.

Заметим, что языки типа *Ассемблер* отражают в себе систему команд и все особенности структуры параллельной вычислительной системы:

— число процессоров,
— состав и распределение регистров и оперативной памяти,
— систему коммутации и так далее.

Ассемблер потенциально позволяет в наибольшей степени приспособить параллельный алгоритм к особенностям структуры конкретной параллельной ЭВМ и обеспечить наивысшую производительность ЭВМ. Несмотря на это, Ассемблер не используется широко для прикладного программирования параллельных ЭВМ по следующим причинам:

— программирование параллельных ЭВМ на ассемблере является очень трудоемким;
— быстрая смена элементной базы параллельных ЭВМ и самих этих ЭВМ означает переработку всего запаса ассемблерных программ.

Имеется три основных подхода к созданию параллельных языков высокого уровня (ЯВУ).

Первый подход заключается во введении в состав традиционных последовательных ЯВУ средств поддержки параллелизма *на уровне языковых конструкций или библиотечных функций*. Достоинством подхода является простота освоения ЯВУ, а недостатками — противоречие концепций базового языка и параллельных средств, что затрудняет их реализацию и делает менее эффективной.

Второй подход состоит в создании *оригинальных языков параллельной обработки, ориентированных на конкретную архитектуру* или тип параллельных вычислительных систем. специализированных языков параллельного программирования и языков управления процессами, который дал ряд интересных идей по представлению и масштабированию параллельных вычислений (например, ***APL, Symula-67, Sisal, БАРС, Occam, Поляр*** и другие). Достоинства подхода: возможность использования эффективно реализуемых, гибких языковых конструкций. Недостатки подхода: трудность освоения, низкая мобильность.

Третий подход — *создание новых параллельных ЯВУ, не зависящих от конкретной архитектуры* или типа параллельных вычислительных систем

(***АДА***, например). Достоинство подхода состоит в возможности использования в ЯВУ концепций, повышающих эффективность и надежность написанных на этом языке программ; недостатки подхода: в трудности освоения и сложности создания эффективных объектных кодов при трансляции.

Современные подходы в разработке параллельных языков программирования подразумевают усиление структурированности и абстрактности, благодаря чему «начинка» языков всё чаше берёт на себя выполнение задач, которые программистам раньше приходилось решать самостоятельно. Благодаря росту такой автоматизации разработчики могут уделять больше внимания поиску и исправлению багов. Во многих случаях это даже улучшает производительность языков, поскольку автоматизированные алгоритмы эффективнее решают задачи распараллеливания вычислений, не совершая многих простых ошибок.

Сейчас существует пара сотен языков программирования. Какие-то языки умирают, их перестают использовать, другие же, наоборот, постоянно создаются и совершенствуются.

Каждый из этих языков создавался для удобного решения определённого круга задач. Одни функциональнее, другие безопаснее, на одних удобнее писать игры, на других — драйвера. Приведем сначала наиболее востребованные языки программирования на сегодняшний день.

Cи/Cи++. Можно считать одними из наиболее универсальных языков высокого уровня. ***Cи*** один из старейших языков, и на нём часто пишут компиляторы других языков, а также многие фундаментальные продукты такие, как операционные системы, драйвера и приложения.

Java. Язык программирования создавался, как универсальный язык, на котором можно писать программы под любые устройства от компьютера до стиральной машины. Практически безраздельно владеет нишей программ под платформу *Android*.

C-Objective/Swift используется с той же функциональностью, что и ***Java***, но для операционных систем *Apple* (об этом ниже).

HTML/CSS + PHP + JavaScript. В интернете наиболее распространена такая связка языков. Они предназначены для разработки сайтов, но почти не используются где-либо за пределами интернета.

Python. Небольшие скрипты, которые предназначены для обработки файлов удобно писать на ***Python***. Хотя их можно писать и на Си++, но Си++ оказывается мощным, но слишком громоздким языком для малых задач.

Теперь приведем примеры наиболее употребительных и перспективных языков параллельного программирования на сегодняшний момент. Необходимо отметить, что эти языки не являются универсальными. Так, один из языков создан для статистического анализа, другие модернизируют классические языки. Некоторые вообще не являются языками, это просто препроцессоры.

R **—** язык программирования для статистической обработки данных и работы с графикой, а также свободная программная среда вычислений с открытым исходным кодом в рамках проекта ***GNU***. Изначально ***R*** был разработан сотрудниками статистического факультета Оклендского университета *Россом Айхэкой* (англ. *Ross Ihaka*) и *Робертом Джентлменом* (англ. *Robert Gentleman*) (первая буква их имён — ***R***); язык и среда поддерживаются и развиваются организацией ***R Foundation***.

