М. А. Бульонков, П. Г. Емельянов, И. Н. Скопин

БАЗОВЫЕ ПОНЯТИЯ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

М. А. Бульонков БАЗОВЫЕ ПОНЯТИЯ
П. Г. Емельянов, И. Н. Скопин И МЕТОДЫ ПРОГРАММИРОВАНИЯ

ISBN 978-5-4437-1495-0

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

М. А. Бульонков, П. Г. Емельянов, И. Н. Скопин

БАЗОВЫЕ ПОНЯТИЯ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Новосибирск
2023

УДК 004(075.8)
ББК В185.0я73-1
Б908

Рецензент:
Андрей Николаевич Терехов, д-р физ.-мат. наук,
проф., зав. кафедрой системного программирования
Санкт-Петербургского государственного университета

Бульонков, М. А.
Б908 Базовые понятия и методы программирования : учеб. пособие / М. А. Бульонков, П. Г. Емельянов, И. Н. Скопин ; Новосиб. гос. ун-т. – Новосибирск : ИПЦ НГУ, 2023. 366 с.

ISBN 978-5-4437-1495-0

Учебное пособие охватывает круг вопросов, связанных с современными методами и понятиями программирования и технологическими аспектами разработки программного обеспечения, которые включают: принципы построения языков программирования, базовые структуры данных и приемы программирования, принципы параллельного программирования и сложности алгоритмов.

Пособие предназначено для студентов первых курсов математических направлений подготовки укрупненных групп 01 и 02.

УДК 004(075.8)
ББК В185.0я73-1

Учебное пособие разработано в соответствии с требованиями ФГОС ВПО к профессиональному циклу основных образовательных программ по группе 01 направлений подготовки (математика, механика, информатика) и группе 02 (математика и компьютерные науки).

ISBN 978-5-4437-1495-0

© Новосибирский государственный университет, 2023

ОГЛАВЛЕНИЕ

Предисловие ... 7
Введение ... 9
1. Прикладное программирование ... 9
2. Системное программирование ... 12
3. Технология программирования ... 12
4. Теоретическое программирование ... 15
1. «Окружение» программирования ... 16
1.1. Логическая модель ЭВМ ... 16
1.2. Операционная система ... 17
2. Поколения языков программирования ... 20
2.1. Машинный язык ... 20
2.2. Мнемокод ... 20
2.3. Макроассемблер ... 21
2.4. Алгоритмические языки высокого уровня ... 24
2.4.1. Императивные языки ... 25
2.4.2. Функциональные языки ... 26
3. Реализация языков программирования ... 28
3.1. Интерпретаторы ... 28
3.2. Трансляторы ... 29
3.3. Т-диаграммы ... 29
4. Системы программирования ... 36
5. Языки программирования ... 44
5.1. Лексика ... 47
5.2. Синтаксис ... 54
5.2.1. Форма Бэкуса–Наура (БНФ) ... 54
5.2.2. Синтаксические диаграммы ... 61
5.2.3. Устойчивость синтаксиса ... 63
5.3. Абстрактный синтаксис ... 65
5.4. Контекстно-зависимый анализ ... 69
5.5. Семантика ... 71
5.5.1. Денотационная семантика ... 71
5.5.2. Операционная семантика ... 74
5.5.3. Аксиоматическая семантика ... 79
5.6. Стиль ... 83

5.6.1. «Лесенка» ... 83
5.6.2. Неиспользование умолчаний ... 86
5.6.3. Мнемоничные идентификаторы ... 87
5.6.4. Комментарии ... 88
5.7. Прагматика ... 89
5.8. Преемственность ... 90
6. Препроцессор ... 92
6.1. Синтаксис ... 93
6.2. Макросы и вызовы ... 94
6.3. Включение файлов ... 102
6.4. Условная трансляция ... 103
6.5. Генерация лексем ... 108
7. Объекты и типы ... 111
7.1. Области видимости ... 111
7.2. Типы данных ... 115
7.2.1. Анализ типов ... 117
7.2.2. Классификация типов ... 121
7.2.3. Логические типы ... 122
7.2.4. Символы ... 122
7.2.5. Целые числа ... 124
7.2.6. Вещественные числа ... 126
7.2.8. Множества ... 130
7.2.8. Перечисления ... 131
7.2.9. Структуры ... 133
7.2.10. Объединения ... 134
7.2.11. Указатели ... 136
7.2.12. Массивы ... 137
7.2.13. Строки ... 144
7.3. Типы данных в языке С ... 145
7.3.1. Целые и символы ... 145
7.3.2. Логические типы ... 147
7.3.3. Битовые шкалы ... 148
7.3.4. Перечисления ... 150
7.3.5. Вещественные числа ... 151
7.3.6. Приведение типов ... 152
7.3.7. Указатели ... 153
7.3.8. Массивы ... 156
7.3.9. Строки ... 157

7.3.10. Нетипизированные указатели и sizeof 159
7.3.11. Описания, структуры и объединения 161
7.3.12. Присваивания 165
7.3.13. Нотационная путаница 170

8. Управление 172
8.1. Выражения 173
8.2. Операторы 177
8.2.1. Базовые операторы и блоки 177
8.2.2. Метки и `goto` 178
8.2.3. Ветвления 181
8.2.4. Циклы 187
8.3. Процедуры и функции 196
8.3.1. Описание функций 196
8.3.2. Вызов функции 196
8.3.3. Рекурсия 199
8.3.4. Реализация функций 202
8.3.5. Хвостовая рекурсия 209
8.3.6. Вложенные процедуры 211
8.3.7. Оптимизации 212
8.3.8. Функциональные значения 215
8.3.9. Подстановка параметров 217
8.4. Обработка исключительных ситуаций 230

9. Распределение памяти 239
10. Ввод-вывод 249
11. Оценка ресурсоемкости программ 256
11.1. Элементы теории сложности 256
11.2. Понятия сложности в «простой» вычислительной модели (машина Тьюринга) 257
11.3. Пример: подсчет количества бит 259
11.4. Реализация, пессимизация, оптимизация 265
11.5. Организация информационных множеств 268
11.5.1. Общие понятия 269
11.5.2. Битовые шкалы 272
11.5.3. Массивы 274
11.5.4. Деревья поиска 278
11.5.5. Хэш-таблицы 295

12. Специальные методы программирования 303

12.1. Программы, управляемые данными (Data-driven programming) ... 303
12.2. Предметно-ориентированное программирование (Domain-specific languages) ... 306

13. Параллельные вычисления ... 312
13.1. Последовательные, параллельные и совместные вычисления ... 312
13.2. Разработка параллельных программ ... 314
13.3. Порождение параллельных процессов и передача сообщений между процессами ... 321
13.4. Механизм send-receive ... 323
13.5. Событийный механизм и параллелизм ... 329
13.6. Синхронизация параллельных вычислений ... 331
13.7. Взаимодействие поставщиков и потребителей данных ... 333
13.7.1. Локальная копия ... 334
13.7.2. Синхронизация ... 335
13.7.3. Комбинированный способ ... 336
13.8. Мониторы ... 337
13.9. Специализированные параллельные вычислительные системы ... 340
13.10. Классификация архитектур вычислительных систем ... 341
13.10.1. SISD (single instruction stream over a single data stream) ... 342
13.10.2. SIMD (single instruction stream over a multiple data stream) ... 343
13.10.3. MISD (multiple instruction stream over a single data stream) ... 345
13.10.4. MIMD (multiple instruction stream over a multiple data stream) ... 346
13.10.5. Поддержка распределенной и разделяемой памяти ... 348
13.10.6. Квазипараллельные системы ... 350
13.10.7. Разделение времени ... 352
13.10.8. Системы с дискретными событиями ... 353

Литература ... 357
Предметный указатель ... 361

ПРЕДИСЛОВИЕ

Это пособие основывается на конспектах лекций, которые в разных вариантах читались на первых курсах ММФ Новосибирского госуниверситета в течение последних пятнадцати лет. Отправной точкой в развитии предлагаемого курса является пособие М. М. Бежановой и И. В. Поттосина «Современные средства и методы программирования» [3], написанное в конце прошлого века. Авторы этой книги совместно с академиком А. П. Ершовым стояли у истоков преподавания программирования в НГУ и отразили в ней свой опыт, а также опыт своих коллег. Понятно, что многое поменялось за это время: появлялись и сходили на нет языки программирования, кардинально повысилась мощность и, что еще важнее, доступность вычислительных машин, формировались и приобретали первостепенное значение новые области применения и новые методы программистского мышления. Как следствие, изменялось и представление о профессиональных знаниях, умениях и навыках, которыми должны обладать программисты. Поэтому сегодня курс, в котором излагаются базовые понятия и методы программирования, нуждается в обновлении.

Данный курс не претендует ни на полноту, ни на соответствие последним тенденциям современного программирования, а ставит целью критическое осмысление базовых понятий и их выражение в различных языках программирования. Предполагается, что курс закладывает основу для дальнейшего освоения объектно-ориентированного программирования, методов работы с базами данных, разработки пользовательских интерфейсов, вычислительной математики, машинной графики и многих других областей. С другой стороны, ожидается, что слушатель обладает знаниями в объеме школьного курса «Основы информатики» и соответствующими навыками работы с компьютером. Предполагается, что обучаемый способен самостоятельно составлять простые программы.

Одним из самых спорных вопросов, который возникает при постановке подобного курса, является вопрос о выборе языка программирования. Не вдаваясь в детали этого обсуждения, сразу ска-

жем, что в данном курсе это язык С. Однако ни в коем случае наше пособие не следует рассматривать как справочное руководство по языку. Мы считаем, что осваивать конкретный язык образованный программист должен самостоятельно в процессе решения практических задач, пользуясь соответствующей технической документацией. Нам же язык С требуется для демонстрации конструкций и понятий из «лексикона» программирования, а там, где его окажется недостаточно, мы будем привлекать и другие языки, такие как Паскаль, Алгол-60, Фортран, АПЛ, Алгол-68 и пр. Наша задача не столько в том, чтобы читатель в совершенстве освоил конкретный язык или стал компьютерным полиглотом, а в том, чтобы он научился ставить вопросы: «Почему так и как можно было бы по-другому?»

ВВЕДЕНИЕ

Программирование можно определить как способ заставить кого-то достичь поставленной кем-либо цели. Часто в качестве примера программирования приводят кулинарные рецепты, которые описывают пошаговый процесс приготовления некоторого блюда из исходных продуктов. Программой может служить и математический алгоритм. Например, алгоритм Евклида задает последовательность действий, которые достаточно выполнить для нахождения наибольшего общего делителя двух заданных целых чисел. Однако следует иметь в виду, что последовательность и пошаговость не является неотъемлемой чертой программирования. Даже в случае кулинарных рецептов некоторые операции можно выполнять в произвольной последовательности или одновременно, как говорят программисты, – параллельно. Если же в качестве примера программы обратиться к Правилам дорожного движения, то можно заметить, что их цель – обеспечение безопасности и избежание заторов на дорогах – в большей степени достигается не явным предписанием того, что и в какой последовательности надо делать, а формулировкой ограничений – того, чего делать не надо. Этот пример демонстрирует и то, что исполнителей программы может быть несколько. К программированию можно отнести описание технологических процессов, рекламу, уставы организаций и многое другое. Но нас в рамках учебного пособия будет интересовать в первую очередь программирование для компьютера (ЭВМ).

Программирование как вид деятельности может иметь весьма разные аспекты. Условно выделим четыре вида программирования, которые рассмотрим ниже.

1. Прикладное программирование

Прикладные (или пользовательские) программы составляют конечную и основную цель программирования, поскольку именно они влияют на повседневную жизнь обычных людей. Этим объясняется и огромное многообразие таких программ. Сюда можно отнести, например, текстовые процессоры и электронную почту,

игровые программы, бухгалтерские и банковские системы, автомобильные навигаторы, интернет-магазины, встроенные программы управления бытовой техникой и многое-многое другое. Соответственно, основное внимание в прикладном программировании должно уделяться вопросам *надежности* конструируемых программ, их *безопасности* и *интуитивной понятности*, а также *эффективности*, т. е. производительности вычислений. Кроме того, программа должна удовлетворять *гуманитарным требованиям*, связанным с особенностями тех, кто ее использует. Понятно, что эти требования могут относиться не только к прикладному программированию, но именно для него они являются ключевыми.

Ниже приводится небольшой комментарий, раскрывающий мотивировки представленных требований:

- *Надежность*, устойчивость к условиям эксплуатации: постоянно ломающаяся, «зависающая» программа не представляет никакой ценности для пользователя;
- *Безопасность*, «защита от дурака». Программа может быть не только полезной, но и вредной. Даже если не рассматривать компьютерные вирусы и другое целенаправленное нанесение ущерба, должны быть предусмотрены все возможные пользовательские сценарии, в том числе и «неразумные», но все-таки возможные;
- *Интуитивная понятность* того, как программа выполняет заложенные в ней функции и как это выполнение активизируется посредством пользовательских воздействий (связано, обусловлено ими). Последнее обычно характеризуется как удобство и эффективность пользовательского интерфейса. Взаимодействие программы с пользователем должно формулироваться в понятных ему терминах и соответствовать той деятельности, для которой программа используется. Эффективность интерфейса подразумевает, в частности, минимизацию суммарного количества действий (не только механических, но и, в частности, умственных!), которые необходимы для достижения цели. Понятно, что действия могут различаться по сложности: от простейших – нажатий на клавиши или перемещения курсора, до выбора элемента в длинном неупорядоченном списке и т. п.;

- *Эффективность*. Программа должна обладать адекватным быстродействием. Адекватность здесь понимается как соответствие времени исполнения и других потребляемых программой ресурсов ожиданиям пользователя. Очевидно, что эти ожидания меняются со временем – по мере развития вычислительной техники там, где раньше пользователи готовы были ждать час, теперь их раздражает задержка в несколько секунд. Уместно отметить и другой аспект зависимости пользовательских ожиданий от времени, в которое реально применяется программа. Очень часто, а для реально хороших программ практически всегда, в течение срока ее эксплуатации нарастает потребность применения программы для все более сложных задач, которые требуют более эффективных вычислений;
- *Гуманитарные аспекты*, к которым можно отнести учет того, что пользователь может плохо различать цвета, иметь физические проблемы с использованием клавиатуры, не знать иностранных языков, иметь культурные и национальные особенности и т. п.

Прикладное программирование решает задачи в различных прикладных областях, которые в свою очередь накладывают определенную специфику на то, как решаются эти задачи. Например, чрезвычайно востребованная область программирования – программирование различного рода вычислительных пакетов и программ, без которых невозможно представить современную индустрию и общество. Для вычислительного программирования к его характерным требованиям относятся точность вычислений, производительность, адекватность программной модели реальным процессам и явлениям. Если проблематика вычислительного программирования не очень знакома широкому кругу, то с решениями, разрабатываемыми в коммуникационном программировании, сталкивается практически каждый человек на планете. Для него характерна работа с физическими каналами связи, вопросы надежной и эффективной передачи информации с учетом требований безопасности, и т. д. Описание специфических областей прикладного программирования можно продолжать долго, однако мы ограничимся этими примерами, так как это не является предметом нашего пособия.

2. Системное программирование

Область программирования, целью которой является поддержка процесса создания или исполнения других программ, называется *системным программированием*. Таким образом, пользователями системных программ являются сами программисты, тогда как для конечных пользователей программ, которые мы назвали прикладными, системные программы используются косвенно. К системным программам можно отнести следующие:

- *Системы программирования*, осуществляющие перевод программ с языков программирования в машинные команды и сборку программ из модулей, поддерживающие процесс отладки, тестирования, документирования программ и т. п.
- *Операционные системы* (ОС), осуществляющие запуск и взаимодействие программ как между собой, так и с внешними устройствами.
- *Системы управления базами данных* (СУБД), предназначенные для хранения и быстрого доступа к большим объемам информации.

Разделение на прикладное и системное программирование достаточно условно. Так, например, определенные знания и навыки программирования требуются инженерам при использовании системы автоматизации проектирования (САПР) и математических пакетов, реализующих сложные вычислительные методы.

3. Технология программирования

Программирование не ограничивается собственно написанием программ. Большие программные системы создаются большими коллективами разработчиков, тестировщиков, менеджеров, между которыми надо обеспечить взаимодействие, организовать последовательность работ, гарантировать выполнение как внешних, так и внутренних требований и соглашений и т. п. В этом смысле программирование можно рассматривать как коллективный инженерный процесс создания программного обеспечения. Как и любой подобный процесс, он не может развиваться спонтанно и должен поддерживаться подходящими для конкретной ситуации инструментами и методами их применения. Этими вопросами занимается

исследовательская и инженерно-конструкторская дисциплина, которую часто называют *технологией программирования*.

Необходимо заметить, что понятие технологии очень расплывчато и для разных областей человеческой активности определяется по-разному (см. [56]). Даже в одной отрасли возможны разные трактовки этого слова. Так, технологией программирования часто называют совокупность методов и средств, используемых в процессе разработки программного обеспечения, но для программистов разной квалификации для разработки одной и той же программы могут потребоваться различные «методы и средства» [4, 5, 18, 25, 35, 36, 48, 49, 50, 54, 55]. Означает ли это наличие нескольких технологий производства одного и того же продукта? Также приходится говорить о стандартах разработки и о качестве, в частности, необходимо понимать, что имеется в виду, когда утверждается, что некая программа разрабатывалась «нетехнологично». Все это требует уточнения понятий, в рамках которого мы приходим к следующему «универсальному» определению (читателя не должно «пугать» очень длинное предложение – каждая строка его указывает на определенный аспект понятия технологии):

Технология – это среда поддержки выполнения определенной целенаправленной деятельности, обладающая:

- *средствами* и *инструментами*, а также *методами* их применения, неукоснительное следование которым каким бы то ни было
- *исполнителем* с *определенной квалификацией*,
- гарантированно обеспечит *производство*, т. е. получение из предоставляемых *ресурсов* и *материалов* требуемого *продукта-результата*, соответствующего
  - *целям*,
  - в требуемом *объеме,*
  - за известное *время* и
  - с приемлемым уровнем *качества*.

Технология последовательно и *детерминированно* ведет к решению задачи организуемой и выполняемой деятельности. Существует всего лишь три варианта производства продукции человеком: искусство, ремесло и технология. В этом перечне у настоящего искусства никакого детерминизма нет и не может быть, в ремесле происходит накопление полезного опыта для использования его

в дальнейшем, и это не что иное, как развитие ситуаций с детерминированными действиями, а технология – абсолютное подавление недетерминизма.

Вернемся к обсуждению того, что можно рассматривать в качестве технологии программирования.

Помимо собственно составления, т. е. производства программ (называемого также *кодированием*), технология должна поддерживать весь *жизненный цикл* программы, в частности:

- *спецификацию* создаваемого программного обеспечения, проверку соответствия требованиям заказчика;
- *проектирование*, т. е. разработку общей архитектуры системы, взаимодействие компонентов и т. п.;
- *отладку* и *тестирование* – нахождение и своевременное исправление ошибок;
- *документирование* в виде как инструкций для пользователя, так и технического описания самой программы для тех программистов, которые будут ее далее развивать или сопровождать;
- *сопровождение*, *версионность* – реакцию на замечания и рекламации пользователей, необходимость поддерживать несколько версий программы и т. п.

Даже этот перечень, довольно схематически раскрывающий деятельность по составлению программ, показывает, что технология любого (не побоимся этого слова!) производства складывается из других, вложенных технологий, которые определяют производства частей целевого продукта, возможно, средств и методов и других составляющих нашего определения без исключений [1]! Во многих случаях, а если быть совсем точным, то всегда технология не может обойтись без применения двух других вариантов производства продукции: без ремесла и искусства. Если продукты этих недетерминированных производств готовы к использованиюв сумми-

---

[1] Быть может, у читателя возникнет вопрос о том, как можно производить *исполнителя*, если это давно, успешно и без всяких технологий научились делать родители. Увы! Это заблуждение: *исполнителями* с *определенной квалификацией*, о которых шла речь в определении, не рождаются – они являются продуктом деятельности, производящей квалифицированного сотрудника. Может ли эта деятельность быть технологией или нет, вопрос спорный, а вот быть технологичной она должна быть безусловно.

рующей технологии, то они являются внешними по отношению к ней, и эта технология остается детерминированной. Противоположный случай указывает на то, что используется «недостроенная» технология.

Идеальная технология программирования не должна допускать появления ошибок в программе, опираясь на строгий формальный или формализованный *вывод* программы из спецификации. Реальное положение дел, однако, свидетельствует, что именно отладка программы занимает большую часть времени разработчиков.

Большая часть технологии может быть в той или иной степени автоматизирована и, в частности, поддержана соответствующими системными программами.

4. Теоретическое программирование

Сложность реальных программ для анализа и обработки приводит к необходимости построения формальных моделей и их математического исследования. Предметом исследования *теоретического программирования* являются как структурные, так и поведенческие свойства программ. Естественно, что теоретическое программирование, как математическая дисциплина, опирается на знания из смежных областей.

Дискретная математика и *кибернетика* используются для изучения структуры данных и алгоритмов. Синтаксические деревья, графы управления и потоков данных и многие другие модели программ требуют знаний из *теории графов*;

Теория вероятности и *математическая статистика* необходимы для анализа сложности вычислений и анализа вычислительных моделей, включающих случайность как существенную часть поведения;

Алгебра, *логика*, *теория алгоритмов* – при формальном описании семантики программ и верификации, т. е. проверки и доказательства соответствия программы спецификации;

Системный анализ необходим при проектировании программ, поскольку они представляют собой большое количество разнородных взаимодействующих компонент.

Понятно, что этим перечисление не ограничивается, поскольку при переходе, к примеру, к прикладному программированию, сюда привлекаются знания из конкретных областей – физики, социологии, экономики, биологии и т. д.

1. «ОКРУЖЕНИЕ» ПРОГРАММИРОВАНИЯ

1.1. Логическая модель ЭВМ

Для дальнейшего изложения нам потребуется общее представление о структуре ЭВМ. Мы сразу ограничимся так называемой *фон-неймановской архитектурой*, которая предполагает, что как программа, так и данные хранятся в памяти. Кроме того, мы (временно) исключим из рассмотрения работу с периферийными устройствами, различие между оперативной и внешней памятью и многое другое, что подробно обсуждается в курсе по архитектуре ЭВМ. Нам будет достаточно схемы, представленной на рис. 1.1.

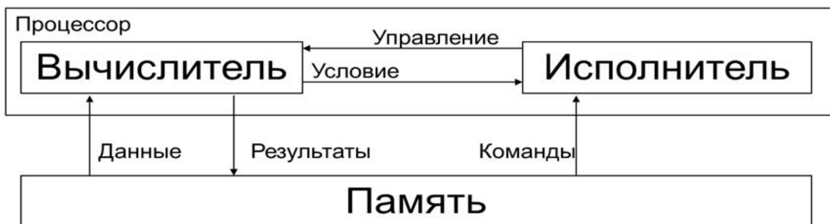


Рис. 1. 1. Структура ЭВМ

Процессор состоит из исполнителя и вычислителя. *Исполнитель* выбирает из памяти очередную команду и при необходимости указывает вычислителю, какие операции и над какими данными надо выполнить. *Вычислитель* извлекает из памяти требуемые данные, выполняет операцию, помещает результат обратно в память и/или сообщает вычислителю о том, какую команду выбрать следующей. Таким образом, команды процессора можно разделить на:

- арифметические и битовые, выполняемые вычислителем;
- управляющие, выполняемые исполнителем;
- присваивания, пересылки;
- ввод/вывод.

Далее мы будем предполагать, что память дискретна, т. е. является организованным набором *битов*. Бит – элементарная единица представления информации, имеющая два возможных значения: 0 и 1. Биты объединяются в *байты* – минимальные адресуемые группы из 8 бит. Из четырех байтов формируется слово, с которым может

оперировать машинная команда. Схематично это отображено на рис. 1.2.

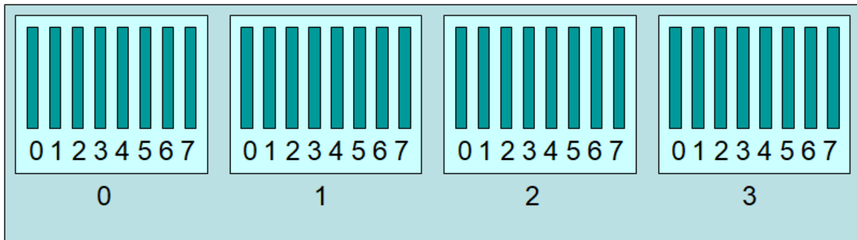


Рис. 1.2. Дискретная память

Не следует воспринимать эти сведения как догму. Так, можно поинтересоваться следующими вопросами:

- Почему бит содержит только два значения, а не три или, скажем, 10? Ведь, например, при математическом моделировании или финансовых расчетах мы изначальные данные вводим и выводим в десятичном виде;
- Почему в байте 8 бит? Ведь, например, для кодирования всех символов, доступных на клавиатуре, достаточно 6 бит;
- Почему в слове 4 байта, а не 2 или 6?
- Как нумеруются биты внутри байта и внутри слова?

Конечно, можно ответить: «Потому, что это так в моем компьютере», но вряд ли такой ответ можно признать удовлетворительным. И действительно, существуют такие компьютеры, в которых в байте 9 бит, и такие, в которых в слове 8 байт. И вполне возможно, что нам придется писать программу, которая это учитывает.

Мы оставляем также в стороне более сложные вопросы, связанные с организацией памяти – сегментирование, виртуальную память, тегирование памяти и т. п. Сказанного выше достаточно для дальнейшего изложения.

1.2. Операционная система

Для того чтобы программа попала в память и начала выполняться, необходимо выполнить определенные действия. Изначально эти действия выполнял *оператор* ЭВМ – человек, который заполнял с помощью тумблеров нужную область памяти командами, соответствующими программе, и ее данными и нажимал по завершению

этого процесса кнопку «Пуск», передавая программу на выполнение. Достаточно быстро этот рутинный процесс переложили на ЭВМ. Программа, обеспечивающая запуск других программ и их взаимодействие, получила название *операционной системы* (ОС). Функции операционных систем постепенно усложнялись, и условно их можно разделить на внутренние и внешние.

К внутренним функциям относятся, например, следующие:

- управление ресурсами, основными из которых являются потребляемые программой процессорное время и оперативная память. Эта функция особенно важна, если ОС допускает одновременное выполнение нескольких программ, возможно, с разными приоритетами. Помимо времени и памяти, к ресурсам можно относить и доступ к внешним устройствам, таким как принтер, сеть и т. п.;
- реакция на сигналы от разнообразных периферийных устройств, которая обеспечивается специальными подпрограммами, называемыми *драйверами*. В задачу операционной системы входит передача соответствующего сигнала нужной программе. Типичным примером может служить процесс обработки нажатия кнопки на клавиатуре или на мышке от замыкания контактов до выполняемых программой команд:
- обработка аварийных ситуаций – «нормальное» завершение программы в случае, если программа попыталась выполнить запрещенную операцию (деление на ноль, обращение по несуществующему адресу и т. п.) или была насильно остановлена пользователем или самой операционной системой.

Внешние функции в большей степени видны пользователю:

- создание процессов и их взаимодействие;
- файловая система обеспечивает хранение данных на внешних носителях и имеет в большинстве случаев иерархическую систему каталогов, папок, директорий или т. п. С точки зрения программы, операционная система в значительной степени экранирует особенности конкретного физического носителя, будь то жесткий или компактный диск, флэш-память и т. п.;
- безопасность – широкий круг вопросов, важнейшими из которых являются конфиденциальность данных и безотказная работа компьютера;

- интерфейс – отрисовка окон, рассылка сообщений о действиях пользователя и т. п.;
- полномочия пользователей – не все пользователи имеют равные права в системе;
- статистика – сбор информации о функционировании системы, такой как загруженность, использование периферийных устройств, попытки проникновения извне с целью нанести вред.

Как уже было упомянуто, что как набор функций, так и само понятие операционной системы существенно меняется по мере развития информационных технологий. Да и сама операционная система построена зачастую как матрёшка, в самом центре которой находится ядро ОС, реализующее самый базовый уровень, т. е. управление драйверами, запуск и синхронизацию процессов и т. п., а каждая следующая оболочка строится с использованием тех функций, которые предоставляет ее «содержимое».

2. ПОКОЛЕНИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

2.1. Машинный язык

Как уже говорилось, процессор выбирает команды для исполнения из памяти. Таким образом, каждая программа является данными, представленными последовательностью машинных слов (байтов, битов). Естественно, что не любая последовательность слов является программой, а только та, которая принадлежит *машинному языку*. Обычно каждая команда в машинном языке представляется одним словом, которое условно разбивается на операцию и аргумент(ы) – указание данных, над которыми выполняется операция. Разнообразие команд и способы адресации данных существенно различаются в зависимости от типа процессора.

2.2. Мнемокод

Понятно, что запись программ в двоичном коде воспринимается сложно. Еще сложнее вводить в такую программу изменения, поскольку, скажем, вставка дополнительных команд изменяет нумерацию остальных команд и данных, и требует изменения других команд, где эти номера используются. Решение этих проблем дает *мнемокод*, который предоставляет две основные возможности:

- вместо двоичной записи операций использовать их обозначения, например, ADD для операции сложения, MOV – для операции пересылки и т. п. [2];
- вместо номеров команд и ячеек данных использовать их мнемоничные имена.

---

[2] Здесь и далее в этой главе нас интересуют в первую очередь содержательные понятия, а не способ их оформления в конкретном языке. Поэтому читателю можно не вникать в подробности способа записи, пунктуации, обозначений и т. п.

Пример небольшой программы, использующей мнемокод:

```
.MODEL SMALL
             .DATA
b          DW     5
c          DW     3
a          DW     ?
           .CODE
begin   MOV   AX,@DATA
          MOV     DS,AX
          MOV     AX,B
          ADD     AX,C
          MOV     A,AX
          MOV     AH,4CH
          INT       21H
          END     begin
```

Программа в такой записи легко автоматически преобразуется в программу на машинном языке: достаточно непосредственно перед преобразованием установить соответствие имен и номеров ячеек. Использование абстракции, т. е. имен вместо конкретных значений, является первым, но очень важным шагом на пути повышения уровня языка программирования.

Таким образом, главным достоинством мнемокода по сравнению с машинным языком является лучшая понимаемость программ.

2.3. Макроассемблер

Поскольку машинные команды (обычно) выполняют только самые элементарные действия, то содержательные действия чаще всего реализуются некоторыми последовательностями команд. Одна и та же последовательность, возможно с небольшими изменениями, может многократно использоваться в программе. Даже просто с точки зрения экономии усилий имеет смысл как-то назвать эту последовательность, отметить изменяемые части, чтобы в дальнейшем не выписывать ее снова и снова, а там, где она требуется, указать название последовательности и то, что надо подставить вместо заменяемых частей.

Такие именованные последовательности называются *макроопределениями* или кратко *макросами*. Сама последовательность называется *телом макроса*, а изменяемые части макроса, называемые *формальными параметрами макроса*, либо нумеруются, либо

именуются, чтобы различать их между собой. Обычно именование предпочтительнее нумерации с точки зрения понимаемости. Тем не менее, нумерация иногда оказывается весьма выразительной. Например, она может использоваться для организации циклически повторяемых действий для всех параметров. Один и тот же формальный параметр может многократно использоваться в теле макроса – вместо всех использований будет подставляться один и тот же текст. Макроопределение может выглядеть, например, так:

```
MI MACRO C1,C2,CP,MP
MOV ax,C1
MUL C2
MOV CP,dx
MOV MP,ax
ENDM
```

Здесь `MI` – имя макроса, `C1`, `C2`, `CP`, `MP` – имена параметров, а тело макроса состоит из четырех команд: вплоть до команды ENDM, означающей конец макроса.

Фрагмент программы, который задает использование макроопределения, называется *вызовом макроса*, а те фрагменты, которые следует подставить вместо вхождений формальных параметров, – фактическими *параметрами.* Например, два вызова макроса `MI` могут выглядеть как

```
MI DI,A,S1,S2
MI S,2,DI,SI
```

С точки зрения вычислений это то же самое, что и последовательность из восьми команд:

```
MOV ax, DI
MUL A
MOV S1,dx
MOV S2,ax
MOV ax, S
MUL 2
MOV DI,dx
MOV SI,ax
```

поскольку в первом вызове вместо формального параметра `C1` подставляется текст `DI`, вместо `C2` – `A`, вместо `CP` – `S1`, вместо `MP` – `S2`,

а во втором – вместо `C1` – `S`, вместо `C2` – `2`, вместо `CP` – `DI`, вместо `MP` – `SI`.

Налицо существенное сокращение записи. Отметим, что макросредства задают только преобразование текста программы: выполнение всех подстановок вызовов макросов приводит к получению текста программы на мнемокоде, которая уже напрямую транслируется в машинную программу. Более того, работа *макропроцессора* – программы, осуществляющей преобразование программы с макроопределениями, – не зависит от того, что именно записано в теле функции – мнемокод или любой другой текст. Для нее существенно только то, как оформляются макроопределения и вызовы макросов.

Отметим еще несколько непосредственных следствий введения макросов:

- Мы можем изменять тело макроса, не меняя все его вызовы до тех пор, пока имя макроса и набор формальных параметров не изменились. Например, если макрос реализует вычисление корней квадратного уравнения по заданным коэффициентам, то мы можем улучшать эту реализацию, изменяя только тело макроса, и при этом это улучшение будет «автоматически» происходить во всех вызовах.
- В теле макроса могут быть использованы вызовы других макросов. Таким образом, если считать, что макросы определяют новые команды, то у нас появляется возможность неограниченного расширения языка и повышения уровня абстракции команд, из которых строится программа.
- Макросы могут порождать очень большие тексты. Вообще говоря, размер результирующей программы может возрастать экспоненциально в зависимости от количества вызовов макросов и/или вхождений формальных параметров в тело макроса.
- Одни и те же макросы можно использовать при написании разных программ. Это открывает возможность создания библиотек макросов: не обязательно каждый раз выписывать все используемые макроопределения непосредственно в тексте программы. Можно собрать некоторый содержательный набор макросов в отдельном файле-библиотеке, а макропроцессору указать (например, специальной директивой в тексте программы) о необходимости исполь-

зования этой библиотеки. Далее, эта возможность способствует, например, совместной разработке большой программной системы коллективом разработчиков.

2.4. Алгоритмические языки высокого уровня

Хотя макроассемблер и предоставляет средства расширения языка путем определения новых «команд», он остается «привязанным» к конкретному машинному языку. Следующий шаг в повышении уровня языка был сделан с появлением так называемых *алгоритмических языков высокого уровня* (АЯВУ), ориентированных в первую очередь на формулировку алгоритма, нежели на перевод программ, записанных на этих языках, в машинный язык. В этих языках появляются средства конструирования для типовых структур данных (массивы, векторы, матрицы, кортежи, записи и т. п.) и шаблонов управления (формулы, циклы, ветвления, определяемые функции и процедуры и т. п.).

Большая часть программ на АЯВУ оказывается машинно-независимой. Действительно, если, например, алгоритм оперирует с целыми числами при помощи обычных арифметических операций, то до определенного предела неважно, как именно представляются целые числа машинными словами и какие именно машинные команды (или вспомогательные подпрограммы) реализуют операции умножения и сложения. Конечно, в языке программирования высокого уровня могут быть конструкции низкого уровня, такие как побитовые операции, но их использование ограничивается либо необходимостью доступа к специфическим машинным данным в системных программах, либо повышенными требованиями к эффективности программы.

В настоящее время насчитываются тысячи языков программирования. Большинство языков являются универсальными в том смысле, что с их помощью можно записать некоторый алгоритм для вычисления любой интуитивно вычислимой функции [6, 39], и в этом смысле все они эквивалентны. Однако некоторый язык может оказаться удобнее, компактнее, выразительнее, чем другие, для решения конкретного класса задач. Существуют и специализированные языки, которые разрабатывались, например, для специализированных процессоров, без какой-либо претензии на универсальность.

Следует, однако, признать, что ориентация языка на класс задач может носить в значительной степени субъективный характер. Так, например, язык Фортран [43] традиционно считается ориентированным на научные расчеты. Но его вполне можно использовать и для экономических задач, и для машинной графики, и для создания трансляторов. С другой стороны, очевидно, существует целый ряд ЯВУ, которые не только нисколько не хуже Фортрана для научных расчетов, но и за счет других языковых аспектов имеют определенные преимущества. Кроме того, сам по себе уровень языка (неформально – его синтаксическая и, что более важно, семантическая удаленность от машинного кода) и его выразительные возможности не предохраняют от написания на нем плохих программ. По выражению Э. Дейкстры, «фортрановскую программу можно написать на любом языке программирования».

Алгоритмические языки высокого уровня можно разделить на классы в зависимости от базовой системы понятий, на которой они основаны. Ниже мы кратко рассмотрим наиболее часто встречающиеся в соответствующих дискурсах парадигмы: императивные и функциональные языки программирования. Помимо них существуют и другие, например, логические языки программирования (Prolog), языки, основанные на нормальных алгоритмах Маркова (Refal) и т. д. Вообще говоря, это разделение весьма условно, поскольку в реально используемых современных языках программирования сосуществуют конструкции, относящиеся к разным парадигмам.

2.4.1. Императивные языки

Для императивных языков программирования процесс выполнения программы определяется как пошаговое исполнение инструкций, меняющих состояние памяти. Таким образом, в программе на императивном языке присутствуют три составляющие:

- описание структуры хранящихся в памяти данных;
- базовые операторы, изменяющие состояние памяти. Наиболее часто используемым и присутствующим во всех императивных языках является, по-видимому, оператор присваивания;
- организация последовательности исполнения базовых операторов.

С такой точки зрения машинные языки также можно отнести к императивным. Императивные АЯВУ начали появляться в конце 50-х годов XX века – Алгол-60, Фортран, Кобол. Потом создание новых языков приняло лавинообразный характер – Simula-67, Паскаль, Modula-2, C, C++, Java, C# и десятки других. Были попытки создания универсального, подходящего для всех прикладных областей, языка программирования – Алгол-68, PL/I, Ada, но успешными их назвать нельзя [20, 41, 57]. Одна из главных причин – фиксация на уровне языка удобных (по мнению разработчиков языка) и надежных средств решения любых (опять же по мнению разработчиков языка) задач практически невозможна. Новые языки программирования появляются до сих пор.

2.4.2. Функциональные языки

Характерное для императивных языков программирования пошаговое исполнение инструкций не является неотъемлемым свойством понятия алгоритма. Примером того, что программу вычислений можно сформулировать точно, но при этом оставляя большую свободу в выборе последовательности действий и возможности для параллельного вычисления подзадач, являются функциональные языки программирования: Lisp, Scheme, Miranda, ML, Haskell, Scala и т. д. Ввиду того что в данном случае мы декларируем наш «макровзгляд» на решение проблемы, описывая связь между входом и выходом, такие языки еще называют декларативными.

Для этого класса языков характерным является конструирование программ в виде совокупности функций. Язык обычно предоставляет некоторый набор базовых функций, реализующих, например, арифметические операции. Программист для определения новых функций может использовать как базовые функции, так и любые ранее определенные. Характерными для функциональных программ является отсутствие (или очень ограниченное использование) изменения состояния памяти, а также *рекурсия*, т. е. явное или скрытое использование функцией самой себя для

решения подзадачи меньшего (в некотором смысле) размера. Например, определение функции вычисления факториала

$$fact(n) = \begin{cases} n * fact(n-1), & \text{при } n > 0 \\ 1, & \text{иначе} \end{cases}$$

на языке Scheme записывается как

```
(define (fact n)
   (if (> n 0)
         (* n (fact (- n 1)))
         1))
```

Запись программ в функциональных языках программирования может показаться на первый взгляд непривычной, но во многих случаях она гораздо ближе к формулировке задачи.

Естественный параллелизм в функциональных программах заключается в том, что если у функции есть несколько аргументов, то их можно вычислять в произвольном порядке, либо при наличии вычислительных возможностей – одновременно.

3. РЕАЛИЗАЦИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Алгоритмические языки высокого уровня в отличие, скажем, от языка макроассемблера, в значительной степени потеряли непосредственную связь с машинным языком. Это дистанцирование, т. е. стремление сделать язык машинно-независимым, приблизить его к предметной области и алгоритмам, потребовало создания инструментов, позволяющих выполнять программы на ЭВМ. Иными словами, мы должны выразить смысл конструкций АЯВУ в терминах машинного языка. Два основных способа сделать это реализуются соответственно *интерпретаторами* и *трансляторами*.

3.1. Интерпретаторы

Интерпретатор можно сравнить с переводчиком-синхронистом, который воспринимает очередную, обычно достаточно небольшую фразу и сразу ее произносит на другом языке. Он, конечно, может знать о специфическом для переводимого текста (выступления) лексиконе и характерных речевых оборотах, но не может видеть весь текст целиком. Если же уровень языка, с которого осуществляется перевод, значительно выше того, на котором говорит переводчик, то для краткой фразы может потребоваться многословная трактовка, которую переводчик вынужден повторять всякий раз, когда эта фраза повторяется в тексте. Так или иначе, если считать, что смысл «исполнения» текста состоит в донесении его до слушателя, то интерпретатор выполняет эту работу.

Формально говоря, каждый язык программирования L сопоставляет тексту программы p некоторый смысл, например, функцию $L[p]$, отображающую входные данные в выходные. *Интерпретатором* для языка L в языке I, называется программа *int*, записанная в языке I, удовлетворяющая следующему свойству:

$$I[int](p,d) = L[p](d).$$

Иными словами, интерпретатор на вход получает программу p в языке L и ее данные d и делает то же, что и программа p над данными d. Отметим, что в данных рассуждениях одна и та же программа p присутствует как пассивно, т. е. как данное, которое можно, в частности, передать на вход интерпретатору, так и

активно – как функция, которая приписывается ей языком L и представляется в языке I.

3.2. Трансляторы

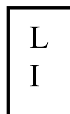
В отличие от интерпретатора транслятор можно сравнить с литературным переводчиком. Он получает текст либо сразу целиком, либо значительную его часть, имеет возможность прочитать и проанализировать текст многократно, сделать подстрочный перевод и лишь в конце составить окончательный текст. Именно то, что этот процесс в значительной степени заключается в «составлении» текста, объясняет другое название – компилятор. При этом родной язык переводчика может отличаться как от входного, так и выходного языка. Таким образом, формально *транслятор* (*компилятор*) с языка L_1 в язык L_2 – это программа *comp* на языке I, удовлетворяющая следующему свойству: если p – программа на языке L_1, то $I[comp](p)$ – есть программа *obj* на языке L_2, такая что для любых данных d:

$$L_2[obj](p) = L_1[p](d).$$

Основное отличие от интерпретатора заключается в том, что компилятор не выполняет входную программу, а только переводит ее на другой язык.

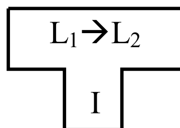
3.3. Т-диаграммы

Графически мы будем изображать интерпретатор прямоугольником, в верхней части которого указан реализуемый язык, а в нижней – язык, на котором интерпретатор написан:



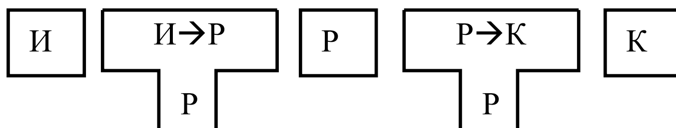
Этот кирпичик можно рассматривать как элементарное устройство, на которое сверху можно положить любую программу на языке L, и, если это устройство удастся заставить работать, то заработает и положенная программа. Заставить же работать сам интерпретатор можно, например, положив этот кирпичик (как программу на языке I) на интерпретатор языка I и т. д.

Для графического изображения транслятора используется Т-блок [3], в основании которого указывается язык реализации, слева – входной язык транслятора, справа – выходной:



Входные данные, т. е. программы на языке L_1 подаются слева, а результат указывается справа.

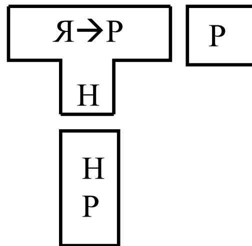
Из таких кирпичиков можно строить разнообразные схемы, соблюдая при этом правила стыковки. *Многофазная трансляция* может быть полезна для разбиения сложного процесса на последовательность более *простых* или *доступных* переводов. Она означает, что перевод с исходного на конечный язык осуществляется не напрямую, а в несколько этапов с использованием промежуточных языков. Продолжая аналогию с переводчиками, предположим, что у нас не оказалось знакомого переводчика с итальянского на китайский, но зато есть два знакомых переводчика: с итальянского на русский и с русского на китайский, тогда схема перевода может выглядеть как



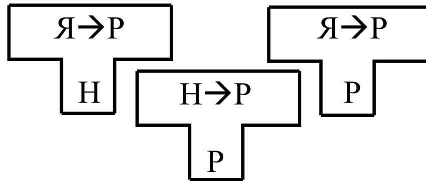
где блоки, помеченные буквами И, Р и К означают тексты на итальянском, русском и китайском языке соответственно. Заметим, что здесь мы предполагали, что оба переводчика «думают» по-русски. Правильнее было бы полагать, что у нас есть не переводчики, а инструкции по переводу на русском языке, которые мы, как носители языка, можем «исполнять» без посторонней помощи. Если же одна из инструкций оказалась написана на другом

[3] Понятие Т-диаграмм возникло около 1960 года в работах американских разработчиков-компиляторов.

языке, скажем на немецком (Н), то для того чтобы ее выполнить, нам потребовалась бы помощь интерпретатора,



либо следовало предварительно перевести инструкцию с немецкого на русский:

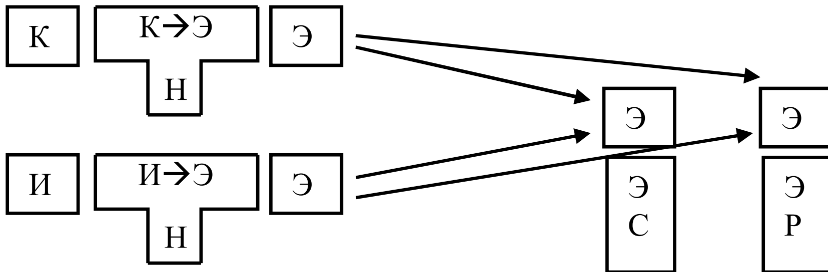


и затем использовать ее по старой схеме.

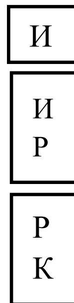
Типичным примером многофазной трансляции является трансляция с языка высокого уровня в язык ассемблера, который затем транслируется в машинные команды «штатным» транслятором. Пошаговая трансляция может состоять из более чем двух шагов. Так, например, одна из возможных реализаций языка C++ состоит в том, чтобы сначала перевести его в язык C, затем с языка C – в язык ассемблера, и, наконец, с языка ассемблера в машинный язык.

Если же имеется множество входных языков, то «взаимопонимания» можно добиться трансляцией их всех в один, возможно, специально для этого разработанный общий язык. Этот язык, в отличие от языка ассемблера, может быть достаточно высокого уровня, что обеспечит возможность его реализации на различных устройствах либо путем трансляции в машинные команды, либо интерпретацией. Если в качестве примера такого языка взять эсперанто (Э), то для того чтобы на русском (Р) и на санскрите (С) «понимать» китайский (К) и итальянский (И), достаточно перевести китайский и итальянский в эсперанто и научиться «понимать» только один язык. Причем совершенно неважно, на каком языке осуществляется

перевод, например, на немецком (Н). Одним из примеров такого общего языка в практике программирования выступает *байт-код*, реализуемый на разнообразных встроенных или мобильных устройствах интерпретаторами (*виртуальными машинами*).



Промежуточные языки могут использоваться не только для трансляции, но и для *многоуровневой интерпретации*. Вернемся к примеру перевода с итальянского на китайский. Для того чтобы обеспечить синхронный перевод, мы можем привлечь двух переводчиков: первый будет слушать итальянскую речь и тут же повторять ее на русском, а второй будет слушать то, что говорит первый переводчик, и передавать ее китайскому слушателю:

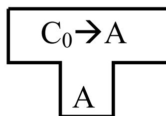


Ясно, что качество такого перевода вызывает сомнения, поскольку первому переводчику для точного перевода краткой и емкой фразы на итальянском может потребоваться пространный текст на русском, второй же переводчик, не зная, что сказал итальянец, будет еще более пространно излагать русскую речь на китайском. Кроме того, кратно увеличивается время перевода.

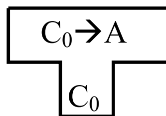
Каким же образом транслятор с некоторого языка появляется на машине? Представим себе ситуацию, когда нам принесли новую

машину, на которой ничего нет, кроме операционной системы и (чтобы немного смягчить ситуацию) языка ассемблера, а нам требуется создать работающий на этой машине транслятор с языка C. Конечно, можно просто засучить рукава и начать писать требуемый транслятор на ассемблере, но ввиду того, что язык C далеко не самый простой, а язык ассемблера далеко не самый удобный и надежный, эта работа потребует чрезмерно больших усилий и времени, если вообще закончится успешно.

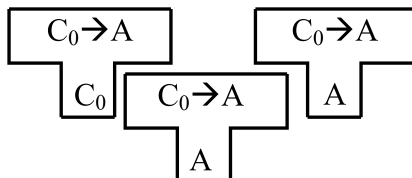
Один из классических подходов состоит в использовании *метода раскрутки*, при котором мы параллельно усложняем решаемую задачу и используемый для этого инструментарий. Начинается с того, что мы выбираем некоторое совсем небольшое, но универсальное подмножество языка C_0, в котором есть только простые выражения, присваивания, указатели, безусловные и условные переходы по метке и процедуры без параметров. На случай, если этих средств оказывается недостаточно, можно добавить в этот язык возможность вставки фрагментов на ассемблере. Теперь задача становится более обозримой, и мы реализуем на ассемблере транслятор с языка C_0 в язык ассемблера A:



После этого мы тут же переписываем этот транслятор на языке C_0:

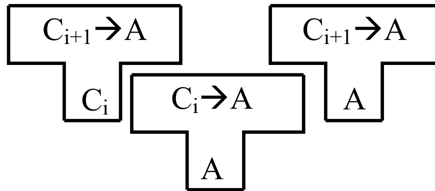


Поскольку второй транслятор является программой на языке C_0, то мы можем применить к нему первый транслятор:



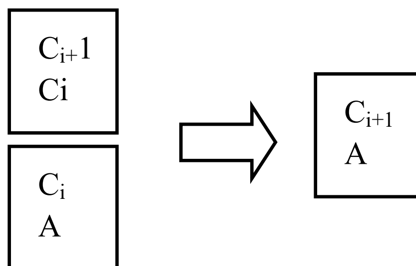
Заметим, что если предположить, что оба транслятора эквивалентны, но написаны на разных языках, то в результате такого применения получится еще один эквивалентный транслятор, также написанный на языке ассемблера.

Далее мы шаг за шагом расширяем язык, добавляя в него новые конструкции. Например, на следующем шаге мы добавляем сложные выражения с возможностью использования скобок и учетом приоритета операций, получая язык C_1. Затем добавляем структурные управляющие операторы `if`, `while`, `switch` – язык C_2, затем – сложные структуры данных – язык C_3, и т. д. И каждый раз мы реализуем язык C_{i+1} на языке C_i, поскольку, имея исполняемый транслятор с C_i в ассемблер, мы можем получить и исполняемый транслятор с C_{i+1} в ассемблер:



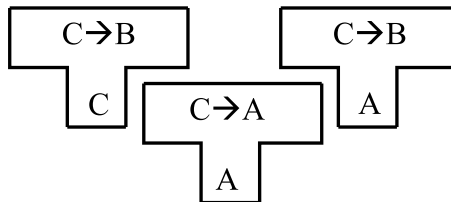
В конце этого процесса, когда будут включены уже все конструкции языка C, осталось выкинуть все лишнее, что мы вставили в C_0.

В принципе можно представить себе и использование раскрутки при реализации интерпретатора, когда каждый следующий в цепочки языков обрабатывается интерпретатором, реализованном в предыдущем. Однако для того чтобы получить конечный интерпретатор, необходимо иметь нетривиальную возможность свернуть пару интерпретаторов в один:

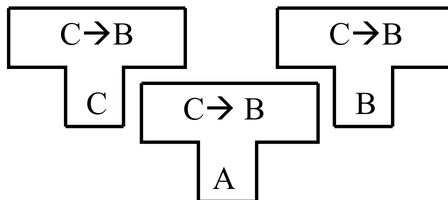


Один из возможных подходов к реализации этого преобразования дают *смешанные вычисления*, которые выходят за рамки данного курса [6].

Предположим теперь, что мы уже реализовали язык C для одной машины (A) и нам необходимо реализовать его для другой (B). Совсем не обязательно повторять весь процесс раскрутки на новой машине – можно использовать так называемую *кросс-компиляцию*. Всю работу по созданию нового транслятора мы выполняем на старой машине,



используя новую лишь для проверки того, что разрабатываемый транслятор порождает правильный код для машины B. Когда же разработка закончена, мы применяем его к самому себе, получая транслятор, исполняемый на машине B и генерирующий код для этой же машины:



Завершая обсуждение схем реализации языков программирования, отметим, что на практике они вряд ли встречаются в чистом виде. Так, например, интерпретатор, прежде чем приступить собственно к выполнению программы, может преобразовать (т. е. транслировать) ее в более удобное и необязательно текстовое представление. С другой стороны, транслятор в ходе своей работы может выполнить часть обрабатываемой программы, если для этого имеются все необходимые данные.

4. СИСТЕМЫ ПРОГРАММИРОВАНИЯ

Исполнение программы, о котором мы говорили в предыдущем разделе, т. е. переход от текста программы на исходном языке программирования к выполнению соответствующих ей машинных команд, имеет несколько более сложный характер. Это связано с тем, что программа чаще всего собирается из составных частей. Разбиение программы на части позволяет, во-первых, участвовать в ее создании нескольким разработчикам и, во-вторых, использовать стандартные составные части.

Сборка (комплексация) программ может проходить на нескольких уровнях. Самый простой способ – собирать программу на уровне исходного текста. Именно так используются библиотеки макросов в макроассемблере. Для текстовой сборки язык должен поддерживать макрообработку и предоставлять возможность «вставить» в указанное место текст из другого файла. В языке C это реализуется *директивой*

```
#include "имя файла"
```

а *включаемые файлы* обычно имеют расширение `.h`. Включаемые файлы чаще всего содержат описание используемых структур данных и функций. Формирование подлежащих компиляции файлов из текстовых заготовок выполняет *препроцессор*. Получаемые на выходе файлы уже не содержат директив препроцессора и имеют расширение `.i`.