Широко используется как статистическое программное обеспечение для анализа данных и фактически стал стандартом для статистических программ.

В 2010 году ***R*** вошёл в список победителей конкурса журнала *Infoworld* в номинации на лучшее открытое программное обеспечение для разработки приложений.

Java 8. Язык ***Java 8*** отличается от ***Java***, описанного выше**.** В нем появились возможности по внедрению функциональных методик программирования, которые открывают дорогу параллелизму. Использовать их не обязательно, можно оставаться и в рамках ***Java***, тогда нельзя будем повысить структурированность для оптимизации исполнения в среде языка, не будет возможности мыслить в рамках парадигмы функционального программирования и писать более чистый, быстрый код с меньшим количеством багов (ошибок).

Swift **—** усовершенствованный ***Objective-C*** компании *Apple* для разработок под ***MacOS*** или ***iOS***. ***Swift*** по стилю написания гораздо ближе к современным языкам, вроде ***Java*** или ***Python***. Это не просто синтаксически подчищенный ***Objective-C***: в нём более ясный и понятный синтаксис, меньше низкоуровневых операций с указателями. У ***Swift*** обширная спецификация и велико количество нововведений, которое несколько затрудняет его изучение. Есть некоторая трудность: обратная совместимость требует иногда думать о битах и байтах.

Go **—** чистый и простой язык для работы с данными серверных ферм *Google*. В ***Go*** нет сложных абстракций или сложного метапрограммирования — лишь базовые возможности, выражаемые незатейливым синтаксисом. Это облегчает работу в командах, хотя иногда требуются более широкие возможности.

CoffeeScript — это не столько язык, сколько способ сэкономить время на выписывании *JS*-синтаксиса, инструмент для препроцессинга, преобразующий синтаксические сокращения в обычный *JavaScript*. Для некоторых пользователей отсутствие фигурных скобок в *CoffeeScript* облегчает понимание кода, однако, для профессионалов отсутствие их — недостаток, так как скобки помогают лучше разбираться в многочисленных уровнях рекурсии.

D **—** язык программирования с минимальным синтаксисом, базирующийся на Си и Си++ с добавлением современных удобств вроде

управления памятью, вывода типа и граничной проверки (*bounds checking*), структура чётко транслируется в процессор, в него включены некоторые из наиболее важных новых возможностей из других языков. ***D*** называют также «портируемым ассемблером»,

Less.js, как и ***CoffeeScript***, — всего лишь препроцессор, упрощающий создание сложных ***CSS***-файлов. В ***Less.js*** возможна обработка в циклах, переменных и прочих программных конструкциях. Например, создав переменную для оттенка зелёного, используемого одновременно для фона страницы выделения текста. Для изменения цвета для разных компонент достаточно будет внести изменение в одном месте.

Введены и более сложные конструкции, вроде миксинов[35] и вложенных правил, позволяющие эффективно создавать блоки команд стандартной вёрстки, которые потом можно включать в любое количество CSS-классов. Если нужно будет отказаться от некоторого атрибута, можно просто изменить эту настройку в корне, а ***Less.js*** распространит новое правило на все остальные определения.

MATLAB — это язык для математиков и учёных, которым нужно вычислять сложные системы уравнений, выполняют сложные исследовательские работы математического и статистического анализа, и ***MATLAB*** становится всё востребованнее: в нем реализованы стабильные и качественные классические алгоритмы для сложных вычислений. Этот язык оттачивался в течение десятилетий, и теперь он может быть полезен более широкому кругу пользователей, хотя не всегда устраивает реализованная точность вычислений.

Arduino — это не столько новый язык, сколько набор Си и Си++ функций, которые пользователь собирает в строки, а компилятор делает всё остальное.

Многие из функций новы, например, можно считывать вольтаж, проверять статус выводов платы, и даже управлять миганием светодиодов, чтобы передавать определённые сообщения пользователю устройства.

CUDA — язык параллельного программирования компании *Nvidia*, базирующийся на использовании вычислительной мощности видеокарт не только для программирования графических и игровых продуктов.

Чтобы работать с ***CUDA***, необходимо научиться определять параллельные фрагменты разрабатываемого алгоритма, что не всегда легко. Найдя их, можно воспользоваться огромными преимуществами видеокарт в параллельных вычислениях. Некоторые задачи, вроде майнинга криптовалют[36], решаются весьма просто. Другие, например, отбраковка и молекулярная динамика, потребуют крепко поразмыслить.

---

[35] ***Миксин*** (англ. *mix in*) — элемент языка программирования (обычно класс или модуль), реализующий какое-либо чётко выделенное поведение. Используется для уточнения поведения других классов, не предназначен для порождения самостоятельно используемых объектов.