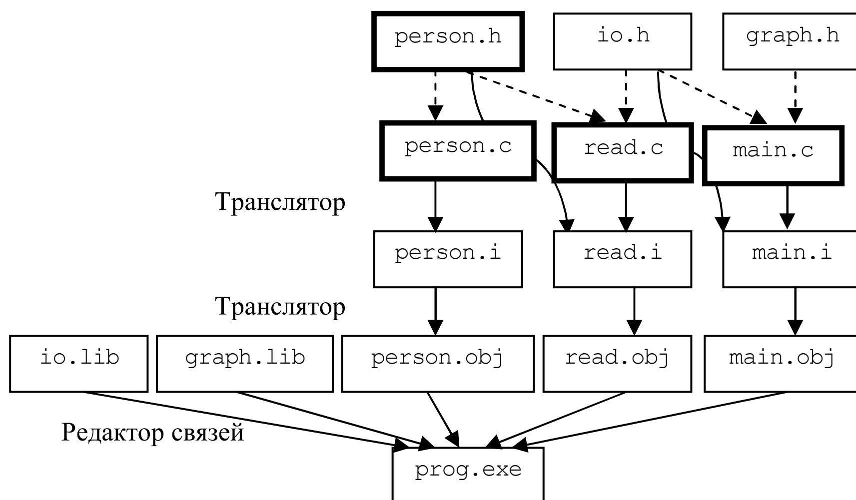
Далее полученные `.i` файлы поступают на вход собственно транслятору, который для каждого из них получает объектные модули. *Объектный модуль* – это фрагмент программы на машинном языке, в котором могут использоваться ссылки на элементы (например, функции), определенные в других модулях. Кроме того, для того чтобы другие модули могли ссылаться на данный модуль, он должен содержать описание тех его элементов, на которые можно сослаться.

Далее осталось собрать все объектные модули воедино, подключая при необходимости библиотеки стандартных объектных модулей, обычно имеющих расширение `.lib`. При этом возможно, что части объектных модулей перегруппируются, например, так, чтобы данные из всех модулей собрать в одном месте, а команды –

в другом (это бывает необходимо при сегментной организации программ, когда разные по смыслу части программы группируются по сегментам: сегмент текстовых констант, сегмент глобальных переменных, сегмент исполняемого кода, сегмент стека и т. д.). Таким образом, на этом этапе необходимо определить для каждого фрагмента объектного модуля его место в готовой программе и заменить ссылки на реальные машинные адреса. Всю эту работу выполняет *редактор связей*, или *ассемблер*.

Наконец, готовую программу необходимо поместить в память машины и передать управление на ее начало, что делается *загрузчиком*, который является частью операционной системы, поскольку не зависит от входного языка и того, как машинная программа была получена.

Изложенную выше схему исполнения программы демонстрирует следующий рисунок, на котором жирным выделена та часть, которую разрабатывал сам программист, непрерывные стрелки описывают зависимости, которые используются для порождения новых файлов (необязательно эти файлы порождаются физически в файловой системе), а пунктирные стрелки указывают на наличие директив препроцессора:



Количество фаз трансляции больше двух, например, когда файлы на языке C получаются из программ более высокого уровня, и система программирования может предоставлять свой набор стандартных

«заготовок». Программы могут получаться не обязательно из других программ. Например, некоторые системы программирования обращаются к базе данных, с которой будет работать конечная программа. Из базы данных извлекается ее схема, т. е. информация о структуре таблиц и связях между ними, на основе которой порождаются описания структур и функций на языке C, реализующие доступ к данным, которые используются специальным препроцессором.

Как видно, порождение исполняемой программы должно учитывать множество взаимосвязей: что из чего и в какой последовательности получается. Кроме этого, можно заметить, что раздельная трансляция дает возможность при изменении отдельных исходных файлов не повторять весь процесс построения полностью. Например, если изменения коснулись только файла `read.c`, то достаточно транслировать его в `read.obj` и после этого собрать готовую программу из старых объектных модулей. Если же был затронут включаемый файл `io.h`, то заново транслировать нужно `read.c` и `main.c`, но не `person.c`.

Таким образом, построение становится достаточно сложным, чтобы потребовать от системы программирования поддержки этого процесса. *Система построения* (build, make) может извлекать зависимости либо автоматически, например, путем просмотра исходного текста на языке C и обнаружения директив препроцессора, либо доверить пользователю описать все зависимости и порядок построения в специальном файле, либо использовать какой-нибудь комбинированный подход, когда пользователь описывает лишь нестандартные зависимости и процессоры.

Заметим, что даже в приведенном выше примере были задействованы несколько языков программирования, хотя некоторые из них, например, язык ассемблера, носят промежуточный характер и скрыты от программиста. На практике оказывается полезным использование в одной системе нескольких входных языков в зависимости от типа решаемой задачи или вида деятельности: язык высокого уровня для записи алгоритма, низкоуровневый язык для компонентов, где критически важна эффективность или доступ к аппаратуре, отдельный язык для доступа к базе данных, отдельный – для запуска процессов и взаимодействия с операционной системой и т. п. Таким образом, система программирования должна обеспечивать *многоязыковую среду разработки*.

В принципе этим функции системы программирования можно и ограничить, поскольку она уже предоставляет необходимые инструменты для исполнения программы. Однако деятельность, связанная с программами, далеко не ограничивается их исполнением. Оставив за рамками рассмотрения вопросы *анализа предметной области*, *требований заказчика* и *проектирования* (которые в идеальном случае тоже должны быть поддержаны соответствующими инструментами), будем считать, что программа исходно представляется своим текстом, который может быть создан любым текстовым редактором. Однако если редактор используется именно для работы с программами на определенном входном языке, то естественно ожидать от него некоторой помощи в работе с программой. Например, автоматического форматирования, лексической раскраски текста (выделение служебных слов), синтаксических (идентификация парных скобок) и семантических (что может появиться в данном месте программы) подсказок и т. п. Кроме того, пользователь заглядывает в текст программы не только в момент ее написания, но и, например, когда транслятор обнаружил ошибку либо когда программа была приостановлена в процессе исполнения. При этом хотелось бы, чтобы редактор сам указал нам нужную точку в программе. Естественно, что для этого требуется более тесная интеграция редактора с другими компонентами системы программирования: он должен «понимать» сообщения транслятора, а для объектных модулей и готовой программы должна поддерживаться их связь с исходным кодом, причем пронизывающая весь путь преобразования программы. Таким образом, система программирования чаще всего включает в себя собственный *языково-ориентированный текстовый редактор*, а мы можем говорить об *интегрированной среде разработки*, для которой редактор является основным интерфейсом, через который осуществляется не только изменение текста программы, но и многие другие действия, возникающие при разработке программ.

Знание редактора о том, что он обрабатывает именно текст программы на некотором языке программирования, позволяет существенно расширить его возможности. Например, он может включать в себя *справочную систему*, которая может выдать информацию об указанной языковой конструкции, стандартной функции или типе данных. Если же система поддерживает *документирование* программ, то справочная система может предоставить информацию не

только о предопределенных конструкциях и объектах, но и об определенных программистом, если для них были обеспечены соответствующие аннотации, оформляемые чаще всего как комментарии специального вида. Система документирования может извлечь из текста программы все эти аннотации и собрать их в отдельный документ или совокупность взаимосвязанных документов. Документация программы должна освещать по крайней мере следующие аспекты [14]:

- *техническая документация* предназначена для программистов и описывает собственно структуру и внутренние взаимосвязи программы, такие как предназначение переменных, функций, параметров и т. п. или ограничения на использование тех или иных объектов;
- *системная документация* предназначена для администраторов и описывает, где и как должна быть установлена программа, какие ресурсы она использует и т. п.;
- *пользовательская документация* описывает, с какими параметрами можно запустить программу, реализуемые ею сценарии, пользовательские интерфейсы и т. п.

Включенность в систему программирования позволяет редактору осуществлять и более сложные редактирующие действия. Например, помимо простого текстового поиска, редактор может использовать информацию, полученную от транслятора и загрузчика, для отображения *перекрестных ссылок*: перехода от использующего вхождения переменной или функции к соответствующему определению и наоборот – поиска всех использований объекта, причем не только в редактируемом файле, но и во всей системе. Редактирующие действия могут быть расширены на языковые конструкции. Простейшим примером такого действия является переименование объекта, что, естественно, должно приводить к поиску и переименованию всех использующих вхождений. Можно реализовать и более сложные трансформации: открытую подстановку функций/макросов (т. е. подстановку определения вместо использования с соответствующей заменой параметров), изменение формы циклов, и т. п. Такого рода операции служат для так называемого *рефакторинга* программ, т. е. изменения их текста или структуры без изменения реализуемой функциональности с целью улучшения понимаемости, простоты дальнейшего сопровождения или развития и т. п.

Естественным расширением понятия исполнения является *отладка* программы. Целью отладки является поиск и исправление обнаруженных ошибок поведения программы. По-видимому, самый простой способ отладки состоит во вставке в текст программы дополнительных операторов, выводящих текущую информацию о состоянии исполнения (исполняемой инструкции, значении переменных и т. п.) или проверяющих условия, которые обязаны в данной точке выполняться. Вставка подобных дополнительных действий, которые (в идеале) не меняют поведение программы, а предназначены для анализа программы, называется *инструментированием*. Программист может после выполнения такой расширенной программы проанализировать ее вывод и, если повезет, понять причину ошибки. *Интерактивная отладка* дополнительно позволяет задавать точки останова, с достижением которых программа в отладочном режиме приостанавливается, указывает текстовому редактору позицию в исходном файле, позволяет пользователю проинспектировать текущее состояние вычислений, после чего продолжить или прервать исполнение. Современные системы предоставляют и более продвинутые средства отладки. Например, для точки останова можно указать дополнительное условие вида: «каждый 100-й раз, и только в том случае, когда x<eps». Бывает возможным «откатить» исполнение на несколько шагов назад, «пропустить» часть инструкций или даже изменить программу в процессе выполнения.

Еще одной задачей, которая, как и отладка, может быть решена инструментированием, является *профилирование*. Его целью является сбор информации о производительности программы или о потребляемых ресурсах. Это может быть полезно для обнаружения «горячих» точек программы, оптимизация которых может дать наибольший эффект. Понятно, что инструментирование должно учитывать то обстоятельство, что оно само может повлиять на производительность программы.

Как уже было сказано, потребность в отладке возникает, когда обнаруживается, что программа работает неправильно. Однако невозможно предсказать, когда ошибка проявит себя. Может пройти много лет эксплуатации до того момента, когда вдруг программа «сломалась», и нет никаких гарантий, что в программе не осталось других ошибок, либо что исправление найденной ошибки не приве-

ло к новой. Альтернативой отладки в этом смысле является *тестирование* [54], цель которого состоит в нахождении такого набора входных данных, правильная работа программы на которых, если и не доказывала, то убеждала бы в ее правильности. Система программирования может предоставлять средства для автоматического построения набора тестов как для отдельных компонент программы (например, функций), так и программы в целом. Построение полного набора тестов для любой программы является неразрешимой задачей. Поэтому ограничиваются лишь выполнением некоторого *критерия тестирования*, такого, например, как гарантия того, что каждая точка тестируемого компонента будет выполнена, либо того, что любой цикл выполнится не менее двух раз, и т. п. Система тестов (построенная либо автоматически, либо вручную) дает возможность для *регрессионного тестирования*: проверки того, что новая версия программы работает так же, как предыдущая. Ввиду того, что системы тестов оказываются весьма большими и, следовательно, требуют больших вычислительных ресурсов для проверки, отдельной задачей является минимизация системы тестов без существенной потери в достоверности.

Еще одним способом исключения ошибок в программе является *верификация* и *анализ*, т. е. автоматическое доказательство либо полного соответствия программы своей спецификации, либо некоторых существенных ее свойств, необходимых для правильности программы. Примерами таких свойств являются следующие утверждения:

- любой переменной присваивается некоторое значение до момента ее использования (это относительно легко сделать для локальных переменных функций и существенно сложнее для глобальных переменных);
- никогда не разыменуется пустой указатель NULL;
- открытие файла всегда предшествует чтению из него;
- индекс массива всегда находится в его границах и т. п.

Далеко не все подобные проблемы являются алгоритмически разрешимыми и даже частные решения, т. е. находящие не все потенциальные ошибки, могут быть чрезвычайно трудоемкими. Тем не менее, ущерб, который наносит ошибка при использовании программы, может многократно превосходить затраты на ее исключе-

ние при разработке. Следует отметить, что какие-то элементы семантического анализа выполняются и во время собственно трансляции. Однако если сделать глубокую верификацию и анализ семантики обязательной частью транслятора, то это слишком удлинит время трансляции. Разработчику транслятора приходится делать выбор между скоростью трансляции и глубиной семантического анализа. Самые сложные его варианты выносят в отдельные компоненты системы разработки, которые пользователь может применять по выбору. Тем не менее доказательное программирование [15, 24, 34], реализующее принцип *раннего обнаружения ошибок*, является перспективным направлением создания надежного программного обеспечения, которое все сильнее и сильнее проникает в среду индустриального программирования.

Самым трудоемким в жизненном цикле программного обеспечения является его *сопровождение*. Это объясняется следующими факторами:

- длительное время использования – иногда десятилетия, что особенно важно, учитывая, что за такое время могут кардинально поменяться как вычислительные средства, так и технология программирования;
- большое количество пользователей, от которых могут поступать разные запросы по развитию функциональности программы, причем запросы от разных пользователей могут быть несовместимы;
- большой коллектив разработчиков, который может со временем полностью поменяться.

Отчасти эти проблемы решают подсистемы *поддержки версий программного обеспечения*, которые позволяют одновременно работать над одной и той же программной системой нескольким разработчикам, не допуская конфликтов или разрешая их при параллельных изменениях одного и того же компонента, создавать новые версии программы, откатываться к старым, сливать версии и т. п.

Таким образом, современные системы программирования существенно ушли в своем развитии от собственно исполнения программы, и решаемые ими задачи намного многоплановее и сложнее.

5. ЯЗЫКИ ПРОГРАММИРОВАНИЯ

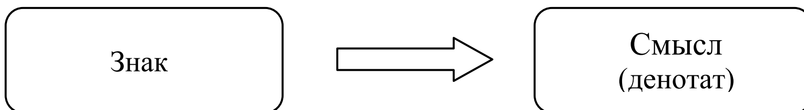
Мы не будем детально обсуждать вопрос о том, как реализуются трансляторы и интерпретаторы – это тема отдельного курса. Но нам необходимо понимать, какой смысл заключен в программе как в тексте на языке программирования хотя бы для того, чтобы наше представление соответствовало тому, что реализует система программирования. Как и в естественных языках, нам приходится решать две задачи:

- Как на данном языке выразить то, что мы хотим сказать? То есть нам требуется преобразовать смысл во фразу – задача *синтеза*;
- Как понять смысл фразы на данном языке? Это задача *анализа* или *разбора* фразы.

В своей практике программисты постоянно решают эти две задачи, пытаясь написать новую программу и понять то, что написали другие программисты.

Отличие от естественных языков состоит в том, то что языки программирования намного проще и, чаще всего, понимаются однозначно, а их правила могут быть в значительной степени описаны формально [4]. Ниже мы рассмотрим некоторые аспекты языков программирования и способы их описания.

Целью языка программирования как знаковой системы является сопоставление последовательности символов некоторого смысла:



Продемонстрируем это на простом примере. Попробуем понять, что означает запись

$$45.7$$

Естественным ответом будет: «Действительное число 45.7». Однако если мы попытаемся проанализировать, каким образом

---

[4] Здесь слово «проще» не означает, что языки программирования простые. Правильнее будет понимать это утверждение так, что языки программирования сложные, а естественные языки – совсем сложные.

мы дали такой ответ, то все окажется не так просто. Во-первых, исходно мы видим не число, а линии и пятна на бумаге или экране дисплея, которые мы идентифицируем как символы – десятичные цифры и точку, записанные в определенной последовательности. Далее мы анализируем эти символы и понимаем, что они образуют запись десятичного числа с дробной частью. Для этого мы убеждаемся, например, в том, что точка встречается только один раз; запись «8.383.363.40.20» как десятичное число не воспринимается. Далее мы подсознательно сопоставляем каждой цифре число в пределах от 0 до 9, применяем позиционную форму записи, умножая каждое из чисел на соответствующую (положительную или отрицательную) степень 10, и, наконец, все суммируя, получаем некоторый абстрактный объект, принадлежащий полю действительных чисел. Для того чтобы выразить полученный результат, мы делаем обратное преобразование.

Заметим, что в ходе этого мыслительного процесса мы подсознательно сделали несколько важных предположений. Во-первых, мы сразу решили, что это действительное число, а не, скажем, номер дома. Во-вторых, мы воспользовались десятичной системой счисления, хотя никто нам не говорил, что это именно десятичные цифры, а не восьмеричные или шестнадцатеричные. В-третьих, мы решили, что точка используется для отделения целой части от дробной (или иных каких-то целей), хотя в нотации, свойственной русскоязычной научной традиции, для этого следовало бы использовать запятую.

Так или иначе, мы вложили в исходную запись смысл, а весь процесс сопоставления реализовал *семантическую функцию*, отображающую последовательность символов в абстрактные действительные числа:

$$\mathrm{Sem}(\texttt{"45.7"}) = 4*10^1 + 5*10^0 + 7*10^{-1} = 45.7$$

Аналогичные, но гораздо более сложные процессы происходят и при сопоставлении смысла программам, исходным представлением которых является последовательность символов [5], а конечным, например, функция, реализующая отображение входных данных

[5] Если уж быть совсем точным, то исходным представлением является последовательность битов в памяти программы, а сопоставление им символов из входного алфавита требует дополнительного объяснения.

программы в выходные. Концептуально процесс разбивается на отдельные фазы:

- *лексический анализ* подобен орфографии естественного языка, где мы выделяем слова и знаки препинания, классифицируя при этом слова как существительные, глаголы, наречия, частицы и т. п. Лексический анализ выделяет так называемые лексемы, для каждой из которой известен ее класс – идентификатор, служебное слово, число, служебный символ и т. п. Отметим, что лексемы, хотя и могут иметь привязку к исходному тексту, являются абстрактными объектами, и для последующих стадий уже не важно, каким образом была получена, скажем, лексема «действительное число 45.7» – как запись «45.7» или «0.457e+2». Важно лишь знать класс лексемы – «число» и специфичный для этого класса атрибут – действительное число;
- *синтаксический анализ* получает на входе последовательность лексем и восстанавливает структуру программы подобно тому, как в естественных языках из слов и знаков препинания строятся предложения с указанием того, что является подлежащим, сказуемым, дополнением, подчиненным оборотом и т. п. Из предложений могут собираться параграфы, сноски, разделы, эпиграфы, главы, послесловия т. д. Таким образом, результатом анализа является иерархия объектов, относящихся к определенным языком синтаксическим категориям;
- *семантический анализ* на основе известной структуры текста определяет его смысл. В большинстве случаев семантика имеет композиционный характер. Например, смысл параграфа получается как композиция смысла составляющих его предложений. Однако зачастую для того, чтобы понять смысл отдельной фразы, необходимы определенные знания о тексте в целом. Простейшим примером является упоминание имени какого-либо персонажа, который появляется в предыдущей главе. Может также оказаться, что один и тот же текст имеет совершенно разные смыслы в зависимости от обстановки, в которой он воспринимается.

Помимо этих ключевых этапов, можно сказать, первичных этапов осмысления текста есть и другие, на которых мы осознаем иные важные свойства текста, такие как лаконичность, стиль и т. п.

Другой круг задач можно назвать *погружением программы в контекст*. Развивая приведенный пример, это понятие можно проиллюстрировать так: то, что символы 45.7 являются действительным числом, читателю, незнакомому с математикой, ничего не скажет, так как в его контексте нет этого понятия.

5.1. Лексика

Лексика языка программирования описывает, из каких слов строятся фразы, т. е. словарный запас. Под словами мы будем далее понимать произвольную последовательность символов входного алфавита. Абстракция понятия слова, при котором мы отвлекаемся от его написания, произношения и словоформы, называется *лексемой*. Типичными классами лексем в языках программирования являются следующие:

- числа: `123.4e2`, `12`, `0x25`
- знаки: `+, !=, [, <<, <`
- идентификаторы: `i`, `Pi2`, `PersonID`
- ключевые слова: `while`, `if`
- строки: `"Hello, World", "while + 1"`
- символы: `'a'`

Задачей *лексического анализа* является разбиение входного текста программы на поток [6] лексем – выходной текст, каждый элемент которого содержит

- *класс лексемы* – признак того, является ли лексема идентификатором, строкой, числом и т. д.;
- *значение лексемы*, зависящее от ее класса: для чисел это значение числа, для строк – последовательность составляющих ее символов, для идентификаторов – приведенное к каноническому виду имя, либо номер идентификатора в глобальной таблице идентификаторов, и т. д.;

---

[6] Под потоком здесь понимается последовательность, конструируемая по мере необходимости. Так, рассматриваемому ниже синтаксическому анализу, который использует результаты лексического, может не требоваться вся последовательность сразу, поскольку он выбирает очередные лексемы одну за одной. Это, в частности, даёт возможность совместной работы лексического и синтаксического анализов.

- *привязку к исходному тексту*, т. е. имя файла, номер строки и позицию в строке, где лексема появилась.

Для лексического анализа необходимо определение того, что же является лексемой в данном языке. Попробуем начать с неформального определения. Например, мы можем сказать, что в языке имеются идентификаторы, которые представляются последовательностями букв и цифр, начинающихся с буквы. На первый взгляд, это вполне понятное определение, но при детальном рассмотрении приходится делать много уточнений:

- Что понимается под «буквой»? Допускается ли кириллица? Например, является ли идентификатором слово "`индекс`"?
- Различаются ли прописные и строчные буквы? Например, `PersonID` и `PerSoniD` – это один и тот же идентификатор или разные?
- Есть ли ограничение на длину идентификатора? Не слишком ли длинен идентификатор `TheBestApproximationReachedSoFar`? А если такое ограничение есть, то оно обеспечивается тем, что слишком длинные идентификаторы приводят к ошибке на стадии лексического анализа, или тем, что лишние символы просто игнорируются?
- Допускается ли использование подчеркивания в идентификаторе, как в `student_count`, а если допускается, то в каком месте? Может ли идентификатор начинаться с подчеркивания или заканчиваться им, как в `__FILE__` и `_1`? А состоять из одних подчеркиваний, как в `_` или `___`?
- Не забыли ли мы про другие символы? Некоторые языки позволяют использовать вопросительный знак в конце идентификатора, как в `IsLegal?`, а также символы `#`, `@` и т. п.
- Можно ли в идентификаторе использовать пробелы? Если да, то входят ли они в имя идентификатора или игнорируются?

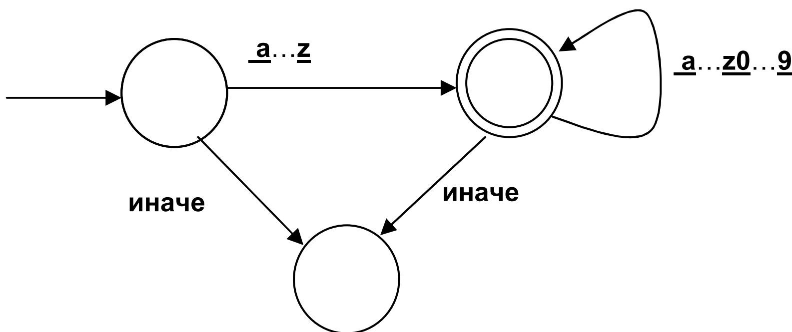
Таким образом, определение лексики требует более строгого, формального описания. Наиболее часто для этой цели используются *регулярные выражения* [1, 11, 26, 31, 40, 57]. Так, например, описание идентификатора может быть записано следующим образом:

(_|[a..z])(_|[a..z]|[0..9])*

Регулярные выражения сами образуют язык в том смысле, что они имеют свою лексику, а каждое регулярное выражение – структуру и определенный смысл. Приведенное выше выражение разбивается на два: (_|[a..z]) и (_|[a..z]|[0..9])*. Первое из них в свою очередь тоже имеет две «альтернативные» части: _ и [a..z], а второе – «применяет» операцию повторения * к подвыражению (_|[a..z]|[0..9]) и т. д. Следуя структуре регулярного выражения, можно определить его смысл – множество допускаемых им цепочек. В данном случае выражение «читается» следующим образом:

- цепочки начинаются либо с _, либо с символа в диапазоне от a до z.
- после чего повторяются ноль или более раз группы символов, где каждая группа:
    - либо _,
    - либо символ в диапазоне от a до z,
    - либо символ в диапазоне от 0 до 9.

Регулярные выражения удобно использовать для описания множества лексем. Обратная задача – определение принадлежности данной последовательности символов множеству лексем эффективно решается с помощью конечных автоматов [1, 11, 26, 31, 40, 57]. Известно, что по любому регулярному выражению можно построить разрешающий конечный автомат. В нашем примере такой автомат может иметь следующий вид:



Вершины диаграммы автомата обозначают его состояния. В процессе распознавания последовательности символов надо «пройти» от начального состояния, к которому ведет дуга «извне»,

к заключительному состоянию, отображаемому двойным кружком, переходя на каждом шаге в новое состояние по дуге согласно очередному символу.

Следует отметить, что не все аспекты обработки входного текста охватываются лексическим анализом, а в некоторых случаях не могут быть формально описаны регулярными выражениями. Например,

- пробелы, переводы строк, табуляции в большинстве случаев рассматриваются не как лексемы, а как разделители лексем;
- в «старых» языках программирования может сохраняться привязка текста к перфокартам, и некоторые позиции, к примеру, с 1 по 7 и с 72 до конца строки игнорируются [7];
- различного вида комментарии также рассматриваются как разделители. При этом язык может допускать, а может и не допускать вложенные комментарии: к примеру, в языке C:

```
/*начало комментария
      /*вложенный комментарий*/
конец комментария*/
```

текст «конец комментария */» на самом деле не будет частью комментария, как можно было бы предположить.

Таким образом, лексический анализ включает некоторую предобработку входного текста, после которой он может быть разобран конечным автоматом.

Отметим, что описание допустимых лексем не всегда позволяет однозначно выполнить лексический анализ, который, напомним, состоит в разбиении входного текста на последовательность лексем. Конфликт возникает, когда одна лексема является префиксом другой. Так, в языке C текст << рассматривается как одна лексема «сдвиг влево», а не как две лексемы «меньше». Утрированно можно задать вопрос, представляет ли AB один идентификатор или два – A и B? Обычно такой конфликт разрешается путем выделения максимально возможной лексемы.

[7] Вообще говоря, это сделать можно, но регулярные выражения получаются слишком громоздкими.

После того как мы выделили из входного потока текст очередной лексемы, необходимо вычислить ее атрибуты. Этот процесс тоже требует формального описания, поскольку не каждая текстуально правильная лексема имеет смысл. Например, запись вещественного числа `1e+100000000000` правильна с точки зрения регулярного выражения, но может оказаться, что такие большие числа не представимы данной реализацией языка. Поскольку разные представления лексемы могут давать одни и те же значения атрибутов, то мы можем рассматривать этот процесс как *нормализацию* лексем, т. е. приведение их к некоему нормальному виду, например:

- Числа `1.23` и `123e-2` приводятся к форме `0.123e+1`
- В языке C восьмеричное число `073` и десятичное число `59` приводятся к одинаковому двоичному представлению.
- Идентификаторы `Count`, `COUNT` и `coUnt` в языке Паскаль приводятся к `count`, поскольку регистр букв этом языке не различается.
- В языке Кобол число «ноль» может быть записано и как ZERO, и как ZEROS, и даже как ZEROES, но все эти записи приводятся к форме 0.