[36] ***Майнинг — п***роцесс добычи ***Bitcoin*** и многих других криптовалют.

CUDA часто используется в научных крупных многомерных симуляциях, так как имеет очень высокую производительность, как минимум для параллельного кода.

Scala **—** наиболее популярный функциональный язык (осуществляет парадигму функционального программирования), предназначенный для исполнения в виртуальной машине ***Java***. ***Scala*** может выполняться там, где работает ***Java***. Функциональный, но достаточно гибкий язык, чтобы хорошо взаимодействовать с другими языками, использующими ***Java***-машину, хотя для ряда задач и приложений может быть непросто использовать функциональный подход.

Haskell **—** функциональный язык с большим количеством пользователей, используется в больших проектах, например, в *Facebook*. В задачах, на первый взгляд, не слишком подходящих для академического кода ***Haskell*** демонстрирует хорошую производительность.

Вопросы для контроля

1. В чем состоит общий принцип построения вычислительных систем? Каковы особенности двух подходов?

2. Что означает параллельность выполнения процессов?

3. Дайте определение производительности, пиковой и реальной производительности вычислительной системы? В каких единицах определяется пиковая производительность?

4. Какие топологии связи процессоров наиболее эффектиавны для многопроцессорных вычислительных систем?

5. Какие подходы существуют для преобразования последовательных программ в параллельные? Какого типа языки используются для каждого из подходов?

Глава 22. Представление о современном системном программном обеспечении.

Учебные цели:

- *дать представление о типовых приемах составления системных программ;*
- *дать представление об основных сведениях о современных высоко производительных вычислительных комплексах.*

22.1. Типовые приемы составления системных программ.

Структурное проектирование

Задачи, решаемые на ЭВМ, являются математическими моделями процессов или явлений реальной жизни. В математической модели находят отражение наиболее существенные связи между реальными объектами. Модели реальных объектов вместе с присущими им связями образуют *структуры данных,* процесс обработки которых и описывается с помощью алгоритмов.

Структурное программирование — методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. Предложена в 70-х годах XX века Э́дсгером Ви́бе Дейкстрой (Нидерланды), разработана и дополнена Никлаусом Виртом (Швейцария).

В соответствии с данной методологией

1. Любая программа представляет собой структуру, построенную из *трёх типов базовых конструкций*:

- *последовательное исполнение* — однократное выполнение операций в том порядке, в котором они записаны в тексте программы;
- *ветвление* — однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия;
- *цикл* — многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла).

В программе базовые конструкции могут быть вложены друг в друга произвольным образом, но никаких других средств управления последовательностью (например, оператор ***GOTO***) выполнения операций не предусматривается.

2. Повторяющиеся фрагменты программы (либо не повторяющиеся, но представляющие собой логически целостные вычислительные блоки) могут оформляться в виде так называемых *подпрограмм* (*процедур* или *функций*). В этом случае в тексте основной программы, вместо помещённого в подпрограмму фрагмента, вставляется инструкция *вызова подпрограммы*. При выполнении такой инструкции выполняется вызванная

подпрограмма, после чего исполнение программы продолжается с инструкции, следующей за командой вызова подпрограммы.

3. Разработка программы ведётся пошагово, методом «сверху вниз».

Алгоритмы большой сложности обычно представляются с помощью схем двух видов: *обобщенной* и *детальной***.**

Обобщенная схема описывает общий принцип функционирования алгоритма и основные логические связи между отдельными этапами.

Детальная схема описывает содержание каждого элемента обобщенной схемы с использованием управляющих структур блок-схемы или псевдокода.

Существует несколько методов проектирования:

— нисходящее (сверху-вниз)
— восходящее (снизу-вверх)
— смешанное, использующее два предыдущих метода.

Нисходящее проектирование предполагает создание сначала обобщенной схемы, а затем детализацию каждого структурного элемента.

Восходящее проектирование предполагает создание сначала детальной схемы для каждого структурного элемента, а затем — обобщенной схемы. Наиболее используемым является *смешанное проектирование*.

22.2. Основные сведения о современных высоко производительных вычислительных комплексах

С ростом числа вычислительных компонентов, включаемых в состав современных высокопроизводительных вычислительных систем (ВВС), существующие подходы, методы и средства для организации системного ПО высокопроизводительных вычислительных систем перестают удовлетворять требованиям. Анализ существующих технологий создания и применения высокопроизводительных вычислительных систем петафлопного уровня (10^{15} flops), а также современных тенденций к построению высокопроизводительных вычислительных систем эксафлопного уровня (10^{18} flops) позволяет сделать вывод о том, что надежность таких систем в рамках используемых на сегодняшний день парадигм и технологий будет являться весьма низкой. В связи с этим актуальным и перспективным направлением исследований в области системного программного обеспечения (ПО) высокопроизводительных вычислительных систем (ВВС) является создание новых подходов, методов и средств управления и мониторинга высокопроизводительных вычислительных систем (ВВС), способных обеспечивать необходимый уровень *отказоустойчивости* и *надежности* для вычислительных сред такого масштаба.

В задачи средств управления высокопроизводительных вычислительных систем (ВВС) входит распределение нагрузки на вычислительные узлы, выполнение непараллельных и параллельных команд с последующей

передачей результата пользователю или оператору, выполнение загрузки и остановки работы вычислительных узлов и ряд других. В задачи средств мониторинга входит сбор данных о работе всех компонентов высокопроизводительных вычислительных систем (ВВС), многокритериальный анализ собираемых данных и в случае обнаружения каких-либо отклонений принятия необходимых воздействий на компоненты.

Мониторинг компонентов высокопроизводительных вычислительных систем (ВВС) условно можно разделить на следующие категории:

1) мониторинг и анализ эффективности выполнения программ в высокопроизводительных вычислительных системах (ВВС)
   —контроль текущего состояния вычислительных процессов и их отдельных экземпляров,
   —оценка эффективности использования выделенных ресурсов;
2) мониторинг, тестирование и диагностика аппаратных компонентов вычислительных узлов
   —жесткие диски,
   —процессоры,
   —оперативная память,
   —сетевые интерфейсы и другое);
3) мониторинг инженерной инфраструктуры высокопроизводительных вычислительных систем (ВВС)
   —системы бесперебойного питания,
   —климатическое оборудование,
   —системы пожаротушения и другое);
4) мониторинг вычислительной инфраструктуры высокопроизводительных вычислительных систем (ВВС)
   —мониторинг текущей загрузки вычислительных узлов,
   —контроль коммуникационных, управляющих и сервисных сетей,
   —систем хранения данных);
5) мониторинг промежуточного программного обеспечения высокопроизводительных вычислительных систем (ВВС)
   — мониторинг функционирования системных служб,
   — очередей задач,
   — агентов,
   — различных подсистем и другое).

Разработка комплексной системы мониторинга, которая обеспечивала бы сбор данных с огромного количества разнородных компонентов, входящих в состав современных высокопроизводительных вычислительных систем (ВВС), является труднореализуемой задачей ввиду отсутствия стандартизованных форматов и протоколов сбора данных со всего множества разнородных программно-аппаратных компонентов высокопроизводительных вычислительных систем (ВВС).

С другой стороны, на сегодняшний день существует огромное количество программных решений, которые в отдельности позволяют обеспечивать мониторинг необходимых компонентов высокопроизводительных вычислительных систем (ВВС0. Более того, многие компоненты высокопроизводительных вычислительных систем (ВВС) уже снабжены системами локального мониторинга. В связи с чем наиболее целесообразным и перспективным направлением развития исследований по созданию комплексных систем мониторинга высокопроизводительных вычислительных систем (ВВС) является агрегация существующих локальных систем мониторинга в рамках комплексной системы мета-мониторинга высокопроизводительных вычислительных систем (ВВС). При этом локальная система мониторинга выступает лишь поставщиком данных, а их экспертный анализ и принятие на основе результатов анализа необходимых регулирующих воздействий отводится системе мета-мониторинга.

Рис. 22.1. Общая схема интеграции локальной системы мониторинга в состав системы мета-мониторинга

В качестве локальных систем мониторинга могут быть использованы как небольшие утилиты для сбора данных об отдельном компоненте высокопроизводительных вычислительных систем (ВВС), так и комплексные системы мониторинга, агрегирующие информацию по набору компонентов и вычислительных узлов.

Вопросы для контроля

1. Дайте определение структурного программирования.

2. Что собой представляют обобщенная и детальная схемы функционирования алгоритма?

3. Что входит в задачи средств управления высокопроизводительных вычислительных систем?

4. В чём состоит мониторинг компонентов высокопроизводительных вычислительных систем?

Заключение. Перспективы совершенствования СПО

Перспективы развития современного программного обеспечения определяются множеством факторов. В настоящем пособии подробно разбирались многие из них. Сформулируем в этом параграфе основные тенденции.

Прежде всего отметим, что прогресс в ускорении работы программного обеспечения базируется не только на возможности улучшения и оптимизации архитектуры суперкомпьютерных вычислительных систем: увеличения их производительности и совершенствовании моделей архитектуры внутренних связей процессоров, но в большей степени на способах параллельной обработки информации, предполагающей одновременное выполнение всех или нескольких шагов задачи.