Некоторые идентификаторы преобразуются в специальные лексемы, называемые *ключевыми словами*, которые играют особую роль в определении синтаксиса, подобную знакам препинания в текстах естественных языков. Например, в языке C около 30 ключевых слов: `while`, `if`, `const`, `extern`, `enum` и т. д. Обычно языки программирования, по крайней мере изначально, запрещают использовать ключевые слова в качестве идентификаторов. Проблемы возникают по мере «взросления» языка и появления в нем новых синтаксических конструкций, которые требуют новых ключевых слов. В некоторых языках количество ключевых слов исчисляется сотнями и даже тысячами. Более того, может оказаться, что новые ключевые слова уже используются в программах, написанных на предыдущей версии языка. Чтобы смягчить остроту этой проблемы, из множества ключевых слов выделяют относительно небольшое множество зарезервированных, которые нельзя использовать для иных целей, а остальные становятся ключевыми только в определенном контексте. Следствием такого подхода является то, что лексический анализ нельзя выполнить независимо от синтаксического.

Некоторые языки программирования предоставляют возможность использовать произвольное слово в качестве идентификатора, если оно декорировано специальными символами. Например, некоторые диалекты SQL используют для этой цели квадратные скобки: `SELECT` – зарезервированное ключевое слово, `[SELECT]` – идентификатор с именем SELECT.

Поскольку мы уже затронули вопрос о кириллице, то рассмотрим подробнее проблемы, связанные с *национальными версиями* языков программирования. Ниже приведены на языке Алгол-60 три версии процедуры вычисления наибольшего общего делителя, использующей вспомогательную функцию вычисления остатка от деления двух целых чисел. Первая версия – чисто английская, где мы должны привлечь некоторое знание английского языка и перевести содержательные используемые понятия: «НОД» – greatest common devisor (GCD) и остаток – remainder (Rem). Во второй версии мы используем английские ключевые слова, но русские идентификаторы. Наконец, третья версия – чисто русская.

```
procedure GCD(x,y,z);
value x,y; integer x,y,z;
begin
  integer procedure Rem(A,B);
   value A,B; integer A,B;
   Rem := A – (A % B) * B;
   begin
    integer u;
    for u:=Rem(x,y) while u≠0
     do
      begin
       y := x; x := u
      end;
    end;
    z := x
end
```

```
procedureНОД(x,y,z);
value x,y; integer x,y,z;
begin
  integer procedure ОСТ(A,B);
   value A,B; integer A,B;
   ОСТ:= A – (A % B) * B;
   begin
    integer u;
    foru:=ОСТ(x,y) while u ≠ 0
    do
     begin
      y := x; x := u
     end;
   end;
   z := x
end
```

```
процНОД(x,y,z);
знач x,y; цел x,y,z;
начало
  цел проц ОСТ(A,B);
   знач A,B; цел A,B;
   ОСТ := A – (A%B)*B;
   начало
    цел u;
    для u:=ОСТ(x,y) пока u≠0
    цикл
     начало
      y := x; x := u
     конец;
   конец;
   z := x
конец
```

Можно было бы рассмотреть и вариант, в котором используется транслитерация с кириллицы на латинский алфавит: NOD вместо НОД и OST вместо ОСТ.

Если предположить, что программист совсем не владеет английским языком, то ему, конечно, третья версия покажется наиболее предпочтительной. Если программист владеет английским в объеме

словаря ключевых слов языка программирования, то ему подойдет и вторая версия. Однако он должен отдавать себе отчет в том, что в этом случае его код будет нечитаем для другого программиста, который не знает русского, если оба они работают в транснациональной компании.

Проблемы национальных версий этим не ограничиваются:

- Для «правильного» перевода бывает нужно менять не только лексику, но и синтаксис, структуру фраз.
- Программа разрабатывается и исполняется в окружающей обстановке, которая может и не поддерживать русские имена. Кроме того, если программа использует «иноязычные» библиотеки, большой набор которых обычно поставляется с системой программирования, то в тексте программы образуется смесь идентификаторов на разных языках.
- Если возникнет задача переноса программы на другую платформу, то имеющийся там транслятор может не поддерживать нужную национальную версию.
- Изображение данных в разных обстановках может также различаться:
  - числа – десятичная точка или десятичная запятая?
  - даты – 09/01/04 или 04/01/09?
- Если используются английские ключевые слова и русские идентификаторы, то становится очень неудобно набирать текст из-за необходимости частого переключения раскладки клавиатуры. Это на первый взгляд мелкая, но очень раздражающая проблема.
- Возрастает опасность совпадения разных букв по начертанию: например, если в идентификаторе ОСТ вторая буква окажется не кириллической «С», а латинской «C», то причина выдаваемой ошибки будет совсем неочевидна.

Таким образом, можно сделать вывод, что национальные версии языков имеют смысл только в том случае, если заранее ограничена область их применения. Примерами таких языков могут служить автокод Эль-76, разработанный специально для советского суперкомпьютера Эльбрус, язык и система программирования Альфа, для которых были сделаны даже специальные устройства ввода, внутренний язык системы 1С, ориентированный на специфику делопроизводства на российских предприятиях.

5.2. Синтаксис

Имея на входе поток лексем, синтаксический анализ разбирает структуру предложения языка программирования. В этом разделе мы будем рассматривать контекстно-свободный синтаксис, т. е. такой, при котором структура фразы определяется вне зависимости от того, в каком месте эта фраза написана. Следует отметить, что на данный момент контекстно-свободный синтаксис является стандартом *де-факто* для дизайна языков программирования, хотя есть языки и с более сложно устроенным с точки зрения теории формальных языков синтаксисом (например, Алгол-60). Для задания контекстно-свободного синтаксиса мы будем использовать РБНФ и синтаксические диаграммы.

5.2.1. Форма Бэкуса–Наура (БНФ)

Далее, для описания синтаксиса мы будем использовать *контекстно-свободные грамматики*, задаваемые в форме Бэкуса–Наура (БНФ). Это существенно более мощный механизм, чем регулярные выражения. Поэтому из соображений наглядности его можно использовать и для описания лексики. Однако регулярные языки допускают гораздо более эффективную реализацию.

Грамматика представляется в виде совокупности правил, каждое из которых дает «толкование» некоторому определяемому понятию, называемому также *нетерминалом*. Для одного и того же нетерминала может быть несколько правил, и в этом случае они рассматриваются как альтернативы. *Терминальными символами* (или сокращенно *терминалами*) называются элементы, которые не требуют дальнейшего раскрытия. В случае описания лексики терминалами являются символы входного текста, а в случае синтаксиса – лексемы. В дальнейшем нетерминальные символы мы будем выделять *курсивом,* а терминальные – **жирным** подчеркнутым. Тогда каждое правило грамматики имеет вид:

нетерминал ::= последовательность терминалов и нетерминалов

Значок «::=» читается как «это» («это есть»).

В таких обозначениях рассмотренное выше определение идентификатора может быть представлено следующей БНФ-грамматикой:

> буква ::= **_**
> буква ::= а
> ...[8]
> *буква* ::= **z**
> *цифра* ::= **0**
> …
> *цифра* ::= **9**
> *букра* ::= *буква*
> *букра* ::= *цифра*
> *букры* ::=
> *букры* ::= *букра букры*
> *идент* ::= *буква букры*

Для сокращения записи грамматики используются *регуляризованные БНФ* (РБНФ), которые допускают в правой части правил нотацию, свойственную регулярным выражениям. Так, несколько правил, определяющих один и тот же нетерминал, можно заменить одним, разделив правые части вертикальной чертой "|". Так мы сможем записать

> *буква* ::= **_**|**a**|…|**z**

Пару правил вида

> *можетбыть* ::=
> *можетбыть* ::= *чтото*

можно трактовать так, что *можетбыть* – это возможно отсутствующее вхождение *чтото*, и записать как одно правило, в котором квадратные скобки означают возможное отсутствие:

> *можетбыть* ::= [*чтото*]

Это позволит нам определить

> *букры* ::= [*букра букры*]

---

[8] Формально мы должны выписать все 26 правил для букв и 10 правил для цифр.

Повторение некоторой конструкции ноль или более раз задается в РБНФ с помощью итерации Клини *. То есть пара правил вида

несколько ::=
несколько ::= *один несколько*

может быть записана правилом:

несколько ::= *один**

Так мы можем определить *букры* как:

букры ::= (*букра*)*

Здесь скобки используются как спецсимволы для точного указания того, к чему применяется операция итерации. Зачастую удобно бывает и *ненулевая итерация*, обозначаемая +, т. е. повторение один или более раз. Пара правил вида

несколько ::= *один*
несколько ::= *один несколько*

может быть записана правилом:

несколько ::= *один*$^{+}$

Так, например,

букра (*букра*)* экививалентно (*букра*)$^{+}$
(*букра*)* экививалентно [(*букра*)$^{+}$]

Кроме того, если для некоторого нетерминала имеется единственное правило, то мы можем подставить его правую часть вместо использования этого нетерминала в другом правиле. Если же при этом нетерминал не появляется в правой части определяющего его правила, то это правило можно удалить.

Такие обозначения позволяют привести нашу грамматику для идентификаторов к следующему виду:

буква ::= **_**|**a**|…|**z**
цифра ::= **0**|…|**9**
идент ::= *буква* (*буква* | *цифра*)*

Переход от БНФ к РБНФ не привносит никакую содержательную информацию и служит лишь для сокращения записи – любую РБНФ можно преобразовать в эквивалентную ей БНФ (языки, порождаемые БНФ и РБНФ, – равны).

В качестве примера будем использовать РБНФ для описания арифметических выражений, которые строятся над переменными и константами с помощью знаков операций и скобок:

выр ::= **перем**
| **конст**
| (+ | -) *выр*
| *выр* (= | ≤ | <= | <> | + | - | * | /) *выр*
| **(** *выр* **)**

Отметим, что переменные и константы в этой грамматике являются терминалами.

Задача состоит в том, чтобы выяснить, допускается ли цепочка лексем данной грамматикой. Будем говорить, что из некоторого нетерминала N можно *вывести* цепочку терминалов $t_1,\ldots,t_n$, если существует последовательность цепочек $s_1,\ldots,s_k$, состоящих как из нетерминальных, так и терминальных символов, где $s_1 = N$, $s_k = t_1,\ldots,t_n$ и каждая последующая цепочка получается из предыдущей заменой некоторого нетерминального символа на правую часть одного из соответствующих ему правил. Такая последовательность цепочек называется *выводом*. Например, начиная с нетерминала *выр*, мы можем получить следующий вывод:

выр
выр+*выр*
x + *выр*
x + *выр* * *выр*
x + **2** * *выр*
x + **2** * **y**

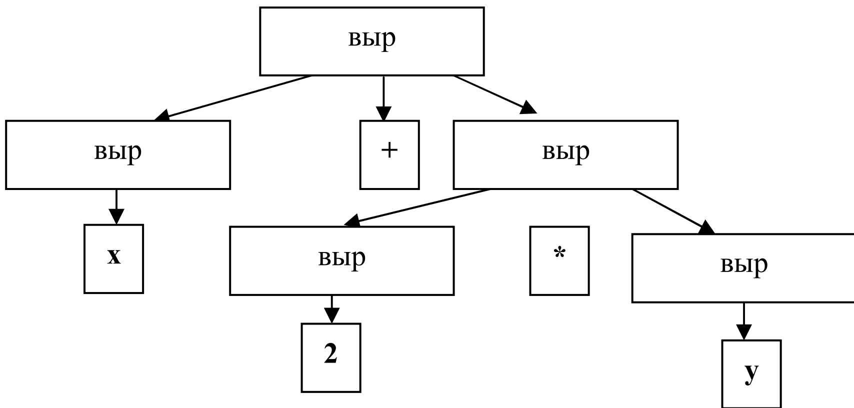
Здесь предполагается, что x и y обозначают лексемы-переменные, а 2 – лексему-константу.

В этом выводе мы в качестве заменяемого нетерминала всегда выбирали самый левый. Очевидно, что поскольку мы используем

контекстно-свободную грамматику, этот выбор оказывается несущественным. Более того, мы можем на каждом шаге заменять не один нетерминал, а все, укорачивая тем самым длину вывода.

Если в грамматике задан *главный нетерминал*, то говорят, что грамматика *допускает* некоторую цепочку терминальных символов, если ее можно вывести из главного нетерминала.

Для того чтобы сохранить и отобразить процесс вывода, используют *деревья вывода*, называемые также *деревьями разбора*. Корнем этого дерева является начальный нетерминал, листьями – терминалы, а любая вершина, являющаяся нетерминалом, имеет ровно столько потомков и ровно в той последовательности, сколько встречается в одном из правил для этого нетерминала. Рассмотренный ранее вывод может быть представлен следующим деревом разбора:

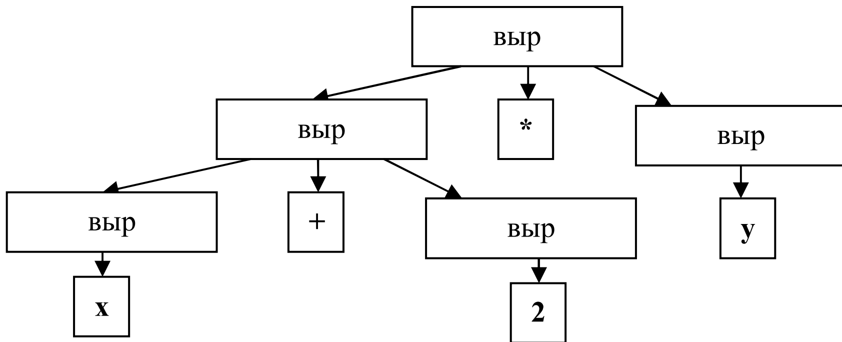


Искомая цепочка терминальных символов получается обходом всех листьев дерева слева направо.

Мы не будем вдаваться в подробности теории синтаксического анализа, основы которой можно найти в [1, 11, 57]. Отметим только, что эта задача с теоретической точки зрения эффективно разрешима.

Поскольку дерево разбора задает синтаксическую структуру предложения, на основе которой далее определяется его смысл, то принципиальным является вопрос об *однозначности*: может ли оказаться, что для рассматриваемого предложения существует несколько деревьев разбора в данной грамматике? Несложно

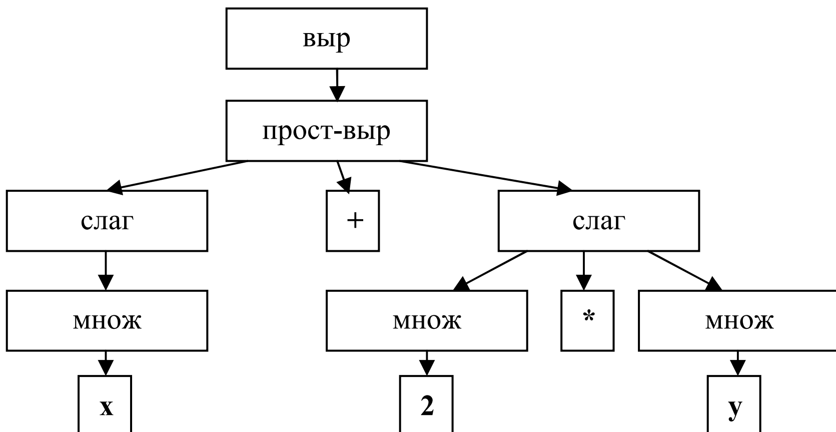
заметить, например, что **x+2*y** может быть разобрана и следующим образом:



В некоторых случаях возможно устранить неоднозначность, изменив грамматику. Здесь проблема возникла из-за того, что в грамматике никак не было отражено, что операции могут иметь разный приоритет. Изменим грамматику, введя вспомогательные понятия простого выражения, слагаемого и множителя:

выр ::= *прост-выр* [(= | < | <= | <>) *прост-выр*]
прост-выр ::= [+ | -] *слаг* ((+ | -) *слаг*)*
слаг ::= *множ* ((* | /) *множ*)*
множ ::= (*перем* | *конст* | (*выр*))

В этой грамматике выражение **x+2*y** может быть разобрано единственным способом:



Следует отметить, однако, что произведенные изменения в грамматике имеют и побочные эффекты. Например, стали недопустимы следующие выражения, которые допускались исходной грамматикой:

$$A < B + C < D$$
$$+ - + 2$$
$$X + - Y$$

В нашем случае можно считать, что мы ничего не потеряли, поскольку такие выражения были «неправильными» или «избыточными», однако, в общем случае придется показать, что при изменении грамматики мы не повлияли на распознаваемый язык. Это поднимает вопросы о доказательстве эквивалентности и системах эквивалентных преобразований контекстно-свободных грамматик, но они выходят за рамки данного курса.

В некоторых случаях избавиться от неоднозначности путем преобразования грамматики не удается. Классическим примером является неоднозначность, связанная с условным оператором `if` в некоторых языках программирования. Рассмотрим, например, следующий оператор языка C:

```
if (x > 0)
  if (x < 2)
    x = x+1;
else
  x = x-1;
```

Судя по расположению текста кажется, что внешний оператор `if` имеет две ветки – `if (x<2)x=x+1;` и `x=x-1;`. На самом деле это не так: внешний оператор `if` имеет только одну ветвь, а оператор `x=x-1;` относится к внутреннему, ближайшему `if`. Конечно, можно решить проблему, расставив скобки:

```
if (x > 0)
{
  if (x < 2)
    x = x+1;
}
else
  x = x-1;
```

но ее бы вообще не возникло, если бы условный оператор имел обязательный завершитель, как, например, в языке Modula-2:

```
IF x>0 THEN
  IF x<2 THEN
    x := x+1
  END IF
ELSE
  x := x-1
END IF
```

5.2.2. Синтаксические диаграммы

Еще один, и в некоторых случаях более наглядный, способ задания контекстно-свободных грамматик представляют синтаксические диаграммы: определение нетерминального символа задается в виде структурированного ориентированного графа с одним входом и одним выходом, вершинами которого являются нетерминалы и терминалы. Вход и выход обозначаются стрелками следующего вида:

Структурированность означает, что каждый такой граф строится из подграфов с помощью одного из следующих способов композиции:

Обязательные элементы: — элемент1 – ... – элементn —

Здесь под *элементами* подразумеваются либо нетерминалы, либо терминалы, либо такого же вида подграфы. Отличие игнорируемого элемента от необязательного состоит в том, что его отсутст-

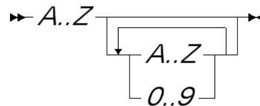
вие не влияет на смысл программы. Таким образом, игнорируемые элементы используются только для улучшения читаемости программы и относятся к тому, что называется «синтаксическим сахаром».

Диаграмма *допускает* цепочку терминалов, встречающихся на пути от входа к выходу с «заходом» в диаграммы, соответствующие встречающимся нетерминалам. Несложно показать, что любая контекстно-свободная грамматика может быть представлена диаграммой.

Пример: диаграмма для

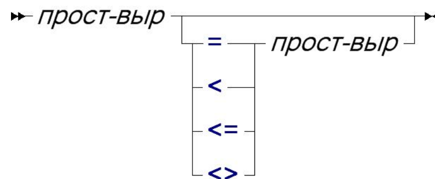
$$\textit{идент} ::= A..Z\ [(A..Z \mid 0..9)*]$$

имеет вид

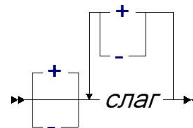


Пример: описанная выше грамматика для арифметических выражений задается совокупностью диаграмм:

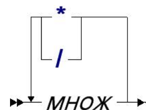
выр:



прост-выр:

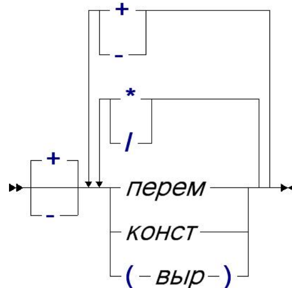


слаг:



множ:

Поскольку основное предназначение графического представления – облегчить восприятие информации, размер диаграмм часто выбирают так, чтобы сделать их обозримыми, в частности, чтобы они умещались на одной странице. Слишком большое количество диаграмм и вспомогательных понятий также нежелательно. Так, в нашем примере будет разумно избавиться от понятий *слаг* и *множ*, подставив их в диаграмму для *прост-выр*:



5.2.3. Устойчивость синтаксиса

Формальное описание синтаксиса и однозначность разбора несомненно важны для автоматической обработки текста программы. Однако не менее важно, чтобы текст программы воспринимался человеком [9]. А человеку свойственно ошибаться. Мы будем говорить, что синтаксис языка *устойчив к ошибкам*, если опечатки, слабо изменяющие текст программы, приводят к синтаксическим ошибкам. Иными словами:

1. Случайные ошибки и опечатки должны обнаруживаться;
2. Разные конструкции должны визуально различаться.

Поскольку данное определение неформально и весьма расплывчато, продемонстрируем понятие устойчивости на нескольких примерах. Рассмотрим следующий оператор цикла на языке C:

```
for (i = 0; i<N-1; i++);
    A[i] = A[i+1];
```

[9] Назначение любого языка программирования – это, во-первых, способ передачи алгоритмического знания от человека к машине и, во-вторых, средство накопления такого знания.

Очевидно, что здесь имелось в виду, что оператор присваивания `A[i]=A[i+1];` выполнится N–1 раз. Однако «случайная» точка с запятой в конце первой строки кардинально меняет структуру, и оператор присваивания оказывается вне цикла. Соответственно, цикл выполнит N–1 «холостую» итерацию, после чего пересылка будет выполнена лишь один раз. Заметим, что если бы оператор цикла имел завершитель, то такой ситуации не возникло, как, например, в языке Алгол-68:

```
.for i .from 0 .to N-2 .do
    A[i] := A[i+1]
.od
```

Рассмотрим пример на языке C, где присваивается переменная `y`, а источник присваивания достаточно длинный, чтобы разумно было записать его на двух строчках:

```
y = a[0]/2 + a[1]//3 + a[2]/5 + a[3]/7
   + a[4]/11 + a[5]/13 + a[6]/17 + a[7]/19;
```

«Случайно» удвоенный символ деления / превращается в начало комментария, заканчивающегося концом строки, но оператор остается синтаксически правильным и эквивалентным:

```
y = a[0]/2 + a[1]
   + a[4]/11 + a[5]/13 + a[6]/17 + a[7]/19;
```

Пример на языке Фортран, где цикл DO имеет вид:

`DO` *переменная-цикла=начальное-значение* [, *шаг*], *конечное значение*

В следующем операторе «случайная» точка вместо запятой в заголовке цикла

```
10 DO I = 1.2,N
      S = S * I
   CONTINUE
```

приводит к тому, что шаг цикла становится равным не 2, а умолчательному, равному 1.

Последний пример – на языке Алгол-68:

```
for i from 10 .to N .do
    print(" ")
od;
```

Здесь сыграли роль три решения, заложенных в синтаксисе языка. Во-первых, ключевые слова отличаются от идентификаторов точкой в начале. Во-вторых, внутри идентификаторов для лучшей читаемости можно использовать пробелы. В-третьих, в цикле `.for` можно опускать начальное значение, по умолчанию равное 1. В результате «случайно» забытая точка перед ключевым словом `.from` делает цикл эквивалентным

```
.for ifrom10 .from 1 .to N .do
      print(" ")
.od;
```

Таким образом, устойчивость синтаксиса языка также служит для раннего обнаружения ошибок: гораздо дешевле и безопаснее выявить ошибку на этапе синтаксического анализа, чем откладывать это на неопределенный срок с непредсказуемыми последствиями.

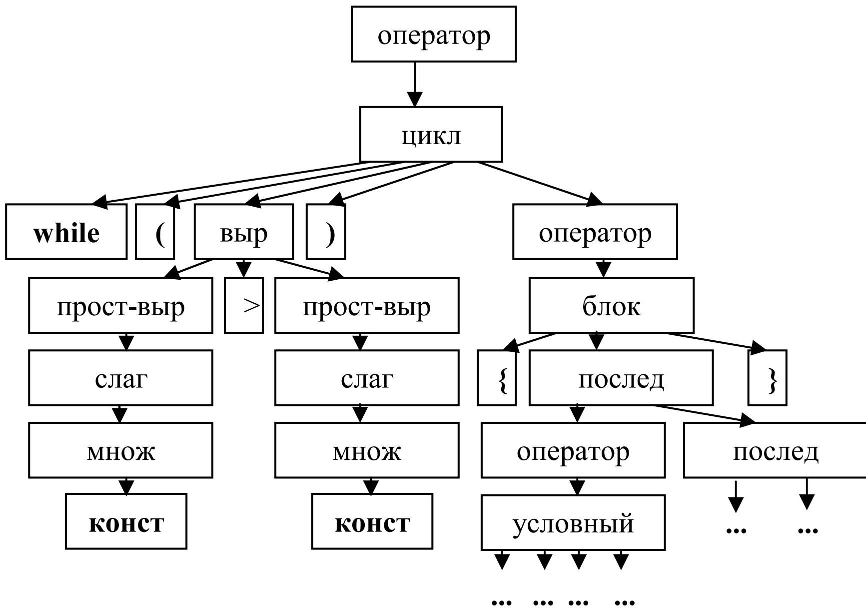
5.3. Абстрактный синтаксис

Дерево разбора программы может оказаться избыточно детальным для последующей обработки.

Например, если у нас имеется оператор цикла

```
while (n > 0)
{
    if (n%2)
        y = y*x;
    x = x*x;
    n = n / 2;
}
```

то синтаксическое дерево для этого цикла может иметь примерно следующий вид:



Такое дерево несомненно полезно, если мы, например, занимаемся рефакторингом и собираемся осуществлять синтаксические преобразования. Однако если нас интересует смысл программы (например, мы выступаем в качестве исполнителя данной программы), то нам уже не интересны подробности о том, что в записи оператора цикла или условного оператора имеются скобки, что последовательность операторов заключается в фигурные скобки, а также вся последовательность промежуточных правил вывода типа «выр→прост-выр→слаг→множ→...», которая вполне возможно имела лишь вспомогательный характер в задании грамматики языка. По существу, нам интересно лишь то, что есть оператор цикла, у которого есть условие и тело, состоящее из трех операторов, для каждого из которых нужны аналогичные сведения. Таким образом, мы абстрагируемся от того, как именно в языке записываются те или иные конструкции и выделяем лишь их существенные свойства и связи. В результате можно сформулировать так называемый *абстрактный синтаксис*, в котором каждое понятие озна-

чает синтаксическую категорию, для каждой альтернативы понятия указан ее тип, а также перечень именованных и типизированных атрибутов и связей. Естественно, что для абстрактного синтаксиса становятся неуместными вопросы приоритета операций, однозначности, устойчивости и т. п. Абстрактный синтаксис языка, на котором записана приведенная выше программа, может иметь следующий вид:

программа ::= оператор*

оператор ::= (while условие:выр тело:оператор*)

| (if условие:выр то:оператор* иначе:оператор*)

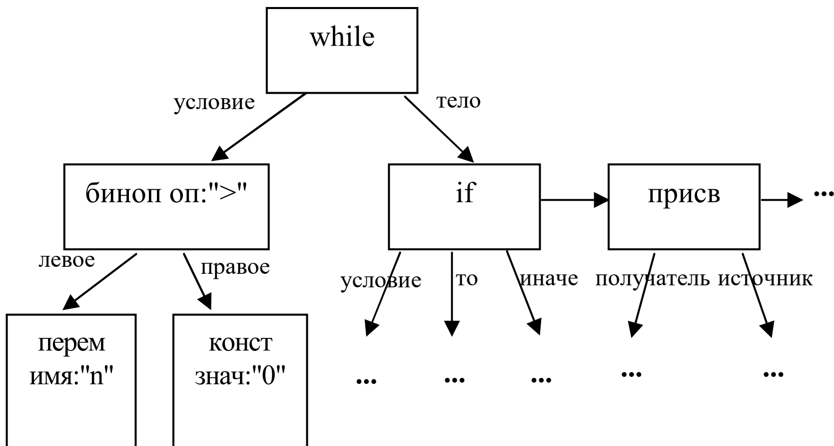
| (присв получатель:перем источник:выр)

выр ::= (перем имя:строка)

| (конст знач:число)

| (бинои оп:(+,-,...) левое:выр правое:выр)

Абстрактный синтаксис можно рассматривать как набор правил для формирования *дерева абстрактного синтаксиса*, в котором каждая вершина помечена типом вершины и значениями атрибутов, а дуги – именами составных частей. Например, дерево абстрактного синтаксиса того же цикла будет иметь следующий вид:



Нарисовав такое дерево, мы неявно ввели конкретный синтаксис для языка абстрактного синтаксиса, хотя он и имеет графическую

форму. То же дерево может быть изображено и в текстовой форме, которой мы будем придерживаться в дальнейшем:

```
(while
  условие:(бинop оп:">"левое:(перем имя:"n")правое:(конст знач:"n"))
тело:{(if
   условие:(бинoп оп:">"
          левое:(перем имя:"n")
          правое:(конст знач:"0")))
   то:{(присв
          получатель:(
          перем имя:"у"
          источник:(бинoп оп:"*"
               левое:(перем имя:"у")
               правое:(перем имя:"у"))))}
   иначе:{})
  (присв
          получатель:(
          перем имя:"x"
          источник:(бинoп оп:"*"
               левое:(перем имя:"x")
               правое:( перем имя:"x")))
  (присв
          получатель:(
          перем имя:"n"
          источник:(бинoп оп:"-"
               левое:(перем имя:"n")
               правое:(конст знач:"1"))))})
```

Фигурные скобки здесь означают список элементов, получающийся из грамматической конструкции ненулевой итерации.