Дальнейшие идеи параллелизма развиваются по двум направлениям:

1. программирование нужной архитектуры ЭВМ;
2. объединение в одной ЭВМ разных методов организации параллелизма.

Программирование нужной архитектуры реализуется в соответствии с особенностями алгоритмов, которыми характеризуется класс предполагаемых задач (научные, экономические, поисковые, инженерные).

Объединение разных методов в одной ЭВМ перспективно в том случае, если возможно распределение частей задач между различными подсистемами. Тогда возможны комбинации методов, например, параллелизм программ и данных или параллелизм потоков команд и данных и так далее.

Улучшение архитектуры проявляется не только в параллелизме, но и постепенном переходе к нелинейной памяти.

Совместное хранение программ и данных в линейном пространстве памяти усложняет понимание смысла программ. Существует огромная смысловая разница между операциями на языке программирования высокого уровня и операциями, определяемыми архитектурой ЭВМ. Отсюда невозможно выяснить природу возникающих ошибок: ошибка связана с командой или с данными.

Отказ от линейной памяти предполагает использование алгоритма, с помощью которого можно добавлять ко всем данным дополнительную информацию, использующуюся для распознавания данных. Это является серьезным отступлением от классических неймановских идей, так как управление вычислительным процессом зависит не только от потока данных. Развитие программного обеспечения выдвигает более серьезные причины отказа от линейной памяти.

Повышение производительности ЭВМ осуществляется не только за счет параллелизма, но и за счет повышения тактовой частоты. Еще больше производительность возросла в вычислительных системах с массовым параллелизмом на основе объединения тысяч микропроцессоров.

Параллельно с развитием суперсистем идет быстрое развитие микропроцессорной техники, в котором выделяют два направления:

— разработка и создание микропроцессорных средств общего применения, являющихся основой для построения персональных ЭВМ;
— разработка и создание микропроцессорных средств, ориентированных на конкретные области применения (управление оборудованием, транспортом, связью и так далее).

Особое внимание уделяется миниатюризации микропроцессов и их производительности. В перспективе на кристаллах, содержащих сотни миллионов транзисторов, можно достичь производительности сотен TFLOPS или даже ExFLOPS.

Как известно, программное обеспечение играет важнейшую роль в функционировании всей вычислительной системы. Поэтому повышение эффективности вычислительной системы требует совершенствования всех направлений программного обеспечения.

По характеру использования программного обеспечения в конкретных приложениях их можно разделить на

— программное обеспечение систем реального времени и
— программное обеспечение микро-ЭВМ, которым нет необходимости работать в масштабе реальных событий, но которые должны обладать развитыми функциональными возможностями.

В программном обеспечении реального времени (ПО РВ) главный упор делается на

— повышение эффективности обработки информации в самые короткие промежутки времени,
— на организацию параллельной обработки многих задач,
— на повышение скорости реагирования на прерывания внешних устройств.

С этой целью разрабатываются новые структуры программного обеспечения реального времени, совершенствуются традиционные компоненты системы:

- планировщик задач,
- диспетчер,
- обработчик прерываний,
- программы отслеживания времени,
- супервизоры ввода/вывода.

В результате даже не очень быстрый микропроцессор может быть использован в системе реального времени, так как программное обеспечение реального времени осуществляет эффективную обработку поступающей информации.

В микро-ЭВМ, предназначенных для решения различных задач не в реальном времени, программное обеспечение совершенствуется в направлении облегчения процесса программирования задачи и увеличения скорости работы на ЭВМ.

Как правило, это работа по развитию языков высокого уровня применительно именно к микро-ЭВМ. Традиционные языки высокого уровня, используемые на больших ЭВМ, в большинстве своем неприменимы для микро-ЭВМ, так как требуют большого объема памяти, развитых операционных систем и так далее. Были созданы и создаются свои типы языков, которые хотя и многое позаимствовали у языков больших машин, но сохраняют особенности, присущие именно языкам микро-ЭВМ. Среди критериев, которыми необходимо руководствоваться при выборе языка, существуют следующие:

- обработка прерываний на уровне данного языка,
- работа с подпрограммами (то есть, возможность модульного программирования),
- выбор трансляции или интерпретации,
- кросс-средства для отладки программного обеспечения,
- структуры данных
    - разрядность,
    - форма представления,
    - побитовая обработка и т.п.),
- объем требуемой памяти,
- временные соотношения при работе с языком,
- переносимость программного обеспечения с одного типа микро-ЭВМ на другой.

В информационных технологиях стремительно развивается эпоха интеллектуализации программного обеспечения, которая началась с интеллектуализации «железа», когда системное и проблемное программное обеспечение стало важнее самого оборудования. Увеличивается скорость передачи данных и пропускная способность, изображения и видео файлы, обмен которыми постоянно происходит во «всемирной паутине», требуют более высокой пропускной способности, расширяются возможности коммуникационных технологий в масштабе реального времени: все больше появляется сетевых приложений, требующих взаимодействия в реальном времени.

В области программного обеспечения существует две тенденции, постепенно меняющие взгляды на само программное обеспечение:

- переход от эпохи алгоритма к эпохе модели.
- переход от закрытых программных систем к открытым.

Модель определяет, что надо вычислять, а алгоритм — как. Очевидно, что без того и другого не обойтись. Однако модели оттесняют алгоритм, превращаясь из пассивных элементов в активные. Примером тому могут служить технологии обработки знаний, представленные фреймами. В этом случае процесс обработки зависит от информации, которая имеется в

определенных слотах, так называемых, демонах. Демоном называется процедура, автоматически запускаемая при выполнении некоторого условия.

Для того, чтобы программный продукт был конкурентоспособным, он должен обладать:

— способностью к *переносимости* прикладных программ на различные платформы ЭВМ;
— способностью к *унифицированному обмену данными* между различными платформами ЭВМ;
— *возможностью замены* одного компьютера на другой без каких-либо затруднений.

Открытость программных систем достигается путем разработки стандартов взаимодействия систем.

В связи с необходимостью отражения в языках программирования новых возможностей ЭВМ сформировались три подхода к их развитию:

1) расширение существующих языков.
2) создание новых языков для конкретных типов машин.
3) создание новых языков, не ориентированных на конкретную вычислительную систему.

Ярким представителем последнего подхода, например, является язык ***Java*** **—** один из языков, упомянутых в параграфе 21 в качестве одного из группы наиболее востребованных. Это простой, объектно-ориентированный, распределенный, переносимый, многопоточный и динамичный язык. Ближайшее будущее несомненно за этим такими языками.

Совершенствование систем управления базами данных (СУБД) определяется ориентацией на объектное программирование. Реляционные СУБД представляли собой значительный прогресс в технологии управления данными. Однако, они оказались неудобными из-за необходимости приведения данных к нормальной форме. В результате терялась семантика (смысл) данных. С развитием объектно-ориентированного подхода появилась возможность описывать не только сложные структуры данных, но также поведение объектов реального мира. Одна из задач — это перевод реляционных баз данных в объектно-ориентированные, другая — разработка новых подходов, где в разрабатываемые базы данных включаются не только неформатированные элементы и полнотекстовые фрагменты, но также базы данных с геоинформацией, мультимедийные БД, и это не исчерпывающий перечень.

К основным операционным системам, широко используемым в России, относятся, *ОС Netware* фирмы *Novell, ОС Windows* фирмы *Microsoft* и *Unix*. Первая из них планирует разработку средств, позволяющих работать с сетью ***Internet***. Фирма *Microsoft* планирует предоставить своим пользователям кроме мультизадачного режима службу справочников, квотирование дискового пространства и усовершенствованные средства удаленного

управления данными. Если в *ОС Netware* делается ставка на язык ***Java***, то в *ОС Windows* — на язык ***SQL***. В российском программном обеспечении делается ставка на переоснащение программных продуктов с переносом или разработкой на платформу ***Unix***-оидов (например, ***ОС Linux, AstroLinux***).

Совершенствование программного обеспечения зависит от уровня развития средств технического обеспечения и потребностей пользователей. В дальнейшем предусматривается изменение операционной системы для перехода на мультипрограммную работу с переменным числом заданий и разделение времени с включением телекоммуникационного метода доступа. Для совершенствования обслуживания прикладных систем, повышения надежности и эффективности разработок предполагается внедрить в промышленную эксплуатацию ряд пакетов программного обеспечения.

Интенсивность совершенствования программного обеспечения зависит

- от степени, предусмотренной (встроенной) ремонтопригодности,
- качества поддержки,
- числа установленных систем,
- частоты выпуска новых версий и
- объема переписанных программ.

В целом совершенствование программного обеспечения ставит перед собой следующие задачи:

- диалог человек - машина на любом языковом уровне;
- автоматическое исправление ошибок пользователей;
- получение пользователем информации любой степени подробности о состоянии вычислительного процесса и обрабатываемых данных;
- широкое использование принципа самоопределяемости данных;
- почти полное отсутствие ограничений на выбор удобного для пользователей представления предложений языка;
- объединение и упрощение языков программирования, их ориентацию на структурное программирование;
- схемная реализация программного обеспечения (его наиболее часто используемой части);
- изменение структуры операционной системы с целью ее иерархической конфигурации, включающей ядро; использование проблемно-ориентированных систем программирования;
- генерация программного обеспечения для решения классов задач;
- оптимизация программного обеспечения;
- комплексное рассмотрение проблем рассматриваемой области.