При переходе от конкретного синтаксиса к абстракному возможны также различные преобразования, связанные с минимизацией количества абстрактных понятий. Например, все формы цикла заменяются одной, у условного оператора добавляется отсутствующая иначе-часть, явным образом вставляются все умолчательные действия и т. п.

5.4. Контекстно-зависимый анализ

Некоторые свойства синтаксиса зависят от контекста и не могут быть описаны контекстно-свободными грамматиками. Но даже в тех случаях, когда такое описание возможно, оно может оказаться неоправданно сложным. Пусть, например, понятие X определено как последовательность понятий (разделов) A, B, C, D и E, некоторые из которых могут повторяться:

$$X ::= (A \mid B \mid C \mid D \mid E)^*$$

но так, чтобы выполнялись следующие ограничения:

- Должен появиться либо раздел A, либо раздел B, но не более одного раза.
- Раздел C может появляться, только если указан раздел B.
- Все разделы E могут появляться, если указан раздел A, но после всех разделов D.

Мы можем изменить грамматику, введя вспомогательные понятия XA – «X, в котором есть A», XB – «X, в котором есть B», XBC – «X, в котором B появляется до C», XCB – «X, в котором C появляется до B», XEA – «X, в котором хотя бы одно E появляется до A», XAE – «X, в котором все E появляются после A»:

$$X ::= XA \mid XB$$

$$XA ::= XAE \mid XEA$$

$$XAE ::= D^* \; A \; D^*E^*$$

$$XEA ::= D^*E+ \; A \; E^*$$

$$XB ::= XBC \mid XCB$$

$$XBC ::= D^* \; B \; D^* \; C \; D^*$$

$$XCB ::= D^* \; C \; D^* \; B \; D^*$$

Хотя преобразованная грамматика в точности выполняет указанные ограничения, она существенно труднее для понимания, чем словесно заданные правила, да и с точки зрения реализации синтаксического разбора может оказаться менее эффективной. Поэтому в подобных случаях для языков программирования обычно ограничения оставляют в виде дополнительных условий к грамматике,

предполагая, что они будут проверяться дополнительным проходом по дереву разбора.

Отдельного прохода требует *идентификация объектов*, т. е. сопоставление определений именованных объектов – переменных, типов, функций, и т. п. – с их использованиями. Обычным требованием здесь является то, что для каждого использования должно быть хотя бы одно определение. Если язык допускает более одного определения, то они должны не противоречить друг другу. Формально контекстные ограничения такого вида могут быть описаны, например, атрибутными грамматиками [1].

В некоторых случаях синтаксическая структура фразы может зависеть от результатов идентификации. Например, в языке Алгол-68 предложение

```
.A x := 2
```

может означать описание переменной `x` с начальным значением 2, если `A` описан как тип, а может – присваивание 2 по адресу `(.A x)`, если `A` определена как операция, вырабатывающая адрес. Аналогичная ситуация возникает и в языке C, где конструкция

```
x * y;
```

может оказаться либо оператором, вычисляющим выражение `x*y`, если `x` – переменная, либо описанием переменной `y` типа указатель на `x`, если `x` описан ранее как тип, т. е. в следующей ситуации:

```
typedef float x;
...
x * y;
```

Наиболее существенную часть контекстно-зависимого анализа составляет *статический анализ типов*, т. е. определение (вывод) типов объектов и выражений и проверка типовой правильности. Далее мы еще обсудим понятие системы типов в языках программирования [10].

---

[10] В общем случае исчисление типов в современных языках программирования может быть весьма сложно как с точки зрения формального описания, так и с точки зрения реализации.

5.5. Семантика

Теперь, когда разобрана структура программы и установлены все связи между объектами и их типы, мы можем приступать к семантике, т. е. определению того, что делает данная программа – смысла исполнения программы некоторым исполнителем [21, 24, 34, 46]. Формальное описание семантики – весьма сложная задача, и поэтому зачастую разработчики языка ограничиваются словесными описаниями. Естественно, это оставляет вероятным неоднозначное и неполное толкование со всеми вытекающими из этого последствиями. Например, две разные реализации одного и того же языка могут работать по-разному (например, в языке C некоторые операции в зависимости от контекста являются реализационно-зависимыми). То же относится и к любым инструментам, опирающимся на семантику: анализу, верификации и т. п. С другой стороны, если формальное описание оказывается таким сложным, что обычный программист не может в нем разобраться или не хочет тратить на это свое время, то он вообще остается беспомощным. К тому же неформальное – совсем не обязательно означает неточное или неполное.

Далее мы кратко опишем несколько подходов к формальному описанию семантики на примере рассмотренного ранее простого языка, в котором есть только присваивания, переменные и циклы, а выражения строятся из переменных, констант и бинарных операций.

5.5.1. Денотационная семантика

Целью *денотационной семантики* является определение для каждой синтаксической категории так называемой *семантической функции*, которая сопоставляет каждой (абстрактной) синтаксической категории реализуемые ей функции.

Пусть D обозначает множество значений, с которыми работает программа. В нашем случае это множество целых чисел. Пусть X обозначает множество всех возможных имен переменных. Состоянием памяти в нашем языке можно считать частичное отображение вида

$$M : X \to D$$

сопоставляющее имени некоторое значение. Тогда вычисление выражения осуществляет отображение вида $M \to D$, а вычисление оператора и последовательности операторов – отображение вида $M \to M$. Семантическую функцию, сопоставленную конструкции s, мы

будем обозначать $Sem[s]$ [11]. Для определения семантики мы воспользуемся структурной индукцией [27, 40], т. е. семантика сложной конструкции будет определяться через семантику ее составных частей. Для вычисления выражений семантическая функция задается следующими правилами.

Вычисление константы просто извлекает это значение из дерева абстрактного синтаксиса:

$$Sem[(\text{конст знач:}val)](M) = val$$

Для вычисления переменной необходимо обратиться к памяти по имени переменной:

$$Sem[(\text{перем имя:}name)](M) = M(name)$$

Для вычисления бинарной операции необходимо вычислить подвыражения с указанным состоянием памяти и затем применить операцию над D, соответствующую операции, указанной в выражении:

$$Sem[(\text{биноп оп:}op \text{ левое:}lhs \text{ правое:}rhs)](M) = \\ = Func(op)\ (Sem[lhs](M), Sem[rhs](M))$$

Для обычных арифметических операций функция $Func$ определена естественным образом, например, $Func(+)(x,y) = x+y$. Однако надо понимать, что хотя оба «+» в этом равенстве выглядят совершенно одинаково, в первом случае это некоторый код операции, а во втором – абстрактная математическая функция. Для логических операций $Func$ определена несколько сложнее, поскольку у нас в языке нет булевых значений, например,

$$Func(>)\ (x,y) = \underline{\text{если}}\ x>y\ \underline{\text{то}}\ 1\ \underline{\text{иначе}}\ 0$$

Вычисление пустой последовательности операторов не изменяет состояние памяти:

$$Sem[\{\}](M) = M$$

[11] Формально нам нужны три разные семантические функции – отдельно для вычисления выражений, для вычисления операторов и для вычисления последовательности операторов. Мы будем их все обозначать одинаково, подразумевая, что понятно из контекста, какая именно функция используется в конкретном месте.

Для вычисления непустой последовательности операторов надо выполнить первый оператор и вычислить остальные операторы на новом состоянии памяти:

Sem[{*stat* ...}](*M*) = *Sem*[{...}] (*Sem*[*stat*] (M))

Для вычисления присваивания необходимо предварительно вычислить значение источника присваивания. Затем надо изменить состояние памяти. Новое состояние будет отличается от предыдущего только для переменной-получателя:

Sem[(присв получатель:*var* источник:*expr*)](*M*) =
= *M*[*var* =>*Sem*[*expr*](*M*)]

Вычисление условного оператора начинается с вычисления условия и затем сводится к вычислению либо то-, либо иначе-части в зависимости от того, отлично ли от нуля полученное значение:

Sem[(if условие:*cond* то:*then_seq* иначе:*else_seq*)] =
= если *Sem*[*cond*](*M*) != 0
то *Sem*[*then_seq*](*M*)
иначе *Sem*[*else_seq*](*M*)

Если при вычислении цикла условие оказалось ненулевым, то семантическая функция цикла возвращает данное состояние памяти. В противном случае необходимо вычислить семантическую функцию тела цикла, получить новое состояние памяти и снова применить к нему семантику цикла:

Sem[(while условие:*cond* тело:*seq*)] (*M*) =
= если *Sem*[*cond*](*M*) != 0
то *M*
иначе *Sem*[(while условие:*cond* тело:*seq*)] (*Sem*[*seq*] (*M*))

Заметим, что это последнее определение некорректно, поскольку оно нарушает принцип структурной индукции и определяет семантику цикла через саму себя. К решению этой проблемы можно подойти следующим образом. Обозначим

f = *Sem*[(while условие:*cond* тело:*seq*)]

Тогда определение можно трактовать так, что *f* не изменятся в результате следующего преобразования:

f = если *Sem*[*cond*](*M*) != 0 то *M* иначе *f* (*Sem*[*seq*] (*M*))

Функций *f*, удовлетворяющих этому условию, может быть бесконечно много, и нам необходимо выбрать самую «информативную» из них. Это достигается с помощью *оператора неподвижной точки*, который обычно обозначается *fix* и находит для любого монотонного функционала *F* наименьшую неподвижную точку, т. е. если $f = fix(F)$, то $f = F(f)$. Функционал *F* в нашем случае определяется как

$F(f)(M)$ = если $Sem[cond](M)$!= 0 то *M* иначе $f\,(Sem[seq]\,(M))$.

Подробнее о том, что такое монотонные функционалы и какие бывают операторы неподвижной точки, можно узнать из курса по денотационной семантике [16, 53].

Таким образом, мы представили формальное математическое определение семантики для очень простого языка программирования. При применении к конкретным конструкциям семантические функции можно упростить. Пусть, например, нас интересует семантическая функция оператора

```
n=n/2;
```

Для краткости будем записывать вместо абстрактного синтаксиса конструкций сами конструкции. Тогда

Sem[n=n/2;](*M*) =
= *M*[n =>*Sem*[n/2](*M*)] =
= *M*[n =>*Func*(/) (*Sem*[n](*M*), *Sem*[2](*M*))] =
= *M*[n => M(n) / 2]

формально описывает, что семантика данного присваивания заключается в замене текущего значения n на n/2. Как и ожидалось.

5.5.2. Операционная семантика

Операционная семантика объясняет смысл программы в терминах исполнения так называемой *языковой машины*. Для того чтобы работа этой машины была понятна, она должна быть либо определена как формальная вычислительная модель (например, машина Тьюринга), либо иметь достаточно точное и полное описание.

Определим языковую машину для нашего модельного языка. Состояние этой машины будет содержать:

1) текущую точку исполнения программы, которую будем изображать символом @. Поскольку программа представлена в виде дерева абстрактного синтаксиса, то точка исполнения должна быть привязана к некоторой вершине. Мы будем допускать положение точки исполнения как перед, так и после вершины. Например,

   @ (бинап оп:">"
левое: (перем имя: "n") правое:(конст знач: "0"))

   будет означать, что машина собирается вычислять применение бинарной операции,

   (бинап оп:">"
левое: (перем имя: "n")@
правое:(конст знач: "0"))

   – закончилось вычисление левого подвыражения и надо переходить к вычислению правого и т. п.;

2) состояние памяти обрабатываемой программы;

3) дополнительные атрибуты, которые можно задавать для вершин дерева абстрактного синтакиса. Например, для вершины-выражения мы можем определить дополнительный атрибут «результат» целого типа.

Описание операционной семантики заключается в наборе правил, переводящих текущее состояние языковой машины в следующее. Далее мы будем обозначать *текущее состояние памяти* обрабатываемой программы M, а следующее состояние памяти – M'.

При вычислении константы в атрибут-результат дублируется хранящееся в вершине значение, и текущая точка переставляется в конец выражения:

@((конст знач:*val*))
=> ((конст знач:*val* результат: *val*) @

При вычислении переменной в атрибут-результат помещается значение, извлекаемое по имени переменной из памяти [12], и текущая точка переставляется в конец выражения

@(перем имя:*name*)
=> (перем имя:*name* результат:M(*name*)) @

Для вычисления бинарной операции сначала точка управления переставляется в начало левой части:

@(бинОП оп:*op* левое:*lhs* правое:*rhs*)
=> (бинОП оп:*op* левое:@ *lhs* правое:*rhs*)

Когда точка управления оказалась в конце левой части, переставляем ее в начало правой:

(бинОП оп:*op* левое:*lhs*@ правое:*rhs*)
=> (бинОП оп:*op* левое: *lhs* правое: @*rhs*)

Когда же точка управления оказалась в конце правой части, то извлекаются сохраненные значения аргументов, удаляется атрибут-результат из вершин-подвыражений, вычисляется и добавляется атрибут-результат ко всему выражению, и точка управления переставляется в его конец:

(бинОП оп:*op* левое:(....результат:*lval*) правое:(... результат:*rval*) @)
=>(бинОП оп:*op* левое:(....) правое: (...)
результат:*Func*(*op*)(*lval*, *rval*)) @

Вычисление пустой последовательности операторов состоит просто в перестановке точки управления за эту последовательность:

@{} => {}@

Если последовательность не пуста, то точка управления помещается перед первым оператором последовательности:

@{S ...} => {@S...}

[12] Точнее сказать: получаемое применением памяти (как функции) к имени переменной.

Когда закончился некоторый оператор в последовательности, точка управления переставляется к следующему:

{...S1@ S2...} => {... S1 @S2...}

После выполнения последнего оператора вся последовательность операторов считается законченной:

{... S@} => {... S}@

Вычисления оператора присваивания начинается с вычисления источника:

@ (присв получатель:*var* источник:*expr*)
=> (присв получатель:*var* источник: @ *expr*)

По завершению вычисления источника изменяется текущее состояние памяти и точка управления переставляется в конец оператора:

(присв получатель:*var* источник:(... результат:*val*)@)
=> (присв получатель:*var* источник:(...)) @
и положить *M'* = *M*[*var* =>*val*]

Вычисление условного оператора начинается с выполнения условия:

@(if условие:*cond* то:*then_seq* иначе:*else_seq*)
=> (if условие:@*cond* то:*then_seq* иначе:*else_seq*)

По завершению выполнения условия точка управления переставляется в начало либо то-, либо иначе-части:

(if условие:(... результат:*val*) @ то:*then_seq* иначе:*else_seq*)
= если *val* != 0
то (if условие:(...) то:@*then_seq* иначе:*else_seq*)
иначе (if условие:(...) то:*then_seq* иначе:@*else_seq*)

Когда завершается выполнение то- или иначе-части, завершается выполнение всего условного оператора:

(if условие:*cond* то:*then_seq*@ иначе:*else_seq*)
=> (if условие:*cond* то:*then_seq* иначе:*else_seq*)@
(if условие:*cond* то:*then_seq* иначе:*else_seq*@)
=> (if условие:*cond* то:*then_seq* иначе:*else_seq*)@

Аналогично для выполнения цикла сначала надо вычислить условие:

@(while условие:*cond* тело:*seq*)
=> (while условие: @*cond* тело:*seq*)

По завершению вычисления условия, в зависимости от полученного значения точка управления переставляется либо в начало тела цикла, либо за весь цикл:

(while условие:(... результат:*val*)@ тело:*seq*)
=>если *val* != 0
то (while условие:(...) тело:@*seq*)
иначе (while условие:(...) тело:*seq*)@

Наконец, после завершения тела цикла, точка управления переставляется в начало условия:

(while условие:*cond* тело:*seq*@)
=> (while условие:@*cond* тело:*seq*)

В отличие от денотационной семантики нам не пришлось использовать оператор неподвижной точки. Однако очевидно, что выполнение программы может быть бесконечным и точка управления никогда не достигнет ее конца.

Отметим, что операционная семантика оказалась более сложной, чем денотационная, в частности, ввиду неочевидного перемещения точки управления по выражениям. От этого можно было бы избавиться, разбив сложные выражения, считая их своего рода синтаксическим сахаром. По существу, мы осуществим промежуточную трансляцию в подъязык, в котором выражение всегда образует одна операция:

программа ::= оператор*
оператор ::= (while условие:строка тело:оператор*)
| (if условие:строка то:оператор* иначе:оператор*)
| (инициализация имя:строка знач:число)
| (пересылка получатель:строка источник:строка)
| (операция получатель:строка источник:строка)
| (биноп получатель:строка
оп:(+,-,...) левое:выр правое:выр)

Исходная программа может быть легко преобразована в следующую:

```
r0 = 0;
r1 = n>r0;
while (r1)
{
  r2 = 2; r3=n%r2;
  if (r3)
    y = y*x;
  x = x*x;
  n = n / r2;
  r1 = n>r0;
}
```

Двигаясь дальше в этом направлении, можно было бы избавиться и от структурных условных операторов и циклов, «обогатив» язык метками и операторами безусловной передачи управления, однако в этом случае мы столкнулись бы с тем, что для определения следующего положения точки управления нам было бы необходимо «видеть» всю программу целиком и описание семантики существенно бы усложнилось.

5.5.3. Аксиоматическая семантика

Аксиоматическая семантика не дает непосредственного представления об исполнении программы, но тем не менее связана со смыслом программы, поскольку позволяет отвечать на вопросы о том, что делает программа. Например, для цикла, используемого нами в качестве примера,

```
while (n > 0)
{
    if (n%2)
        y = y*x;
    x = x*x;
    n = n / 2;
}
```

можно задать следующий вопрос: «Правда ли, что если перед циклом значения x, n и y были равны, соответственно, X, N и 1, и N неотрицательно, то после цикла n будет равно 0, а $y = X^N$?»

Аксиоматическая семантика строится на понятии предусловия и постусловия, а утверждения формулируются в виде троек (так называемой *тройки Хоара*):

$$\{P\}\ S\ \{Q\},$$

где P – предусловие, S – синтаксическая конструкция, Q – постусловие.

Для доказательства утверждений аксиоматическая семантика задает систему аксиом и правил вывода, сводя таким образом задачу к использованию математической логики. Система правил всегда включает в себя правило следствия – замены предусловия более сильным, а постусловия – более слабым:

$$\frac{P \Rightarrow P',\ Q' \Rightarrow Q,\ \{P\}\ S\ \{Q\}}{\{P'\}\ S\ \{Q'\}}$$

Продемонстрируем определение аксиоматической семантики на примере нашего простого языка. Будем считать, что смысл выражений в языке очевиден и, более того, мы можем использовать эти выражения при записи утверждений. Каждый оператор присваивания привносит в систему новую аксиому:

$$\{P[var \rightarrow expr]\}\ (\text{присв получатель:}var\ \text{источник:}expr)\ \{P\},$$

где запись $P[var \rightarrow expr]$ означает утверждение P, в котором все вхождения var заменены на $expr$. Например, пусть P = x>0, тогда для присваивания `x=x+1` мы получим

$$\{(x > 0)\ [x \rightarrow x+1]\}\ \texttt{x=x+1}\ \{x>0\},$$

что эквивалентно

$$\{x+1 > 0\}\ \texttt{x=x+1}\ \{x>0\}$$

и

$$\{x > -1\}\ \texttt{x=x+1}\ \{x>0\}.$$

Еще одна аксиома имеется для пустой последовательности операторов

$$\{P\}\ \texttt{\{\}}\ \{P\},$$

которая означает, что любое условие сохраняется в результате ее выполнения.

В остальных случаях синтаксические конструкции задают не аксиомы, а правила вывода. Для последовательности, состоящей из первого оператора S и остальной части "...":

$$\frac{\{P\}\ S\ \{R\},\ \{R\}\ \{\ldots\}\ \{Q\}}{\{P\}\ \{S, \ldots\}\ \{Q\}}$$

Для условного оператора:

$$\frac{\{\ P\wedge\neg E\}\ S_1\ \{Q\},\ \ \{\ P\wedge\neg E\ \}\ \{S_2\}\ \{Q\}}{\{\ P\ \}\ (\text{if условие:}E\ \text{то:}S_1\ \text{иначе:}S_2)\ \{\text{Q}\}}$$

Для цикла:

$$\frac{\{P\wedge E\ \}\ S\ \{P\}}{\{\ P\ \}\ (\text{while условие:}E\ \text{тело:}S)\ \{P\wedge\neg E\ \}}$$

Наибольшую сложность вызывает правило для цикла, поскольку в нем, в отличие от остальных случаев, одно и то же условие P появляется как в левой, так и в правой части. По этой причине, необходимо отыскать *инвариант цикла*, т. е. такое условие, которое не меняется в результате выполнения тела цикла, если до его выполнения было справедливо также условие цикла. В общем случае это алгоритмически неразрешимая проблема, хотя для конкретного цикла или для определенных классов программ инвариант можно построить.

Для определения инварианта в рассматриваемом примере введем дополнительную переменную k, которая подсчитывает количество итераций цикла: она полагается равной нулю перед циклом и увеличивается на 1 в конце цикла. Тогда в качестве инварианта *Inv* можно предложить следующее условие:

$$N > 0 \wedge x = X^{2^k} \wedge n = \frac{N}{2^k} \wedge y = X^{N[2^k]},$$

где $a[b]$ означает остаток от деления a на b, а деление подразумевает деление нацело. То, что это действительно так, еще предстоит доказать.

Для сокращения записи весь цикл обозначим WHILE, его тело – BODY, а условный оператор внутри тела – IF. Мы собираемся доказать, что

$$\{\ k = 0 \wedge n = N \geq 0 \wedge x = X \wedge y = 1\}\ \text{WHILE}\ \{\ y = X^N \wedge n = 0\}.$$

Применяя правила следствия и вывода для цикла, достаточно показать, что

1. $k = 0 \wedge n = N \geq 0 \wedge x = X \wedge y = 1 \Rightarrow Inv$
2. $Inv \wedge n \leq 0 \;\Rightarrow\; y = X^{\mathrm{N}} \wedge n = 0$
3. $\{Inv \wedge n > 0\}$ BODY $\{\, Inv\}$

Первое утверждение доказывается подстановкой k=0 в *Inv*, поскольку 2^{k}=1 и $N[2^{\mathrm{k}}] = 0$. Для доказательства второго утверждения достаточно заметить, что из $n = N \;\geq 0 \;\wedge\; n \leq 0$ следует $n = 0$, и далее

$$n = 0 \wedge n = \frac{N}{2^{\mathrm{k}}} \Rightarrow 2^{\mathrm{k}} > N \Rightarrow X^{\mathrm{N}[2^{\mathrm{k}}]} = X^{\mathrm{N}},$$

откуда получаем $y = X^{\mathrm{N}}$.

Осталось доказать третье утверждение, т. е. то, что *Inv* действительно является инвариантом цикла. Трехкратное применение правила для последовательности операторов и аксиомы для присвивания сводит его к доказательству утверждения для условного оператора

$$\{\, n > 0 \;\wedge Inv\}\mathrm{IF}\{Inv\left[x \to x^2, n \to \frac{n}{2}, k \to k + 1\right]\}.$$

Оно разбивается на два случая:

$$n - \textit{четно} \wedge n > 0 \wedge Inv \Rightarrow Inv\left[x \to x^2, n \to \frac{n}{2}, k \to k + 1\right]$$

и

$$n - \textit{нечетно} \wedge n > 0 \wedge Inv$$
$$\Rightarrow Inv\left[x \to x^2, n \to \frac{n}{2}, k \to k + 1, y \to yx\right]$$

В любом случае $n = \frac{N}{2^{\mathrm{k}}} \Rightarrow \frac{n}{2} = \frac{N}{2^{\mathrm{k+1}}}$ и $x = X^{2^{\mathrm{k}}} \Rightarrow x^2 = X^{2^{\mathrm{k+1}}}$. Пусть b_{i} – i-й разряд в двоичном представлении числа N, т. е. $b_{\mathrm{i}} = \frac{N}{2^{\mathrm{i}}}[2]$. Очевидно, что $N\left[2^{\mathrm{k}}\right] = \sum_{\mathrm{i}=0}^{\mathrm{k}-1} b_{\mathrm{i}} 2^{\mathrm{i}}$. Тогда

$$N\left[2^{\mathrm{k+1}}\right] = b_{\mathrm{k}} 2^{\mathrm{k}} + \sum_{\mathrm{i}=0}^{\mathrm{k}-1} b_{\mathrm{i}}\, 2^{\mathrm{i}} = \left(\frac{N}{2^{\mathrm{k}}}[2]\right) 2^{\mathrm{k}} + N\left[2^{\mathrm{k}}\right] =$$
$$= (n[2])2^{\mathrm{k}} + N\left[2^{\mathrm{k}}\right].$$

Если n – четно, т. е. $n[2]$=0, то

$$X^{N[2^{\mathrm{k+1}}]} = X^{N[2^{\mathrm{k}}]} = y.$$

Если же n – нечетно, т. е. n[2]=1, то

$$X^{N[2^{k+1}]} = X^{2^k + N[2^k]} = X^{2^k} X^{N[2^k]} = xy.$$

Что и требовалось доказать.

Аксиоматическая семантика может быть использована не только для спецификации результата вычислений программы. В рассмотренном примере несложно показать, что подсчитываемое количество итераций цикла k после его завершения равно $\lceil log\ N \rceil$.

5.6. Стиль

Две фразы могут выражать абсолютно один и тот же смысл, но одна воспринимается легко, а смысл другой понять сложно, потому что «так не говорят». Например, можно составить настолько сложное предложение, что к его концу будет уже непонятно, о чем шла речь в начале, хотя с точки зрения грамматики языка оно будет совершенно правильным. Эти проблемы характерны не только для естественных языков, но и для языков программирования.

Набор правил и соглашений о том, как надо оформлять программы, чтобы облегчить простоту их понимания, составляют *стиль программирования*. Поскольку речь идет о субъективном восприятии, то «правильный стиль» закрепляется корпоративными стандартами оформления программ. Наличие общего стиля особенно важно, поскольку одну и ту же программу в течение ее жизненного цикла может писать и исправлять большое количество людей. Зачастую свод правил может быть достаточно большим. Поскольку большая часть стилистических правил касается либо расположения текста, либо синтаксиса программы, то она может быть эффективно поддержана системой программирования, в частности, рефакторингом. Ниже мы рассмотрим несколько наиболее распространенных требований к стилю.

5.6.1. «Лесенка»

Текст должен располагаться так, чтобы выявлять синтаксическую структуру программы. Это достигается за счет того, что вложенные конструкции располагаются с некоторым (фиксированным)

отступом. Рассмотрим следующий фрагмент программы, в котором убраны все лишние пробелы:

```
int l1=busy_class(cl,d*lessons_per_day+t1);if(t1==t||l
1==-1||lessons[l1]->share[0].teacher!=tch)
continue;if(t1<t-1||t1>t+1)++not_sequence;else
{++total_class_overload;sum+=B_CLASS_OVERLOAD; }
```

Стандартным оправданием для такой записи является то, что «теперь все перед глазами». Иногда добавляют еще и заботу об эффективности: мол, в таком виде программа занимает меньше места [13] и транслятору придется меньше считывать и обрабатывать ненужной информации. Однако если теперь попытаться выяснить, к какому `if` относится `else` в предпоследней строке, то придется потратить несколько секунд для того, чтобы дать уверенный ответ. И это для программы из четырех строк! Если же таких строк будет несколько сотен, то задача будет практически неразрешимой без соответствующей поддержки.

Стилистически правильным будет следующее расположение того же самого текста:

```
int l1 = busy_class(cl, d*lessons_per_day + t1);

if (t1 == t
 || l1 == -1
 || lessons[l1]->share[0].teacher != tch)
    continue;

if (t1 < t-1 || t1 > t+1)
    ++ not_sequence;
else
{
    ++ total_class_overload;
    sum += B_CLASS_OVERLOAD;
}
```

Здесь выполнены следующие требования:

- каждый оператор начинается с новой строки;
- каждая открывающая фигурная скобка находится под ключевым словом охватывающего структурного оператора, а закрывающая – строго под открывающей;

[13] Соображения о размере исходного кода иногда звучат и сейчас, когда код программы в исходном виде передается по сети, например, для языка JavaScript.

- слишком длинные выражения разбиты на несколько строк;
- перед структурными операторами вставляются дополнительные пустые строки.

Напомним, что поскольку речь идет о стиле, то перечисленные выше требования не носят универсальный характер: правила могут быть немного другими, а набор их шире. В современных системах программирования правильные отступы при наборе текста поддерживаются автоматически, а в некоторых языках, как, например, Occam или Python, «лесенка» является обязательной и вложенность конструкций определяется их расположением.

Имеется, кажется, единственное исключение из этих правил, которое касается вложенных условных операторов, разбирающих несколько возможных альтернатив. Например, для записи

```
if (x >=1000)
  ...
else
  if (x > 0)
    ...
  else
    if (x == 0)
      ...
    else
      if (x > -1000)
        ...
      else // x <= -1000
        ...
```

используют следующую форму, которая не приводит к «сползанию» всего текста программы вправо:

```
if (x >=1000)
    ...
else if (x > 0)
    ...
else if (x == 0)
    ...
else if (x > -1000)
    ...
else // x <= -1000
    ...
```

Это настолько частый случай, что в некоторых языках программирования вводят специальное ключевое слово `elseif` и расширяют синтаксис условного оператора.

5.6.2. Неиспользование умолчаний

В языках могут использоваться разного рода умолчания. Придуманные когда-то с целью «облегчить» жизнь программиста, сейчас они широко признаются как потенциальный источник проблем и всячески не рекомендуются к использованию. Рассмотрим, например, следующий фрагмент программы, подсчитывающий количество строк в файле:

```
int cnt;
char line[128]
FILE * file;
...
while (fgets(line, 127, file) != NULL)
    cnt ++;
```

Предположим, что переменная `cnt` объявлена глобальной и программист знает, что реализация языка гарантирует инициализацию нулем всех глобальных целочисленных переменных. Поэтому чтобы избежать излишних действий по инициализации переменной, он положился на умолчание. Все это будет работать до тех пор, пока из каких-то соображений программист не решит оформить этот фрагмент в виде функции. Тогда совершенно неожиданно будет выдан непредсказуемый результат, и для того чтобы исправить ошибку, придется вспомнить про сделанное предположение и про то, что для локальных переменных функций в языке С инициализация не обеспечивается.

В смысле эффективности лучше положиться на то, что транслятор сам обнаружит излишние действия, а если даже не обнаружит, то безопасность кода в любом случае перевешивает соображения эффективности такого уровня. Отметим, что не следует путать «умолчание» и автоматический вывод каких-либо свойств транслятором. Например, современное использование служебного слова **auto** в современных версиях языка С++, которое позволяет не указывать явно тип переменной; он будет выведен автоматически транслятором. В результате типизация переменной будет проведена в соответствии со всеми правилами и следствиями из этого.

5.6.3. Мнемоничные идентификаторы

Критичной для понимания текста является мнемоничность используемых идентификаторов. Рассмотрим для примера следующий фрагмент, состоящий из двух вложенных циклов:

```
int n1, n2;
...
for (int index_of_outer_loop = 0;
         index_of_outer_loop < n1;
         index_of_outer_loop++)
    for (int intIndexJ = 0; intIndexJ < n2; intIndexJ ++)
    ...
```

Здесь программист решил дать длинные названия переменным циклов, чтобы явно их отличать друг от друга. Однако очевидно, что это не улучшило понимаемость программы. Прежде всего раздражает разный стиль в выборе названий. Для первой переменной было подчеркнуто, что это именно параметр цикла, тогда как во втором – что эта переменная целого типа. В любом случае, если предположить, что тело цикла занимает несколько десятков строк, то не требуется просматривать всю программу, чтобы понять, какая переменная к какому циклу относится. С другой стороны, для переменных `n1` и `n2` программист выбрал скромные, ничего не говорящие названия, подсказывающие только то, что они обе целого типа, поскольку начинаются с буквы «`n`», и что они как-то связаны между собой.

Правильнее в смысле читаемости программы было бы назвать переменные следующим образом:

```
int PersonCount;
int ExamCount;
...
for (int p = 0; p < PersonCount; p++)
    for (int e = 0; e < ExamCount; e ++)
          ...
```

Из названия первых двух переменных сразу понятно, что они обозначают количество человек и количество экзаменов. Переменные циклов названы коротко, но согласованно с именами тех объек-

тов, которые они индексируют. Общее неформальное правило можно сформулировать так: «длина идентификатора пропорциональна размеру области его действия».

Отметим еще одно замечание к указанному фрагменту: не следует располагать несколько описаний переменных на одной строчке, даже если это допускается языком.

5.6.4. Комментарии

Хорошо структурированная программа с мнемоничными именами во многом самодокументированна. Однако далеко не всегда все, что хочется и нужно сказать про программу, удается выразить средствами самого языка. В этом случае используются комментарии. Рассмотрим, например, следующий фрагмент программы:

```
int max = 0;
for (int i = 0; i < n; i++)
 if (M[i] > max)
      max = M[i];
```

Несмотря на его простоту, требуется некоторое время, чтобы понять, что именно он делает, а потом соотнести с тем, как он это делает. Программисты зачастую не любят писать комментарии, в частности, потому, что в момент составления программы знание «что» для них очевидно, а «как» – еще не вполне. Обратная ситуация у того, кто читает программу. И положение читателя в определенном смысле сложнее, поскольку почти любая программа делает больше, чем от нее требуется: например, устанавливает значения вспомогательных переменных. Поэтому комментирование текста программы зачастую насаждается административными методами. Скажем, требуется, чтобы любая подпрограмма имела содержательное описание своих аргументов и результатов, а также их допустимых значений. Более того, поскольку наличие комментариев можно достаточно легко проверить автоматически, то программисту просто не дадут «сдать» программу, если комментариев не хватает.

Однако бездумная вставка комментариев может иметь обратный эффект:

```
/* начальник приказал написать
комментарии к каждой строчке
- ему же хуже будет :-[ */
int max = 0;  // присвоить 0
// перебираем i=0..n-1
for (int i = 0; i < n; i++)
  if (M[i] > max)  // сравниваем с max
    max = M[i];   // обновляем, если надо
```

В этом примере комментарии, несмотря на их обилие, абсолютно ничего не добавляют содержательно собственно к коду. Более разумно комментарии к тому же фрагменту могли бы выглядеть следующим образом:

```
/*
 * Нахождение максимума max в массиве M
 */
int max = 0; // предполагается, что все M[i] > 0
for (int i = 0; i < n; i++)
  if (M[i] > max)
    max = M[i];
```

Здесь второй комментарий задает условие корректности, которое можно было упустить при беглом прочтении.

5.7. Прагматика

Прагматика языка – его соответствие поставленным целям. Под этим в зависимости от контекста понимаются довольно разные вещи. Так, некоторые языки создавались для решения определенного класса задач. Например, язык Кобол разрабатывался для создания обработки экономической информации, Фортран – для реализации научных расчетов, Modula-2 – для математического моделирования и т. д. При этом учитывается не только и не столько набор языковых конструкций, сколько возможность эффективной реализации в конкретной обстановке использования. Так, например, для преимущественно вычислительных задач вряд ли подойдут интерпретируемые языки или языки со сложным, неконтролируемым программистом распределением памяти. Но во

многих случаях оказывается, что язык, если и ориентирован, то не на класс задач, а на пристрастия и квалификацию программистов. Например, в языке Кобол нет никаких особых конструкций для управления прохождением финансовых транзакций, а в языке Фортран – ничего специального для обращения матриц.

Можно говорить и о прагматике отдельных языковых конструкций. Одно и то же содержательное действие может быть запрограммировано разными способами, подчеркивающими или, наоборот, скрывающими суть этого действия. Рассмотрим следующие четыре фрагмента, которые делают одно и то же – меняют значение `n` на его абсолютную величину.

`while (n<0){`
`    n = -n;`
`    break;`
`}`
	`if (n<0)`
	`    n = -n;`
		`n = (n<0?-n:n);`	`n>=0||n=-n;`

Первый способ следует признать самым неудачным, хотя он и делает все правильно, но циклы не предназначены для реализации выбора альтернатив. Выбор между вторым и третьим вариантом можно обосновать тем, что именно мы хотим подчеркнуть: то, что что-то выполняется лишь при отрицательном `n`, или то, что целью всей этой конструкции является получение нового значения `n`. Четвертый способ, хотя он и самый краткий, использует связку `||`, которая предназначена для вычисления логической дизъюнкции и не вычисляет второй аргумент, если первый истинен. Кроме того, он использует побочный эффект в выражении, что также нежелательно без весомых на то причин.

5.8. Преемственность

Языки программирования, как и программы, имеют свой жизненный цикл. Появление и развитие языков программирования может обосновываться множеством разнообразных причин: от необходимости решения практических задач до языкового оформления математических концепций. Причем практически всегда новые языки что-то заимствуют из уже существующих. Далеко не всегда, а точнее, лишь в редких случаях новые языки прорабатываются на предмет непротиворечивости, ортогональности, возможности к расширению и т. п.

Если язык живет достаточно долго, то в него включаются дополнительные возможности, отражающие новые области применения, либо популярные в текущий момент концепции. Причем причины, повлекшие расширение, могут со временем становится неактуальными, а конструкции или их следы в языке остаются. Например, исходное описание языка Алгол-60 – достаточно полное, хотя и неформальное – занимало лишь несколько десятков страниц, а описание его прямого наследника Unisys Algol – уже несколько тысяч. В частности, в нем имеются три независимых макропроцессора, специальные конструкции для работы с базами данных, коммуникационными протоколами и пр. Аналогичные истории случились с языками Фортран, Кобол, Лисп и многими другими «ветеранами».

Обратная совместимость, т. е. необходимость сохранения всего накопленного, является главным тормозом развития. Для того чтобы исключить или изменить какую-то языковую конструкцию, требуется сначала предупредить об этом все сообщество пользователей языка и подождать несколько лет, чтобы они могли переделать все существующие программы. Если за это время не нашлось весомых аргументов против, то внести изменения в язык. Понятно, что в такой обстановке проще оставить все как есть.

Язык C много заимствовал, в частности, от языков Фортран и Алгол-60. В свою очередь, он лег в основу целого семейства языков: C++, Java, C#, JavaScript и др. Язык C++ опять же для обеспечения обратной совместимости просто целиком включил в себя язык C. Это дало возможность сохранить весь накопленный багаж и плавно подвести программистов к использованию объектно-ориентированного программирования, но навсегда сделало язык внутренне противоречивым. Язык Java, напротив, унаследовал от языка C лишь стиль большинства синтаксических конструкций в расчете на то, что это облегчит освоение языка. Конечно, возможность использования накопленных библиотек должна быть обеспечена, но все новое программное обеспечение должно разрабатываться на новом языке.

Критикуя решения, принятые при разработке языка программирования, следует помнить, что они создавались в определенной обстановке, включающей уровень развития вычислительных средств и методов реализации языков программирования.

6. ПРЕПРОЦЕССОР

Далее мы будем рассматривать основные конструкции языка С. Хотя он и будет нашей основной целью, рассмотрение мы будем сопровождать сравнительным анализом с другими языками, что должно позволить нам судить о положительных и отрицательных последствиях решений, принятых разработчиками языка, и о том, как это можно было бы сделать иначе.

То, что мы обычно понимаем под программой на языке С, на самом деле является программой на языке препроцессора языка С, из которой уже получается программа на С. Препроцессор является макропроцессором, осуществляющим текстовую обработку на основе *директив*, вставленных непосредственно в текст. Некоторые *директивы* могут определять так называемые *макросы*, т. е. шаблоны правил для преобразования текста. Если такой шаблон встречается в тексте, то он называется *вызовом макроса* и заменяется согласно соответствующему правилу.

Рассмотрим, например, следующую программу:

```
#define SMALL
#ifdef SMALL
#define N 10
#define number short int
#else
#define N 10000
#define number long int
#end if
#define reverse(k) N - k
number A[N];
void main()
{
    for (int i = 0; i<N; i++)
         A[i] = reverse(i) * reverse(i);
}
```

Все, что здесь выделено жирным, относится к командам препроцессора. На самом деле и остальную, невыделенную часть можно рассматривать как команды препроцессора, суть которых состоит в том, чтобы перенести себя в результирующий текст.

В результате работы препроцессора получается следующий текст на языке С [14].

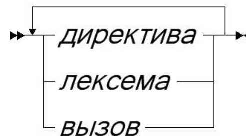
```
short int A[10];
void main()
{
    for (int i = 0; i<10; i++)
         A[i] = 10 - i * 10 - i;
}
```

6.1. Синтаксис

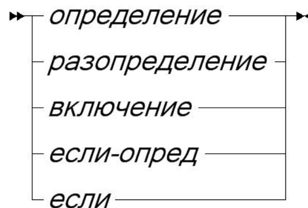
Поскольку мы говорим, что у препроцессора есть свой язык, то мы можем говорить и о всех свойственных ему понятиях – лексике, синтаксисе, семантике. Лексика препроцессора является расширением лексики языка С, т. е. помимо понятий идентификатора, строки, числа и т. п. в нем имеется еще несколько лексем, позволяющих задавать директивы. Кроме этого, поскольку препроцессор учитывает разбиение текста на строки, символ перевода строки также следует рассматривать как лексему. Мы будем обозначать ее ¶.

Весь текст состоит из директив, вызовов и отдельных лексем. Все директивы располагаются на отдельных строках и начинаются с символа #. Если директива очень длинная, то ее можно разбить на несколько строк, каждая из которых, кроме последней, завершается символом \.

текст:



директива:

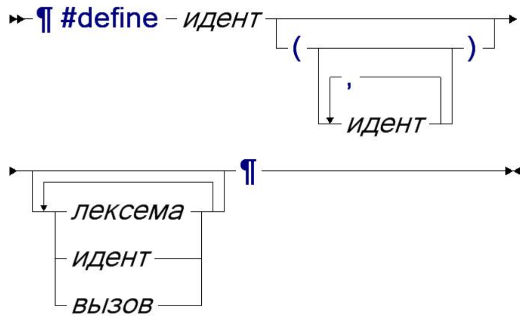


[14] На самом деле текст может выглядеть несколько иначе, поскольку в нем должны остаться команды, обеспечивающие его привязку к исходному тексту.

6.2. Макросы и вызовы

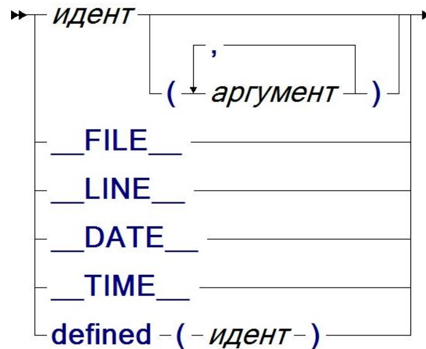
Директива-определение задает макрос. Макрос может иметь параметры, которые заключаются в скобки. Тело макроса – последовательности лексем, вызовов и использования параметров:

определение:



Макросы без параметров называются *макропеременными*. Вызов начинается с идентификатора определенного макроса, за которым могут идти параметры через запятую.

вызов:



аргумент:

Заметим, что слово «определенного» в предыдущем предложении делает разбор существенно контекстно-зависимым. Например, результатом обработки

```
F(1)
#define F 2+
F(x)
#define F(x) (x+3)
F(3)
```

будет текст

```
F(1)
2+(x)
 (x+3)
```

Первое вхождение F(1) вообще не является вызовом макроса, поскольку макрос F к этому моменту еще не определен, второе – вызов макроса без параметров, третье – вызов макроса с параметром.

Понятие аргумента макроса было введено из тех соображений, что значением параметра макроса может быть произвольная последовательность лексем, включая запятые и скобки, что может привести к неожиданным последствиям. Если бы аргумент был определен просто как последовательность лексем, то препроцессор при определенном макросе F, встречая текст

```
F(x , (y))
```

мог бы истолковать его разными способами:

1. вызов макроса с параметром "`x, (y)`";
2. вызов макроса с двумя параметрами "`x`" и "`(y`";
3. вызов макроса с двумя параметрами "`x`" и "`(y)`";
4. и т. д.

Как же передать отдельную скобку или запятую в качестве параметра? Для этого можно использовать следующий прием:

```
#define comma ,
#define F(x,y,z) x y z
F(a comma b)
```

который даст в результате

```
a , b
```

поскольку обработка параметров осуществляется после подстановки тела макроса в место вызова.

Есть несколько предопределенных псевдомакросов со специальным поведением – их раскрытие может зависеть от контекста:

- `__FILE__` – строка, равная имени текущего файла;
- `__LINE__` – число, равное номеру текущей строки в обрабатываемом файле;
- `__DATE__`, `__TIME__` – строки, задающие дату и время обработки данного вызова,
- `defined(`*имя*`)` – логическое значение, истинное тогда, когда определен макрос с данным *именем*. Полезность этого псевдомакроса станет понятна ниже, когда мы будем обсуждать условную компиляцию.

Следующий пример демонстрирует эти возможности:

```
#define COOL
#define N 25
#define begin {
#define end }
#define forever for ( ; ; )
#define printnum(n) fprintf(stderr,"%d",n)
#define printat() fprintf(stderr, \
                       "at:%s[%d]\n",__FILE__,__LINE__)
COOL forever
   begin
       printat();
       printnum(N);
   end
```

который преобразуется в следующий фрагмент на языке С:

```
for ( ; ; )
 {
  fprintf(stderr,"at:%s[%d]\n", "d:\\temp\\prog.c", 11);
  fprintf(stderr,"%d", 25);
 }
```

Макрос `N` является по сути определением константы. Это, по-видимому, самое распространенное использование препроцессора. Можно было бы завести вместо этого обычную переменную и присвоить ей в начале программы нужное значение. Однако, во-первых, доступ к переменной существенно более дорогая операция, чем доступ к константе. Во-вторых, транслятор может проводить

константные вычисления, заменяя, скажем, выражение `N*N + 1` на число 626. Если бы `N` была переменной, то выражение вычислялось бы каждый раз, когда до него доходило исполнение. Наконец, сам препроцессор и язык C в некоторых случаях требует, чтобы выражение можно было вычислить в процессе трансляции, например, при задании размеров массивов.

Отметим, что при определении макроса никаких подстановок не происходит. Это не дает возможности использовать макросы как переменные, значения которых можно перевычислять. Так, например,

```
#define A 1
#define A (A+1)
```

не даст ожидаемого связывания A равным 2, а приведет к бесполезному рекурсивному макроопределению. Связывание происходит только в момент вызова макроса. Поэтому в следующем примере:

```
#define B 1
#define A B
A +
#define B 2
A
```

мы получим на выходе текст

```
1 + 2
```

Довольно сложной проблемой, связанной с семантикой препроцессора, является привязка к исходному тексту. Рассмотрим следующий фрагмент программы:

```
int y;
#define sqr(x) (x,xx)
#define dist(x) sqrt(sqr(x))
dist(y)
```

для которой транслятор должен выдать сообщение о том, что переменная `xx` не определена. Если при этом транслятор укажет в качестве места ошибки вызов `dist(y)`, то это вызовет недоумение, поскольку в этом месте нет никакого `xx`. Если же в качестве места ошибки указать вхождение `xx` в первой директиве `#define`, то будет непонятно, каким образом этот макрос был вызван. Для того чтобы

программист смог разобраться в причине ошибки, транслятор должен выдать всю цепочку вызовов и подстановок параметров, которая привела к появлению `xx` в месте вызова `dist(y)`, что становится весьма затруднительно.

То, что язык препроцессора согласован с синтаксисом языка C только на лексическом уровне и то, что он, в отличие от языка C, учитывает разбиение текста на строки, может приводить к весьма неприятным последствиям. Рассмотрим, например, следующий фрагмент:

```
#define max (X, Y) ( X > Y
                ? X
                : Y)
max(A,B)
```

результатом которого естественно ожидать

```
 (A > B ? A : B)
```

Однако, оказывается, что результатом его станет

```
                ? X
                : Y)
 (X, Y) ( X > Y (A,B)
```

Во-первых, мы забыли, что при разбиении определения макроса на несколько строк надо в конце ставить символ \. Во-вторых, между именем макроса и открывающей скобкой не должно быть пробелов, поскольку иначе все, начиная с этой скобки, попадет в тело макроса [15]. Поэтому определение должно быть исправлено следующим образом:

```
#define max(X, Y) ( X > Y \
                ? X \
                : Y)
```

К счастью, в большинстве случаев такие неточности вызывают ошибки в результирующей C-программе, но понять и найти их бывает очень нелегко, поскольку с точки зрения препроцессора все прошло нормально.

---

[15] Типичный пример неустойчивого синтаксиса!

Еще один типичный пример неправильного использования препроцессора, но уже не вызывающий и синтаксических ошибок. Пусть определен макрос

```
#define reverse(x) 100-x
```

Тогда, если в программе встретилось выражение

```
reverse(20) * reverse(80)
```

то мы ожидаем получить значение 80*20 = 1600, хотя на самом деле результатом будет 100–20 * 100–80 = 2020. Для этого рекомендуется в определении макроса заключать в скобки все выражения:

```
#define reverse(x) (100-(x))
```

Еще большие неприятности может вызвать то, что препроцессор несогласован с языком C на семантическом уровне. Как уже было сказано, в препроцессоре сначала подставляется тело макроса вместо вызова, а уж затем обрабатываются параметры, в отличие от функций языка C, где сначала вычисляются значения параметров, а потом вычисляется тело функции. Например, если в программе встретилось «выражение»

```
max( f(A,B) , sqrt(A*A+B*B) )
```

то, поскольку `max` определен как макрос, реально будет выполняться следующий текст:

```
(f(A,B) > sqrt(A*A+B*B) ? f(A,B) : sqrt(A*A+B*B))
```

Во-первых, очевидно, что этот текст почти вдвое больше исходного. То, что здесь вычисление квадратного корня при ложности условия будет выполняться дважды, приводит к неэффективным вычислениям. Еще хуже, что функция `f` при истинности условия будет вызываться дважды, и, если она будет иметь побочный эффект (изменять глобальную переменную, печатать что-то в выходной файл и т. п.), то он проявится дважды, хотя в исходной программе явно написан один вызов.

Стандартным оправданием для использования макросов вместо функций является то, что вызов функции имеет существенные накладные расходы. Это действительно так. Однако, во-первых, как мы только что заметили, макросы могут приводить к существенно большей неэффективности, а, во-вторых, современные трансляторы

имеют весьма развитые средства анализа, которые позволяют в том случае, когда это обоснованно с точки зрения эффективности, аккуратно подставить тело функции в место вызова.

Те же причины делают бессмысленным определение рекурсивных макросов, хотя сам препроцессор это и не запрещает. Рассмотрим, например, макрос, рассчитанный на вычисление факториала:

```
#define fact(n) (n==0 ? 1 : (n)*fact(n-1))
fact(10)
```

Вызов этого макроса приводит к бесконечной рекурсивной подстановке

```
(10==0 ? 1 : (10)*(10-1==0 ? 1 : (10-1) * (10-1-1==0 ?
(10-1-1) * … )))
```

поскольку все, что происходит при вызове макросов – это операции с последовательностями лексем. Формально это приводит к зацикливанию препроцессора, и чтобы транслятор не «зависал», обычно препроцессоры ограничивают глубину вызовов макросов. Некоторые языки программирования – PL/I, Unysis Алгол и др. – имеют существенно более мощные, по сравнению с C, средства препроцессора, которые в том числе могут делать и вычисления, что позволяет выполнять циклы, условные операторы, рекурсивные процедуры, по сути генерирующие результирующий текст программы. Однако поскольку мы вообще очень негативно относимся к препроцессорам, то не будем углубляться в их изучение.

Дублирование кода препроцессором может привести к тому, что программист потеряет контроль над размером получаемого кода. Определим, например, функцию Фибоначчи с помощью макросов. Уже зная, что макрос в языке C не может быть рекурсивным, определим несколько специализированных функций – по одной для каждого значения аргумента:

```
#define f1 1
#define f2 1
#define f3 ((f2) + (f1))
#define f4 ((f3) + (f2))
...
#define f12 ((f11) + (f10))
```

Тогда вызов макроса

```
printf("%d", f11);
```

приведет к коду длиной 897 символов:

```
printf("%d", ((((((((((((((((((1) + (1))) + (1))) +
(((1) + (1))))) + (((((1) + (1))) + (1))))) +
(((((((1) + (1))) + (1))) + (((1) + (1))))))) +
(((((((((1) + (1))) + (1))) + (((1) + (1))))) +
(((((1) + (1))) + (1))))))) + (((((((((((1) + (1))) +
(1))) + (((1) + (1))))) + (((((1) + (1))) + (1))))) +
(((((((1) + (1))) + (1))) + (((1) + (1))))))))) +
(((((((((((((1) + (1))) + (1))) + (((1) + (1))))) +
(((((1) + (1))) + (1))))) + (((((((1) + (1))) + (1)))
+ (((1) + (1))))))) + (((((((((1) + (1))) + (1))) +
(((1) + (1))))) + (((((1) + (1))) + (1))))))))) +
(((((((((((((((1) + (1))) + (1))) + (((1) + (1))))) +
(((((1) + (1))) + (1))))) + (((((((1) + (1))) + (1)))
+ (((1) + (1))))))) + (((((((((1) + (1))) + (1))) +
(((1) + (1))))) + (((((1) + (1))) + (1))))))) +
(((((((((((1) + (1))) + (1))) + (((1) + (1))))) +
(((((1) + (1))) + (1))))) + (((((((1) + (1))) + (1)))
+ (((1) + (1)))))))))));
```

С одной стороны, в данном случае это дает возможность транслятору обнаружить, что все это выражение является константным, и вычислить его на этапе трансляции. Даже если бы оно не было константным, скажем, если бы мы заменили определения,

```
#define f1 x
#define f2 y
```

то транслятор мог бы обнаружить общие подвыражения, и с помощью вспомогательных переменных обеспечить, что каждое из них будет вычисляться один раз:

```
int r3 = y+x;
int r4 = r3+y;
int r5 = r4+r3;
...
int r11 = r10+r9;
printf("%d", r11+r10);
```

Это позволило бы почти на порядок сократить количество выполняемых операций сложения [16]. Однако может проявиться и обратный эффект: транслятор может отказаться делать сложные оптимизации для слишком больших функций, в результате чего эффективность окажется только хуже, как с точки зрения времени исполнения, так и с точки зрения памяти, поскольку объектный код так или иначе занимает какое-то место в памяти.