СПИСОК ЛИТЕРАТУРЫ

1. Ахо А., Лам М., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструментарий. 4-е издание, М.: ООО “И.Д. Вильямс”, 2008, 1184 с.
2. Грис Д. Конструирование компиляторов для цифровых вычислительных машин. - М.: Мир, 1975, 544с.
3. Кнут Д. Искусство программирования для ЭВМ. Т.1. Основные алгоритмы. 3-е издание. М., "Вильямс", 2017, 720 с.
4. Пратт Т., Зельковец М. Языки программирования: разработка и реализация. 4-е издание, СПб.: Питер, 2002. – 688 с.
5. Братчиков И. Л. Синтаксис языков программирования. М.: Мир, 1978.
6. Вояковская Н. Н., Москаль А. Е., Булычев Д. Ю., Терехов А. А. Разработка компиляторов. – М.: Национальный Открытый Университет “Интуит”, 2016.
7. Гордеев А. В., Молчанов А. Ю. Системное программное обеспечение. СПб.: “Питер”, Эл. издание Челябинск ЮУрГУ, каф “Автоматика и управление”, 2002. – 475 с.
8. Мартыненко Б. К. Языки и трансляции: Учеб. пособие. 2-е изд., испр. и доп., СПб.: Издательство С.-Петербургского унив., 2013, 268 с. (электронный вариант учебного пособия: Единое окно Санкт-Петербургский государственный университет.
URL: http://www.math.spbu.ru/user/mbk)https://www.)
9. Молдованова О. В. Языки программирования и методы трансляции. Учебное пособие. – Новосибирск / СибГУТИ, 2012. - 134с.
10. Молчанов А. Ю. Системное программное обеспечение: Учебник для вузов. – СПб.: “Питер”, 2010. – 400с.: ил.
11. Молчанов А. Ю. Системное программное обеспечение: Лабораторный практикум. – СПб.: Питер, 2009. – 370 с.: ил.
12. Опалева Э. А., Самойленко В. П. Языки программирования и методы трансляции. СПб.: БХВ-Петербург, 2005. - 408 с.: ил.
13. Пентус А., Пентус М. Математическая теория формальных языков. Московский государственный университет имени М.В. Ломоносова. Национальный Открытый Университет «ИНТУИТ»
URL: https://www.intuit.ru/studies/courses/1064/170/info

Реализация программы *перевода выражения из инфиксной записи в обратную польскую запись на Си++*

Реализация алгоритма Дейкстры. Для перевода выражения в обратную польскую запись необходимо запрограммировать только таблицу, а для счета — запрограммировать цикл, стек и пару условий. Формальные требования к разработке заголовочного файла, стека и графического интерфейса разбирать не будем. Организацию стека описывали ранее в главе 10 пункт 10.2 пример 10.4. Поэтому рассмотрим лишь конкретный перевод. Все исходные коды рабочей программы приведены ниже.

Перевод из инфиксной нотации в постфиксную.

Не нарушая общности, для упрощения элементы математического выражения будем вводить через пробел. Для этого создадим функцию ***convert***, которая в качестве формального параметра принимает строку в инфиксной нотации, а, как результат, возвращает строку в постфиксной нотации на Си++:

Код программы на Си++

```
QString MainWindow::convert(QString inf_string)
 {QStringList inf, post;
  stk->clear();
  inf=(inf_string+" $").split(" ");//разбиваем строку по пробелам
  int status=2; /*Переменная для 3-х состояний выполнения преобразований:
0 – не выполнено, ошибка; 1 - выполнено успешно; 2 - в процессе*/
   int i=0;
   QString first; //здесь будет храниться последний элемент в стеке
   while(status = = 2)
    {first = stk -> get_first_element();/*элемент на вершине стека
                                  (последний добавленный в стек элемент)*/
     QRegExp rx("^\\d+\.?\\d*$"); //шаблон числа
     if (rx.indexIn(inf[i]) = = 0)
       {post.append(inf[i]); i++;} /*проверка, является ли элемент числом*/
     if (inf[i] = = "+" || inf[i] = = "-")      //если в буфере + или -
          { if (first = = "-1" || first = = "(" )  { stk->add(inf[i]); i++;}
               else if (first = = "+" || first = = "-" || first = = "*" || first=="/")
                         post.append(stk->take());
          }
        else if (inf[i] = = "*" ||inf[i]=="/")   //если в буфере * или /
              {if (first = = "-1" || first = = "(" || first = = "+" || first = = "-")
                          { stk->add(inf[i]); i++;}
                 else if (first = = "*" || first = = "/")    post.append(stk->take());
              }
```

```
else  if (inf[i] = = "("){ stk->add(inf[i]); i++;} //если в буфере
                    открывающая скобка, то добавляем её в стек
          else if (inf[i] = = ")")  //если в буфере закрывающая скобка
            { if (first = = "-1")  status=0; /* если в буфере закрывающая
скобка, а в стеке нет элементов - изменяем статус на 0 – ошибка*/
                  else if(first=="+" || first=="-" || first=="*" || first=="/")
                                post.append(stk->take());
                        else if (first = = "(")  {stk->take(); i++;}
        } else if (inf[i] = = "$")   //если последний элемент
              { if (first = = "-1")  status=1; //преобразование из
                  инфиксной нотации в постфиксную завершено
    else if (first = ="+" || first = ="-" || first = ="*" || first = = "/")
                                post.append(stk->take());
                          else if (first == "(")  status=0; /* если в
буфере последний символ, а в стеке есть открывающая скобка – ошибка*/
     }else status=0; //неизвестный символ
  }
 if (statu s = =  1)   return post.join(" ");
    else return "0000";
}
```