Директива-разопределение имеет следующий синтаксис:

разопределение: ¶ #undef – *идент* – ¶

и позволяет препроцессору «забыть» определение указанного макроса. Назначение этой директивы связано в основном с условной компиляцией, которая обсуждается ниже.

6.3. Включение файлов

Директива-включение задается следующим синтаксисом:

включение: ¶ #include " – *имя-файла* – " ¶
< – *имя-файла* – >

и предназначена для того, чтобы подставить в текущее место содержимое указанного файла, которое затем обрабатывается препроцессором. В этом смысле семантика директивы-включения практически совпадает с вызовом макроса без параметров, если считать, что в качестве тела макроса используется файл.

Синтаксис понятия *имя-файла* определяется операционной системой и ее файловой системой, в которой работает препроцессор; для MSDOS, Unix и OS2200 они могут быть устроены по-разному. Это может сделать программу непереносимой из одной обстановки в другую. Если имя файла указано в угловых скобках < … >, то файл ищется в системных директориях, которые указываются либо в конфигурации системы программирования, либо параметрами при запуске препроцессора. В противном случае если имя файла

[16] В общем случае вычисление чисел Фибоначчи в таком виде требует экспоненциального количества сложений, а после такой оптимизации – линейного.

заключено в кавычки, то поиск осуществляется относительно расположения текущего обрабатываемого файла. Если файл найти не удается, то его пробуют найти в системных директориях.

Примеры:

```
#include "main.h"
#include "..\\include\\person.h"
#include "../include/person.h"
#include "d:\\projects\\dialogs\\form.h"
#include <stdio.h>
#include "stdio.h"
```

6.4. Условная трансляция

Директива *если-опред* предназначена для реализации *условной трансляции* – включения фрагментов программы в результирующий текст только при выполнении (или невыполнении) некоторого условия. Изначально, с целью простоты обработки, язык C допускал лишь очень простые условия, заключающиеся в определенности макросов. Обычно для этой цели используются специальные макропеременные, трактуемые как логические, причем директива

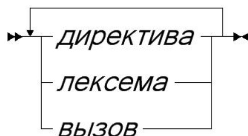
```
#define X
```

присваивает такой переменной истинное значение, а директива

```
#undef X
```

– ложное. Директива *если-опред* имеет следующий синтаксис:

альтернатива:



если-опред:

Для директивы `#ifdef(#ifndef)` первая альтернатива обрабатывается (не обрабатывается) только в случае, если на момент обработки директивы был определен (не определен) макрос *идент*, а в противном случае – следующая за `#else` вторая альтернатива, если она присутствует.

Наиболее часто условная трансляция используется для создания различных версий программы. Пусть, например, мы хотим иметь три версии программы для работы в разных операционных системах. Та непереносимость директивы `#include`, о которой мы говорили ранее, может быть решена следующим образом:

```
#ifdef MSDOS
#include "..\\dmsii\\cm.h"
#endif
#ifdef OSUNIX
#include "../dmsii/cm.h"
#endif
#if OS2200
#include "(WEBB0055)ALG/WEB/I/DMSII/CM."
#endif
```

Другой пример касается внутренней конфигурации программы, например, объема памяти, которую предполагается использовать:

```
#define SMALL
#ifdef SMALL
#define N 100
#define number short int
#else
#define N 10000
#define number long int
#end if
```

Условная трансляция часто используется для отладки. Дело в том, что для получения отладочной информации могут потребоваться дополнительные переменные и вычисления, которые не имеют смысла в «боевой» версии программы и должны быть

удалены из соображений эффективности. Обычно для таких целей определяют макропеременную, называемую DEBUG:

```
#define DEBUG // включить отладочный режим:
#ifdef DEBUG
#define iterStop 1000
int cnt = 0;
#endif
while (...)
{
  #ifdef DEBUG
  if (++cnt == iterStop)
  {
    fprintf(stderr,
      "Достигли очередной %d-й итерации", cnt);
    cnt = 0;
  }
  #ednif
  ...
}
```

Теперь для выключения отладочного режима достаточно поменять первую строку в файле на

```
#undef DEBUG
```

Этот метод легко может быть расширен на случай отладки разных видов путем заведения нескольких макропеременных.

Условная трансляция позволяет реализовать логические операции над макропеременными, используемыми в директиве #ifdef. Например, мы можем (хотя и очень некрасиво и неочевидно) определить макропеременную A = B && C следующей последовательностью директив:

```
#undef A
#ifdef B
#ifdef C
#define A
#endif
#endif
```

Макропеременную можно задать и специальным параметром, передаваемым препроцессору при запуске. Тогда отпадает необходимость менять исходные тексты и обеспечивать согласованность определения макропеременных в разных файлах.

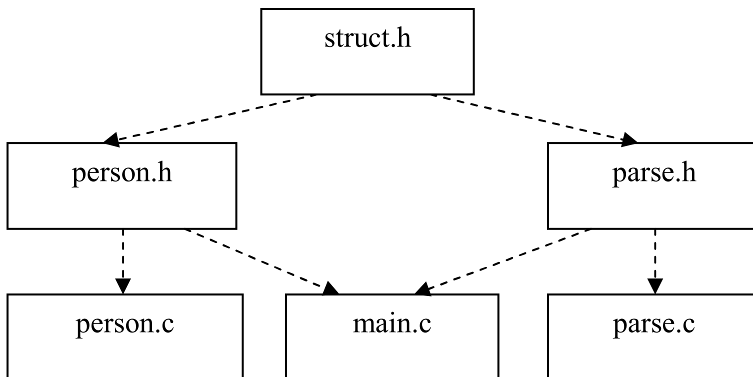
Еще одним типичным использованием условной трансляции является решение проблемы повторного включения одного и того же файла. Рассмотрим следующую ситуацию. Пусть мы определили два модуля, каждый из которых определяет некоторый набор функций и расположен в своем файле, скажем, `parse.c` и `person.c`. Для того чтобы эти функции можно было использовать в других модулях, их спецификации следует поместить в отдельные включаемые файлы, называемые, естественно, `parse.h` и `person.h`. Пусть теперь есть другой модуль, скажем `main.c`, в котором надо использовать и те и другие функции, что легко сделать, вставив в его начало директивы

```
#include “person.h”
#include “parse.h”
```

Пусть теперь оказалось, что и `parse.h`, и `person.h` используют описания структур данных, собранных в файле `struct.h`, т. е. в каждом из них есть директива

```
#include “struct.h”
```

В итоге получилась зависимость по включению файлов, отражаемая следующей диаграммой:



Тогда при обработке файла main.c препроцессор сначала встретит директиву

```
#include "person.h"
```

выполнив которую, обнаружит директиву

```
#include "struct.h"
```

в файле person.h и тем самым «поместит» содержимое struct.h в файл main.c. Закончив с person.h, препроцессор найдет в том же main.c директиву

```
#include "parse.h"
```

а в parse.h – снова директиву

```
#include "struct.h"
```

Таким образом, struct.h будет обработано дважды, что может в дальнейшем привести к синтаксическим ошибкам, поскольку транслятор не допускает повторного определения одной и той же структуры. Для того чтобы избежать этого, используется следующий прием. В начало любого включаемого файла, для которого следует избежать повторного включения, вставляется директива #ifndef с уникальным, предусмотренным специально для этого файла именем, традиционно имеющим суффикс _DEFINED. Тут же за ней вставляется директива #define с тем же именем:

```
#ifndef STRUCT_DEFINED
#define STRUCT_DEFINED
// содержимое файла struct.h
...
#endif
```

В итоге при первой обработке файла будет «установлена переменная» STRUCT_DEFINED и обработано остальное содержимое файла, а при любой последующей – только проверяться условие и обнаруживаться его ложность.

Со временем, по мере увеличения доступных препроцессору вычислительных мощностей, в язык были добавлены более сложные условия, представляемые выражениями языка С с некоторыми

ограничениями [17], главное из которых заключается в том, что после раскрытия всех макросов в выражении должны остаться только константы и операции. В выражении не может быть вызовов функций, поскольку это открывало бы возможность непредсказуемо долгого их вычисления или вообще зацикливания. Кроме того, функции могут иметь доступ к глобальным переменным, которые не существуют во время исполнения препроцессора. В некоторых языках программирования это ограничение более слабое – допускается вызов стандартных функций, таких как аналоги `sin`, `strlen` и т. п.

Так или иначе препроцессор языка С позволяет статически выполнять достаточно сложные вычисления, как например,

```
#define N 18
#define B(k) ((N & ~(k-1)) == 0)
#if (B(8))
#define scale unsigned char
#elseif (B(16))
#define scale unsigned short
#else
#define scale unsigned long
#end if
```

Наличие псевдомакроса `defined` делает избыточными условные директивы препроцессора вида `#ifdef`и `#ifndef`. Определенность макросов можно комбинировать как между собой, так и с другими условиями:

```
#if (defined(A) && !defined(B) || N>3)
...
#endif
```

6.5. Генерация лексем

Все рассмотренные выше команды препроцессора оперируют лишь с теми лексемами, которые так или иначе присутствовали в исходном тексте программы. Однако есть в препроцессоре языка

[17] Отметим, что становятся не вполне точными сделанные ранее утверждения о том, что синтаксис препроцессора никак не согласован с синтаксисом языка С, и о том, что препроцессор лишь манипулирует с последовательностями лексем.

С две операции, которые порождают новые лексемы. Эти операции могут использоваться лишь в теле макросов.

Унарная операция # преобразует лексему в строку, содержащую представление этой лексемы. Например,

```
#define A(x) #x
A(X1)
```

будет развернуто как

```
"X1"
```

В примере типичного использования этой операции

```
#define print(x) fprintf(stderr,"%s=%d\n",#x,x)
print(A);
```

будет преобразовано в

```
fprintf(stderr, "%s = %d\n","A", A);
```

что принципиально невозможно реализовать в языке С с помощью функции, поскольку в объектном коде имена переменных исчезают.

Бинарная инфиксная операция ## применяется к двум лексемам, сначала преобразуя их в строки, подобно операции #, а затем эти строки конкатенируются (склеиваются) и снова преобразуются в лексему. Например,

```
#define C(x,y) x##y
C(a,5)
```

преобразуется в

```
a5
```

Чаще всего операция ## используется для порождения новых имен. Следующий пример показывает, как с помощью препроцессора можно реализовать суррогатные родовые (generic) типы. Пусть, например, мы хотим определить тип односвязного списка,

но так, чтобы у этого типа был параметр – тип элементов. Следующие макроопределения

```
#define List(type) _##type##List
#define DeclareList(type) \
typedef struct __##type##List {\
        type value;\
        struct __##type##List * next;\
} * List(type)
```

позволяют нам далее объявлять тип списка с целыми компонентами, не повторяя описание структуры:

```
DeclareList(int);
```

что разворачивается препроцессором в

```
typedef struct __intList
{
  int value;
  struct __intList * next;
} * _typeList;
```

а новые переменные этого типа описывать как

```
List(int) a;
```

Отметим, что это весьма ограниченное и небезопасное решение. Например,

```
List(long int) x;
```

раскроется в синтаксически неправильную конструкцию

```
_long intList a;
```

Таким образом, эти возможности можно использовать с большой осторожностью либо от безысходности, либо при автоматической генерации C-программ. Современные языки программирования, включая C++, имеют более развитые средства метапрограммирования.

7. ОБЪЕКТЫ И ТИПЫ

Изучение конструкций языка программирования начнем с понятия именования. Возможность дать имя некоторой конструкции, определенной в программе – функции, типу данных, переменной и т. п., – можно рассматривать как средство повышения уровня языка методом абстракции: использование имени объекта вместо него самого позволяет отвлечься от деталей его реализации. Здесь под объектами мы будем понимать именно синтаксические конструкции, а не те объекты, с которыми программа манипулирует в процессе исполнения.

7.1. Области видимости

Поскольку количество используемых программой имен может быть очень большим, то во избежание путаницы требуются средства ограничения видимости. Проводя жизненную аналогию, при упоминании имени Николай мы хотим, чтобы оно означало знакомого нам человека, а не одного из тысяч других Николаев. Но при этом иногда у нас может быть несколько знакомых Николаев и в этом случае требуются какие-то уточнения, а в определенном контексте хотелось бы упомянуть царя Николая I или Святого Николая. В языках программирования для ограничения области видимости используются следующие механизмы.

Блочная структура – иерархия областей, называемых *блоками*, содержащих определения объектов. Блоки обычно связаны с синтаксическими конструкциями, такими как определения функций или типов данных, модулями, структурными операторами и т. п., но не обязательно в точности с ними совпадают. Блочная структура задает *правило видимости*: имя, определенное в некотором блоке, может быть использовано в нем самом и всех вложенных блоках за исключением тех, внутри которых имеется другое определение того же самого имени. Про такие блоки говорят, что они создают «дыру» в области видимости данного имени. Для того чтобы это правило работало, необходима *однозначность*, т. е. требование того, что в блоке не было нескольких определений одного и того же имени. Правило видимости определяет метод поиска определения имени: если имя не определено в том же блоке, где оно использовано, то оно

ищется в охватывающем блоке и т. д. Пример, блочной структуры показан ниже:

```
float power
(float x, int n)

    float s = 0;
    for (int k = 0; k <n; k++)
    {
         int ss = s;
         s *= x;
         printf("%f * %f = %f\n", ss, x, s);
    }
    return s;
}
```

Заметим, что имя функции `power` относится не к тому же блоку, что ее параметры.

Хотя правило видимости и позволяет переопределять во вложенном блоке имя, определенное в охватывающем, такая практика считается вредной, поскольку она зачастую приводит к трудно обнаруживаемым ошибкам. Например, в языке С вполне допустимо следующее определение:

```
int x(int x)
{
    for (int x=0;x<10;x++)
     {
        int x=15;
        ...
      }
}
```

Поэтому переопределение имен во вложенных блоках запрещается в некоторых языках программирования, например, в С#.

Имя, определенное в некотором блоке, может быть использовано и вне этого блока, если оно *квалифицировано*, т. е. в месте использования указано, в каком блоке его надо искать. Наиболее типичными примерами являются использование полей структурных объектов:

```
struct { int x, y; } R, S;
...
R.x = S.y;
```

В некоторых языках программирования, таких как Паскаль и Visual Basic, имеются *присоединяющие операторы*, которые являются блоками, делающими доступными в данной точке программы множество имен из блока, определяющего некоторую структуру, что позволяет избежать многократную квалификацию имен полей. Например,

```
R.x = R.x > R.y ? R.x - R.y : R.y - R.x;
```

в C-подобном синтаксисе может выглядеть так:

```
with (R){ x = x>y ? x-y : y-x; }
```

Отметим, что вложенные присоединяющие операторы не помогают в случае однотипных структур. Например, в операторе

```
R.x = R.x > S.y ? R.x - S.y : R.y - S.x;
```

с помощью присоединяющих операторов можно избавиться от квалификации либо `R`, либо `S`, но не обеих вместе.

Отдельно следует рассмотреть вопрос об использовании имен, определенных во внешних библиотеках. Язык C использует для этой цели препроцессор и включаемые файлы. Например, если для инициализации библиотеки `library1` используется функция `Initialize`, то в соответствующем включаемом файле `library1.h` должна быть строка

```
extern void Initialize();
```

и мы можем использовать это имя в своей программе:

```
#include "library1.h"
...
Initialize();
```

Функция `Initialize` из библиотеки `library1.lib` затем подключится к программе во время работы редактора связей.

Пусть теперь нам потребовалось использовать две библиотеки – `library1` и `library2`, и в каждой их них есть своя функция `Initialize`.

```
#include "library1.h"
#include "library2.h"
...
Initialize();
```

Поскольку обе `Initialize` имеют одинаковую спецификацию, то на уровне трансляции ошибок не произойдет, но у редактора связей возникнет конфликт. Если спецификации функций `Initialize` будут различаться, то ошибку выдаст транслятор. Если же библиотеки определяют один и тот же макрос, то о конфликте предупредит препроцессор.

Конечно, хорошо, что конфликт так или иначе будет обнаружен до исполнения программы, пусть даже без очевидного объяснения причины. Однако у нас не остается никакой возможности использовать эти библиотеки одновременно. Поэтому создателям библиотек на языке C рекомендуется выдумывать уникальные имена для всех объектов, которые можно использовать извне. Например, в нашем случае создатели первой библиотеки должны были назвать функцию `library1Initialize`, а второй – `library2Initialize`. Очевидно, что это, во-первых, отрицательно отражается на читаемости программ и, во-вторых, этот совет бесполезен, если мы не можем повлиять на разработчиков библиотек.

Соглашения об именовании зачастую носят характер обязательной рекомендации: «можно, но категорически нежелательно». Например, «обычным» программистам не рекомендуется начинать идентификатор с двойного подчеркивания, поскольку так именуются системные объекты или макросы: `__FILE__`, `__TIME__` и т. п.

В некоторых языках программирования понятие библиотеки и ее использования выносится на уровень языка. Специальные конструкции реализуют *импорт* библиотеки. В случае возникновения конфликта при использовании имени, определенного в разных библиотеках, об этом может внятно сообщить транслятор, а программист – воспользоваться явной квалификацией, подобной той, которая используется для структур:

```
System.Drawing.Color.Aquamarine
```

Некоторые языки допускают исключения из правила однозначности, позволяя в одном блоке объектам разного сорта иметь одинаковые имена, если они синтаксически не могут появляться

в одном и том же контексте. Например, в некоторых диалектах языка SQL можно написать следующий запрос:

```
select select.select
from select, where
where where.select=select.where;
```

который однозначно трактуется, поскольку

1. запрос должен начинаться с ключевого слова – `select`, `update`, `delete` и т. д., и поэтому здесь не ожидается имя таблицы или столбца таблицы;
2. после ключевого слова `select` должно идти выражение, частным случаем которого является `select.select`, причем здесь перед точкой может быть только имя таблицы, а после – только имя столбца `select`, который должен быть определен в таблице `select`;
3. после ключевого слова `from` должно идти имя таблицы и не может появляться имя колонки и т. д.

7.2. Типы данных

Перейдем теперь к объектам, которые используются во время исполнения программы, т. е. обрабатываемым данным. Доступ к ним осуществляется через имена, определенные в программе – константы, переменные, параметры функций и т. п. Связь между подобной синтаксической конструкцией и объектом может быть неоднозначной: например, в различные моменты исполнения одной и той же переменной могут соответствовать разные объекты, а может не соответствовать ничего. И наоборот, двум разным именам может соответствовать один и тот же объект. Более того, количество объектов времени исполнения может быть существенно больше, нежели количество определенных в программе имен. Таким образом, существуют *анонимные объекты*, не имеющие собственного имени, доступ к которым осуществляется только через имена других объектов. Типичным примером являются элементы массивов или указуемые переменные.

Язык программирования предоставляет *систему типов данных*, которая может зависеть от уровня языка и его ориентации на конкретную область приложения. Для универсального языка программирования, такого как C, типы тоже носят универсальный характер, что приводит к необходимости решения двух задач:

во-первых, отобразить типы данных предметной области в те типы и операции, которые предоставляет язык программирования, и, во-вторых, реализовать типы данных в терминах команд и данных машины – битов, байтов и т. п. Таким образом, при рассмотрении любого типа данных мы должны осветить следующие аспекты:

- моделируемая категория – то, для представления чего этот тип предназначен (например, неотрицательные целые числа);
- синтаксис – способ записи типа данных в программе (например, `unsigned int`);
- литеральные значения – способ записи констант этого типа в тексте программы (например, `0x123`);
- набор операций, которые получают в качестве аргументов или выдают в качестве результатов значения данного типа (например, `+, -, *`);
- реализация – способ отображения значения данного типа в машинные данные.

Заметим, что в зависимости от уровня рассмотрения не все из перечисленных аспектов представляют интерес. Так, теория *абстрактных типов данных* отождествляет тип с набором операций, работающих со значениями этого типа, и аксиомами, которые задают свойства операций. При этом даже представление констант может быть сведено к нульместным операциям. С другой стороны, при описании системы типов данных в языках высокого уровня могут оставаться открытыми вопросы реализации.

Нетривиальным является вопрос об эквивалентности типов данных. Достаточно ли, например, чтобы два типа данных реализовывали одинаковый набор операций или следует потребовать, чтобы реализация была одинакова? В некоторых языках на эквивалентность типов влияет то, какие им даны имена (именная эквивалентность). Это представляется разумным, например, в следующем случае:

```
typedef int Apples; /* количество */
typedef int Distance; /* в километрах */
typedef int LocalDistance; /* в метрах */
```

чтобы не позволять складывать километры с яблоками или перемножать метры на километры. Развивая далее эти соображения, хотелось бы сделать систему типов достаточно выразительной, чтобы

сформулировать, например, что скорость, помноженная на время, дает расстояние.

Но даже если рассматривать только структурную сторону вопроса, т. е. считать, что типы эквивалентны, если они одинаковым образом составлены из одинаковых базовых типов, то проверка этого свойства является сложной алгоритмической проблемой при наличии рекурсивных типов. Например, требуется выяснить, являются ли эквивалентными типы `T1`, `T2` и `T3` в следующем примере:

```
typedef struct S1{int x; struct S2 * next; } *T1;
typedef struct S2{int x; struct S1 * next; } *T2;
typedef struct S3{int x; struct S3 * next; } *T3;
```

Эта проблема решена всего лишь около двадцати лет назад доказательством разрешимости проблемы распознавания эквивалентности детерминированных магазинных автоматов.

7.2.1. Анализ типов

Естественно, что операции должны применяться только к аргументам соответствующего типа. Например, не имеют смысла следующие применения операций/функций:

```
"При" / "вет"
M[1.2]
sin("Привет")
1.2 % 3.4
```

и, значит, необходим *анализ* (или *контроль*) *типов*, который бы гарантировал правильность применения операций в смысле соответствия типов.

Ситуация осложняется тем, что выполнение некоторых операций может существенно отличаться в зависимости от типов аргументов. Правильнее будет сказать, что разные операции из разных типов данных могут обозначаться одинаково. Такая зависимость называется *перегрузкой* операций. Пожалуй, наиболее распространенной перегруженной операцией является присваивание. Если семантика присваивания сводится к пересылке (копированию) данных из одного места в другое, то необходимо знать размер копируемой области, определяемой типами источника и получателя присваивания. Возможно также, что при присваивании происходят нетривиальные преобразования значений из одного типа в другой. Другим распространен-

ным примером перегруженной операции является операция сложения (+), которая бывает определена для разных видов чисел (например, целых и действительных), строк, указателей и др.:

```
1 + 2
1.2 + 3.4
"При" + "вет"
p + 7
```

где p – указатель на объект некоторого типа.

Если речь идет об определенных в программе функциях и операциях, то перегрузка называется *полиморфизмом*. Это позволяет использовать одно и то же имя для разных функций, выполняющих по существу одно и то же действие [18], но для разных способов задания аргументов. Например, в языке C# можно определить три функции рисования прямоугольника:

```
void DrawRectangle(int x, int y, int w, int h);
void DrawRectangle(Location p, Size s);
void DrawRectangle(Rectangle r);
```

Полиморфизм возникает также, если в языке есть понятие *подтипа*, который может *переопределять* некоторые операции родительского типа. Обычно такая возможность появляется в объектно-ориентированных языках программирования, где понятие подтипа реализуется как наследуемый класс. Однако и в языке C можно заметить эти свойства, если рассмотреть множество различных целых (или вещественных) типов данных: char, int, long и т. д.

Перегрузка и полиморфизм существенно повышают понимаемость программ. Той же цели служит *неявное приведение типов*. Например, в случае, когда первым аргументом сложения является целое, а вторым – вещественное число, как в случае

```
1 + 2.0
```

то исполнитель решает, что нужно выбрать операцию сложения вещественных чисел, предварительно преобразовав первый аргумент из целого в вещественное.

[18] Конечно, это только естественное предположение. На самом деле может оказаться, что одноименные функции делают совершенно разные вещи.

Важной характеристикой языка программирования является то, когда именно выполняется анализ типов – во время трансляции или во время исполнения. В первом случае говорят о *статическом*, а во втором – о *динамическом* контроле типов. Существуют и языки, в которых часть контроля типов осуществляется статически, а часть – динамически. Следует отметить, что не существует реальных языков программирования вообще без контроля типов, хотя иногда (неправильно) так говорят, если в языке контроль типов полностью динамический. Даже машинный язык является типизированным. Например, если адрес представляется машинным словом, то перед выполнением команды, извлекающей данные по этому адресу, необходимо проверить, что адрес правильный, поскольку не каждое машинное слово является адресом.

При динамической типизации одна и та же переменная может в разные моменты исполнения хранить значения разных типов. К «достоинствам» динамической типизации можно отнести следующие:

- становится необязательным описание переменных, что особенно «нравится» непрофессиональным программистам;
- если в программе требуются вспомогательные переменные, то можно «сэкономить» их количество, не заводя свои переменные для каждого типа и т. п.

Рассмотрим, следующий пример на языке Visual Basic:

```
If t > 0
   x = 1
   y = 2
ElseIf t < 0
   x = "1"
   y = 2
Else
   x = "1"
   y = "2"
End If
Print x + "+" + y + "=" + (x + y)
```

Будем считать, что в языке Basic есть соглашение, что если одним из аргументов операции + является строка, то второй аргумент тоже необходимо преобразовать в строку. Это представляется

естественным, если обратить внимание на использование + в операторе Print. Тогда программа при t>0 будет печатать текст

```
1+2=3
```

а в остальных случаях (если, конечно, t будет числом, а иначе до печати вообще дело не дойдет):

```
1+2=12
```

Таким образом, динамическая типизация относит обнаружение ошибок несоответствия типов на время исполнения, что во многих случаях противоречит принципу раннего обнаружения ошибок и ведет к написанию ненадежных программ. Удобство и экономия усилий, которые дает динамическая типизация, сводятся на нет усилиями, которые позже потребуются при отладке.

Справедливости ради надо сказать, что бывают ситуации, когда динамическая типизация отражает суть задачи:

- при создании универсальных программ, например, интерпретатора, необходимы переменные, которые хранят значения переменных интерпретируемой программы. А поскольку типы этих значений заранее неизвестны, то переменная в интерпретаторе должна быть некоего универсального типа, и все проверки соответствия будут проводиться во время исполнения;
- если в программе имеется переменная, которая может принимать значения одного из подтипов типа этой переменной, то в процессе выполнения придется выяснять, не переопределена ли некоторая операция для подтипа;
- новые типы могут появляться в процессе выполнения программы, например, при динамической загрузке объектных модулей или динамической генерации кода программы и т. п.

Статическая типизация, напротив, нацелена на то, чтобы выявить ошибки несоответствия типов как можно раньше. В определенной степени можно считать, что статический анализ типов сродни верификации, если явные указания типов объектов рассматривать как дополнительные условия, которые необходимо доказать или опровергнуть. Помимо этого, статическая типизация обеспечивает лучшее понимание программ. Одним из способов обеспечения этого является *строгая* типизация, при которой:

- для каждой переменной или поля структуры указан тип;

- для операций, функций и процедур указаны типы аргументов и результатов;
- все приведения типов должны быть явными и т. д.