Пояснения.

stk — объект класса ***stack***, который хранит в себе элементы строкового типа данных ***QString***, в данном контексте предназначен для хранения операторов.

Счетчик ***i*** увеличиваем только в том случае, если символ, который стоит в буфере, направляется в стек или в вывод. Чтобы понять, как работает данный код, прокомментируем более подробно следующий кусок:

```
if(inf[i]=="+"||inf[i]=="-") //если в буфере + или -
 { if(first=="-1" || first=="("){ stk->add(inf[i]); i++;}
   else if(first=="+" || first=="-" || first=="*" || first=="/")
                          post.append(stk->take());
 }
```

Допустим в буфер пришел символ с «+». Согласно добавить в стек можно только, либо

- — если в данный элемент стек пустой,
- — если последний добавленный в стек элемент — это открывающая скобка «(».

После добавления элемента в стек переходим к следующему шагу. За все описанное отвечает следующая строка:

```
if(first==»-1" || first==»(«){ stk->add(inf[i]); i++;}.
```

Если в буфере находится «+», а на вершине стека «+», «-», «*» или «/», то элемент на вершине стека (последний добавленный в него элемент) необходимо направить в вывод (в строку где формируется обратная польская запись). За это отвечает следующая строка:

if(first==»+» || first==»-» || first==»*» || first==»/») post.append(stk->take());.

Теперь *вычисление обратной польской записи*.

Функция ***calc***, в качестве формального параметра принимает строку с выражением в постфиксной нотации, как результат, возвращает число формата ***float***:

```
float MainWindow::calc(QString postf)
 {QStringList post;
  stk_digit->clear();
  post=postf.split(" ");/*разбиваем строку в постфиксной записи по пробелам*/
  float var_1,var_2;
  for(int i=0;i<post.count();i++)
  {QRegExp rx("^\\d+\.?\\d*$");
    if (rx.indexIn(post[i]) = = 0) stk_digit->add(post[i].toFloat());
     else if (post[i]= ="+")  stk_digit->add(stk_digit->take()+stk_digit->take());
//сложение двух последних элементов
           else if (post[i] = = "-")
             { var_2=stk_digit->take();
               var_1=stk_digit->take();
               stk_digit->add(var_1-var_2);/*вычитание из предпоследнего
                                                элемента последний*/
             }
          else if (post[i] = = "*")
             stk_digit->add(stk_digit->take()*stk_digit->take());//перемножаем
                                                два последних элемента*/
                 else if (post[i] = = "/")
                        { var_2=stk_digit->take();
                          var_1=stk_digit->take();
                          stk_digit->add(var_1/var_2);/* деление
                                предпоследнего элемента на последний*/
                        }
       }
    return stk_digit->take();
}
```

stk_digit — объект класса ***stak_digit***, который хранит в себе элементы типа данных ***float***, в данном контексте предназначен для хранения операндов.

Замечание. Эти функции являются методами класса *MainWindow*, поэтому вместо

float calc(QString postf) писали ***float MainWindow::calc(QString postf)***.

Пример работы программы.

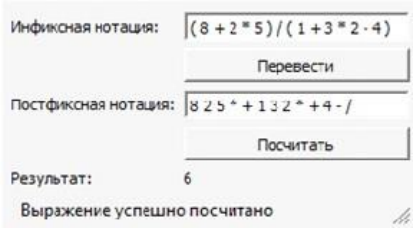
Инфиксная нотация:

(8+2*5)/(1+3*2-4)

Постфиксная запись:

8 2 5 * + 1 3 2 * + 4 - /

Результат вычислений: ***6***



ОГЛАВЛЕНИЕ