Некоторые языки программирования смягчают эти требования, при условии, что они не противоречат статичности типизации: если типы объектов так или иначе могут быть выведены из контекста их использования. Это обеспечивает возможность проверки корректности применения всех операций.

7.2.2. Классификация типов

Перейдем к рассмотрению наиболее распространенных типов данных. Для того чтобы сделать рассуждения о типах более лаконичными, введем их классификацию: мы сможем определять множество свойств типа просто отнесением его к некоторому классу.

Прежде всего все типы можно разбить на *предопределенные*, т. е. предоставляемые самим языком программирования, и *определяемые*, т. е. описанные в программе.

С другой стороны, все типы можно разбить на *простые*, т. е. неделимые с точки зрения языка, и *структурированные* – предназначенные для агрегации компонентов.

Все типы (по крайней мере в тех языках, которые мы будем рассматривать) имеют операции присваивания, сравнение на равенство и неравенство.

Если, кроме этого, тип предоставляет операции, связанные с линейным порядком (<, <=, >, >=), то тип называется *упорядоченным*.

Перечислимым (или *интегральным*) называется тип, множество значений которого отображается в диапазон целых чисел. Любой перечислимый тип, естественно, является упорядоченным.

Арифметическим называется упорядоченный тип, предназначенный для работы с числами, т. е. предоставляющий арифметические операции сложения, умножения, деления и т. п.

Поскольку язык С в силу своей машинной ориентации не делает различия между некоторыми существенно разными предопределенными типами, считая наиболее важным то, сколько места в памяти они занимают, мы начнем с рассмотрения базовых типов на примере языка Паскаль и обсуждения вариаций на тему того, какими эти типы могли бы быть или бывают в других языках программирования.

7.2.3. Логические типы

Логический тип предназначен для представления булевых значений. Традиционно тип обозначается в языке Паскаль зарезервированным словом `boolean`. Литеральными константами являются `true` и `false`. Над логическим определены обычные операции: `and` – конъюнкция, `or` – дизъюнкция, `not` – отрицание, `xor` – взаимное исключение, которое по существу совпадает с равенством. Поскольку тип является перечислимым (`true`=1, `false`=0), то для него определены отношения порядка (`true>false`), которые можно трактовать как импликацию, например,

```
a >= b
```

означает, что из `a` следует `b`.

Вариации

Для представления значения логического типа достаточно одного бита, поскольку в этом типе всего два значения. Потенциально с помощью одного байта можно было бы представить 8 логических значений. Однако здесь возникает противоречие между компактностью представления и временем доступа – выборка одного бита из байта или слова может потребовать нескольких дополнительных команд. Поэтому обычно на представление логического типа отводится по крайней мере один байт, а вопросы упаковки решаются отдельно.

7.2.4. Символы

Символьный тип предназначен для представления букв, цифр и других символов, появляющихся в текстах на компьютерных и естественных языках. Тип обозначается в языке Паскаль словом `char`. Литеральные символьные константы представляются заключенными в апострофы (одинарные кавычки) – `'A'`, `'1'`, `'*'`, `'''`. Тип является интегральным (а значит и упорядоченным) – взаимно однозначное соответствие с диапазоном 0..255 реализуется стандартными операциями `chr` и `ord`.

Вариации

Множество представимых символов и их коды описаны стандартом ASCII – американским стандартным кодом для обмена информацией. Изначально кодировка была семибитной (128 символов),

достаточной для представления букв, цифр, специальных и управляющих символов. К последним относятся такие символы, как перевод строки, табуляция, возврат на один символ назад, перевод страницы и др. Структура кода позволяет эффективно с помощью битовых операций проверять, является ли символ буквой или цифрой и т. п. Проблемы начались с созданием национальных версий, когда потребовалось добавить новые символы, скажем, из французского или шведского алфавита. Для этого предлагалось поставить их на место редко используемых символов, таких как `@`, `{,}`. Однако для алфавитов, которые вводят большое количество новых символов – кириллицы, греческого, иврита и др. – приходится расширить диапазон до 256.

Если же добавляемый алфавит совсем большой, как, например, японская катакана, то по крайней мере некоторые символы будут представляться двумя байтами. Чтобы при этом «обычные» английские тексты представлялись как и раньше, предлагалось использовать управляющие символы SI (shift-in) и SO (shift-out) для временного перехода к двухбайтовой кодировке и обратно.

Но и это не является полным решением проблемы, если в тексте содержатся символы из нескольких алфавитов. Необходима кодировка, которая включает символы из всех алфавитов одновременно. Принципы организации такой кодировки заложены в Unicode, которая в настоящий момент перечисляет более 100 тысяч символов [19]. Собственно, представление этих символов может быть различным и определяться форматом записи. Например, UTF-32 отводит на каждый символ ровно по 4 байта, а UTF-8 – от одного до четырех байтов, но при этом первые 128 кодов совпадают с ASCII, что делает UTF-8 предпочтительнее с точки зрения обратной совместимости.

Некоторые современные языки программирования предоставляют специальные символьные типы для различных кодировок, другие – изначально полагают, что символьный тип моделирует Unicode.

[19] Заметим, что то, как символ изображается, является несомненно важной, но все-таки вторичной его характеристикой, и символы «латинская-прописная-о» и «кириллическая-прописная-о» – это совершенно разные символы, хотя и имеют одинаковое изображение. Также отдельно рассматриваются вопросы шрифтов, размеров и т. п.

7.2.5. Целые числа

Целый тип моделирует целые числа и обозначается в языке Паскаль словом `integer`. Запись целых чисел состоит из десятичных цифр и, возможно, минуса в начале: `1`, `-2`, `123`. Паскаль предоставляет обычный набор арифметических операций над целыми: `+`, `*`, `-`, `div`, `mod` и др. Слово `div` обозначает деление, поскольку привычная косая черта «`/`» задействована для вещественных чисел.

Вариации

Основная проблема при моделировании целых чисел состоит в том, что их бесконечно много. Независимо от того, сколько памяти мы выделим для хранения целого числа, она так или иначе будет конечна, поскольку конечна память всего компьютера. Поэтому обычно реализуются не все целые числа, а некоторый диапазон, зависящий от размера памяти, отводимого для одного числа. Например, если целые числа представляются машинным словом в 16 бит, то реализуется диапазон –32768..32767, чтобы положительных и отрицательных чисел было примерно одинаково. Однако во многих случаях наверняка известно, что число будет неотрицательным и поэтому требуется *беззнаковый целый тип*, в котором мы смогли бы с помощью тех же 16 бит представить числа в диапазоне 0..65535.

Рассмотрим более детально представление целых чисел и начнем с беззнаковых, как более простых в этом смысле. Пусть на представление числа отведено n разрядов и $b_{n-1}b_{n-2} \ldots b_0$ – значения битов, представляющих число. Если эту последовательность рассматривать как запись числа в двоичной системе счисления, то значение числа будет равно

$$\sum_{i=0}^{n-1} b_i * 2^i ,$$

а диапазон представимых чисел – $0 .. 2^n-1$.

Пусть теперь нам надо представить число со знаком. Первое, что приходит в голову – это зарезервировать старший бит b_{n-1} для знака (скажем, 0 – положительное, 1 – отрицательное). Однако при таком подходе оказывается, что 0 будет иметь два равных представления и возникнет странный вопрос о том, равны ли положительный 0 и отрицательный ноль. К тому же усложнится реализация операций. Напри-

мер, для того чтобы сделать сложение, мы должны будем проверить, одинаковы ли знаки числа, и если они разные, то делать вычитание вместо сложения, и при этом сначала определить большее из чисел по абсолютной величине и т. д. Все эти проблемы решаются так называемым *дополнительным кодом*:

$$-b_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} b_i * 2^i.$$

Например, при $n = 8$:

- 0 0000000 = 0
- 0 0111110 = 32+16+8+4+2 = 62
- 1 0000100 = –128+4 = –124
- 1 1111111 = –128+64+32+16+8+4+2+1 =–1

Следующая таблица показывает диапазоны знаковых и беззнаковых целых чисел при разной разрядности:

Разрядность	Без знака	Со знаком
8	0..255	–128..127
16	0 .. 65,535	–32,768 .. 32,767
32	0 .. 4,294,967,295	–2,147,483,648 .. 2,147,483,647

Отметим, что приведенные в этой таблице значения достаточно невелики. Такие практически значимые величины, как размер генома человека, количество байтов оперативной памяти в смартфоне, размер бюджета РФ и многие другие не укладываются в данные ограничения.

Десятичная система счисления не всегда является наиболее подходящей. Конечно, для экономических задач или научных расчетов она привычнее, но в системном программировании, где повсеместно проявляется привязка к бинарному представлению, числа удобнее записывать в системе счисления, основание которой является степенью двойки. При этом если основание системы равно 2^n, то запись в этой системе разбивает двоичное представление на группы длины n. Например, в восьмеричном числе 275 цифра 5 обозначает три младших бита, 7 – следующие три и т. д. Остается только запомнить, какие именно комбинации из трех бит кодирует каждая из цифр от 0 до 7, чтобы легко определить значение любого бита. Наибольшее распространение получили восьмеричная, шестнадцатеричная и двоичная системы.

7.2.6. Вещественные числа

Вещественный тип предназначен для представления действительных чисел и обозначается в языке Паскаль идентификатором `real`. В записи констант вещественного типа помимо целой части может быть дробная, указанная после точки, и порядок – после буквы «е», например, `1.2`, `123.456e7`, `-1e-10`. Заметим, что число `1` не будет записью вещественного числа, поскольку представляет целое. Тип является арифметическим и упорядоченным, но не перечислимым [20].

Вариации

Как и целые числа, вещественные могут быть разного размера. Наиболее распространенными способами представления вещественных чисел при заданном размере являются представления с фиксированной и с плавающей точкой. Начнем с первого, как с более простого, хотя оно не используется ни в Паскаль, ни в С.

При представлении с *фиксированной точкой* помимо разрядности n вводится еще один параметр $p<n$ – размер дробной части. Тогда в последовательности битов $b_{n-1}b_{n-2} \dots b_0$ знак числа определяется битом b_{n-1}, а части $b_{n-2}...b_p$ и $b_{p-1}...b_0$ являются двоичными представлениями целой и дробной частей числа соответственно, т. е. абсолютная часть числа равна

$$\sum_{i=0}^{n-1} b_i * 2^{i-p}.$$

Например, при n=8, p=2:

- 000000 00 = 0
- 1 00000 00 = –0
- 001111 10 = 8+4+2+1+1/2 = 15.5
- 100001 00 = –(1) = –1
- 111111 11 = –(16+8+4+2+1+1/2+1/4) = 31.75

[20] Формально говоря, последнее можно было бы оспорить, поскольку при фиксированном размере памяти, отводимом на представление вещественного числа, в нем может быть лишь конечное множество значений, которые, очевидно, можно перечислить.

Несложно заметить, что тогда максимальное по абсолютной величине число равно $(2^{n-1}-1)/(2^p)$, а минимальное ненулевое – $1/(2^p)$. Таким образом, при одинаковом размере вещественные числа с фиксированной точкой задают меньший диапазон значений, чем целые.

Представление с *плавающей точкой* дает возможность задавать как существенно большие, так и существенно меньшие по абсолютной величине числа путем разделения числа на мантиссу и порядок. Для этого число предварительно приводится к нормализованному виду, в котором точка стоит сразу за первой ненулевой цифрой, например, `123.456e7 = 1.23456e9`. До символа порядка «e» записана мантисса, а после – порядок. Нулевое значение не может быть представлено таким образом и обрабатывается как особый случай. Можно заметить, что в случае двоичных чисел перед точкой обязательно стоит цифра 1, а значит ее можно вообще не хранить. Старший бит b_{n-1} как и раньше означает знак числа. Если на порядок отводится e разрядов, а на мантиссу $m = n-e-1$, то представление числа выглядит следующим образом: биты $b_{m-1}..b_0$ обозначают мантиссу, а $b_{n-1}..b_m$ – порядок. Со знаком порядка поступают несколько хитрее – вводится еще один параметр s – смещение порядка. Разные значения смещения порядка s позволяют либо задавать большие числа, либо увеличивать точность малых чисел.

Таким образом, абсолютная величина числа определяется как

$$e*(1+\sum_{i=0}^{m-1}\frac{b_i}{2^{i+1}}),$$

где порядок

$$e=(\sum_{i=m}^{n-1}b_i*2^{i-m})-s.$$

Например, $n = 8$, $e = 3$, $s = 3$:

- 0 000 0000 = 0 (особый случай)
- 0 001 1110 = 2^{1-3} * (1+0.875) = 0.46875
- 1 000 0100 = – (2^{0-3} * (1+0.25)) = – 0.03125
- 1 111 1111 = – (2^{7-3} * (1+0.9375)) = 31

Следующая таблица показывает примерные диапазоны значений для некоторых видов вещественных чисел, определяемых стандартом IEEE 754:

Тип числа	Наименьшее положительное	Наибольшее положительное
Разрядность 32, порядок 8 бит, смещение 127	$1{,}2\times10^{-38}$	$3{,}4\times10^{+38}$
Разрядность 64, порядок 11 бит, смещение 1023	$2{,}3\times10^{-308}$	$1{,}7\times10^{+308}$

В отличие от представления с фиксированной точкой при представлении с плавающей точкой реализуемые числа распределяются неравномерно: малые по абсолютной величине числа находятся ближе друг к другу, чем большие, но относительная погрешность при этом одинакова, т. е. существует (относительно небольшое) число $\varepsilon>0$, называемое *машинным эпсилоном,* такое, что для любых представимых неотрицательных чисел a и b справедливо $1<a/b< 1+ \varepsilon$. То есть эти числа неразличимы.

Таким образом, основной проблемой реализации вещественных чисел является *потеря точности*, которая в общем случае неизбежна ввиду изложенных ранее соображений. Она проявляется при значениях как с фиксированной, так и с плавающей точкой. Например, если в десятичном представлении количество знаков после точки фиксировано равно 2, то при следующем делении может возникнуть округление:

122.55 / 2 * 2 = 122.50,

а при представлении с плавающей точкой, прибавление маленького числа к большому может потеряться:

1.0e+38 + 1.0e–45f = 1.e+38

Округление может возникнуть уже на этапе трансляции программы при преобразовании числа из десятичной системы счисления в двоичную:

1.0e–40 = 9.999946E–41

На первый взгляд, поскольку погрешность невелика, она не может вызвать больших проблем. Однако, как говорится, «компьютер может

повторить ошибку программиста тысячи раз в секунду». Следующая реальная история наглядно показывает, насколько серьезными могут быть последствия. Во время первой войны в Персидском заливе американская тактическая противоракета «Патриот» промахивается мимо иракской ракеты СКАД, которая поражает казарму американских военных. Погибло 25 солдат. При анализе причин происшедшего было установлено следующее. Таймер противоракеты работал с частотой 10 герц. Регистры бортового компьютера были 24-разрядные двоичные. При сохранении величины $[0.1]_{10}=[0.0(0011)_{\infty}]_2$ происходила ошибка округления $[0.000000095]_{10}$ секунды. У наземной станции управления запуском противоракет и таймер, и регистры были двоичные. За 100 часов непрерывной работы комплекса, прошедших от предшествующей его перезагрузки до ракетной атаки, расхождение между внутренним временем противоракеты и наземной станции составило 0.34 секунды. Скорость ракеты СКАД – 1670 м/с. Таким образом, ошибка локализации цели составила около 500 метров.

Этот инцидент демонстрирует большое количество аспектов реального программирования. Отметим пару из них. Во-первых, в такого рода управляющих системах корректное взаимодействие аппаратуры и программных средств является чрезвычайно важным, особенно в условиях распределенности вычислений и необходимости жесткой синхронизации. Во-вторых, продемонстрирована важность использования кратных систем счисления: если бы частота таймера противоракеты была подходящей степенью двойки, например, равнялась 16 герц, то проблем с округлением и накоплением ошибки не было бы, а значит, не было бы расхождения с наземной станцией.

Некоторые языки и системы программирования реализуют и числа с неограниченной точностью, при которой размер может быть сколь угодно большим, насколько позволяет вся память программы. Вычисления тогда могут проводиться без потери точности. Однако более практичным и более распространенным подходом является контроль ошибок вычислений. Этому аспекту вычислений уделяется огромное внимание в математическом моделировании, использующем вещественные числа, и мы настоятельно рекомендуем читателю обратиться к соответствующей литературе для углубления знаний в этом вопросе. Здесь мы ограничимся простым, но очень важным примером, возникающим не только в математическом моделировании, – суммированием последовательности вещественных чисел методом Кахана.

Целью суммирования методом Кахана является уменьшение ошибки численного суммирования последовательности конечной точности с плавающей запятой по сравнению с суммированием «естественным» образом. В основе метода лежит частичная компенсация накапливаемой ошибки суммирования путем введения отдельной переменной, хранящей малые ошибки по сравнению со всей суммой.

```
#define real float //тип вещественных данных

real *data;          //входная последовательность
int  N;              //длина последовательности
real y;              //скорректированный очередной элемент
real t;              //не скорректированная сумма
real c = 0.0;        //компенсатор потери точности
real sum = data[0];//искомая сумма

for(int i = 1 ; i < N ; ++i)
{
  y = data[i] - c;
  t = sum + y;
  c = (t - sum) - y;
  sum = t;
}
```

Отметим некоторые важные обстоятельства. Количество операций увеличивается, хотя они и сравнительно простые – сложение и вычитание. Что более существенно – изменение порядка операций (например, с целью оптимизации кода) невозможно. Поэтому при трансляции этого кода оптимизирующими компиляторами его оптимизацию следует отключить.

7.2.7. Множества

Тип множества предназначен для представления множества всех подмножеств некоторого перечислимого базового типа T. В языке Паскаль синтаксис типа множества имеет вид `set of T`. Для записи констант типа множества нет специального синтаксиса, поскольку он вкладывается в синтаксис выражения, позволяющий перечислить через запятую элементы множества, заключив все это перечисление в квадратные скобки: `[]`, `['H','e','l','l','o']`, `[(i+1),(i-1)]`. При этом допускается указывать диапазоны эле-

ментов `['A'..'Z', 'a'..'z']`, поскольку базовый тип должен быть перечислимым. Над множествами определены обычные операции: `+` – объединение, `*` – пересечение, – разность, `in` – проверка принадлежности элемента множеству и т. д.

Если базовый тип имеет *n* значений, то в типе соответствующего множества будет 2^n значений и для его представления потребуется по крайней мере *n* бит. Например, для значения типа `set of char` потребуется 256 бит, а для `set of integer` – 65536 бит (8192 байт). Поэтому на размер базового типа обычно накладывается ограничение – в самом «жестком» случае не больше, чем количество бит в машинном слове. Тогда машинное слово можно рассматривать как битовую шкалу, а теоретико-множественные операции свести к битовым. Например, объединение множеств реализуется побитовой дизъюнкцией. Если же допускается больший размер базового типа, то и представление множества состоит из нескольких машинных слов, а для проверки принадлежности элемента множеству надо будет выяснить номер слова и номер бита в этом слове.

Хотя множества и являются совокупностью своих элементов, тип множества нельзя рассматривать как структурированный, поскольку элементы не являются самостоятельными объектами, значения которых можно изменять.

Вариации

Язык программирования SETL предоставляет существенно более выразительные средства для формирования множеств, что позволяет спрятать в них рутинную поэлементную обработку. Например, следующий оператор печатает все простые числа, не превосходящие N:

```
print({n in {2..N} | forall m in {2..n - 1} | n mod m > 0});
```

Очевидно, что большая часть этого оператора – не что иное, как определение простого числа.

7.2.8. Перечисления

Тип перечисления предназначен для представления конечного множества значений, у каждого из которых есть свое имя. Единственное, что требуется от этих элементов – возможность сравнивать их на равенство и неравенство. Тип перечисления задается переч-

нем определяемых значений. Например, в языке Паскаль тип, представляющий цвета светофора, задается как

```
(red, yellow, green)
```

а дни недели –

```
(Mon, Tue, Wen, Thu, Fri, Sat, Sun)
```

Естественно, что тип перечисления является перечислимым: линейный порядок на значениях определяется тем, в котором они появляются в определении типа.

Вариации

По существу, тип перечисления реализует возможность задать несколько различных констант, не указывая их конкретные значения. В противном случае можно было бы просто определить константы, как в языке С:

```
#define red 0
#define yellow 1
#define green 2
```

Однако если бы мы после этого решили добавить константу `orange` между `red` и `yellow`, то нам пришлось бы не забыть дать новые значения и другим константам.

Поскольку тип перечисления вводит новые имена, возникает вопрос об их области видимости. В языках Паскаль и С считается, что все имена элементов перечисления видны в том блоке, в котором определен сам тип.

Однако это может привести к конфликтам, когда надо определить два типа перечисления, в которых есть одно и то же имя. Например, если мы решили завести еще один тип для кодирования RGB представления цвета:

```
(red, green, blue)
```

то при упоминании в программе имени `green` неясно, к какому из двух типов перечисления он относится. В других языках полагается, что тип перечисления вводит новый блок, а использование значения должно быть квалифицировано именем типа.

Поскольку тип перечисления является упорядоченным, многие языки программирования дают возможность получить по значению

следующее и предыдущее, как, например, Succ и Prev в языке Паскаль. Однако, во-первых, они не всегда осмысленны, как в случае RGB, а иногда дают не тот результат, который мы ожидаем: за воскресеньем Sun должен следовать понедельник Mon.

7.2.9. Структуры

Тип структуры (или записи) предназначен для объединения в одно целое нескольких разнотипных элементов, называемых полями. Формально говоря, если заданы типы $T_1, ..., T_n$ со множествами значений $V_1, ..., V_n$, то тип структуры реализует декартово произведение $V_1 \times ... \times V_n$. Типичным примером является запись информации о человеке, которая включает, скажем, год рождения, пол, фамилию, имя и отчество. В языке Паскаль тип данных, описывающий такую информацию, может иметь вид

```
record
    gender    : (male, female);
    birth_year: integer;
    name      : string;
    surname   : string
end
```

Тип записи структурированный, и основной его операцией является выборка поля, изображаемая точкой, за которой следует имя поля, т. е. если переменная p имеет приведенный выше тип, то результатом выражения

```
2020 - p.birth_year - 1
```

будет количество полных лет человека на начало 2020 года, а смена фамилии человека может быть реализована присваиванием полю surname:

```
p.surname := 'Сидорова';
```

Для представления значений такого типа отводится участок памяти, достаточный для хранения всех полей. Пусть, например, размер поля gender равен 1 байту, birth_year – 4 байтам, а name и surname – по 256 байтов. Тогда для хранения всей структуры достаточно 1 + 4 + 2*256 = 517 байтов. Однако для эффективного доступа к полям структуры необходимо, чтобы все поля базового типа располагались со смещением, кратным их размеру. Этот метод назы-

вается *выравниванием*. В приведенном выше примере перед полем `birth_year` необходимо зарезервировать 3 неиспользуемых байта.

Без выравнивания:

gender	birth year	name	...

С выравниванием:

gender				birth_year	name	...

Заметим, что выравнивание зависит от порядка полей в записи. С содержательной точки зрения он может быть произвольным, и транслятор может переупорядочить их так, чтобы потери на выравнивание были бы минимальны. Однако это сделает невозможным выполнение низкоуровневых операций, основанных на произвольном доступе к памяти, и поэтому такие языки, как Паскаль и С считают порядок фиксированным.

7.2.10. Объединения

Если тип структуры реализует декартово произведение, то тип объединения моделирует тегированное или дизъюнктное объединение множеств. Для типов T_1, ..., T_n со множествами значений V_1, ..., V_n множеством значений типа объединения будет $V_1 \cup ... \cup V_n$. Для каждого элемента в таком объединении известно также, к какому из исходных множеств он принадлежал. Корректно с точки зрения типового контроля объединение реализовано, например, в языке Алгол-68, где можно определить тип

```
mode node = union(real, int, compl, string);
```

значениями которого может быть либо вещественное число, либо целое, либо комплексное, либо строка. Так, можно определить переменную `n` такого типа и присвоить ей строковое значение:

```
node n := "1234";
```

Однако непосредственно получить обратно присвоенное значение не удастся, поскольку мы должны сначала разобраться, значение какого типа в настоящий момент хранится в переменной `n`.

Это делается с помощью оператора case, каждая альтернатива которой дает этой же переменной новое имя, но уже определенного типа, и значение получается с использованием этого имени:

```
case n in
  (real r): print(("real:", r)),
  (int i): print(("int:", i)),
  (compl c): print(("compl:", c)),
  (string s): print(("string:", s))
esac
```

Другой подход к пониманию типа объединения исходит от *реализации*: переменная типа объединения просто предоставляет место, достаточное для хранения любого из значений объединяемых типов. Например, в языке Паскаль для этого используются так называемые *вариантные записи*. Тот же тип, что и выше, может быть описан как

```
record
case tag : (tagInteger, tagReal, tagCompl, tagString) of
     tagInteger: (i : integer);
     tagReal   : (r : real);
     tagCompl  : (re, im : real);
     tagString : (s : string)
end;
```

Здесь программисту предоставляется возможность самому определить поле tag, хранящее указание на то, какая из альтернатив является актуальной, и явно указать имена полей, через которые можно получить значения разных типов. Язык не гарантирует корректного использования объединения. Так, если переменная U имеет описанный выше тип, то вполне законно будет сначала присвоить в нее строковое значение, а затем считать его как вещественное число

```
U.s := "abc";
write(U.r);
```

что малоосмысленно сформирует вещественное число из байта, представляющего длину строки и ее первых трех символов.

Вообще говоря, даже выбирающее поле является необязательным, если нужный тип определяется по каким-то внешним соображениям, либо задача состояла именно в том, чтобы «наложить»

разнотипные значения одно на другое, что реализуется довольно странной конструкцией:

```
record case of
    0 : (i : integer);
    1 : (r : real);
    2 : (re, im : real);
    3 : (s : string)
end;
```

Очевидно, что это является дырой в контроле типов и может приводить к весьма неприятным последствиям.

7.2.11. Указатели

Указатели предназначены для моделирования понятия ссылки, с помощью которой можно «добраться» до некоторого объекта. Продемонстрируем это понятие на примере визитной карточки: на ней записаны фамилия, имя и отчество некоторого человека, скажем, Петров Николай Сергеевич, и номер его телефона. Естественно, тот факт, что карточка лежит у нас в кармане, не означает, что там находится сам Николай Сергеевич, но информации на карточке достаточно, чтобы эффективно и корректно обратиться к нему. Понятно, что такая визитная карточка может быть не только у нас и Николай Сергеевич может не знать, у кого есть его визитные карточки. Может случиться, что визитная карточка перестала соответствовать действительности, например, когда Николай Сергеевич сменил телефон или, увы, умер. С другой стороны, мы можем на той же карточке все зачеркнуть и написать информацию о другом человеке, что не будет означать, что Николай Сергеевич исчез – просто мы не сможем добраться до него с помощью этой карточки.

Указатели бывают *типизированные* и *нетипизированные*. В отличие от нетипизированного для типизированного указателя мы заранее знаем тип объектов, на которые он указывает. В этом случае тип указателя определяется над указуемым типом и мы можем задать неограниченно большое количество разных типов указателей. Например, в языке Паскаль если `T` означает некоторый тип, то конструкция `^T` – тип указателя на объекты типа `T`. Ограничением является то, что в качестве `T` мы можем использовать только имя типа.

7.2.12. Массивы

Тип массива служит для представления занумерованной последовательности однотипных элементов. Таким образом, для массива важны по крайней мере два типа: тип *TIndex*, используемый для нумерации, и тип элементов *TElem*. Тогда описание массива в языке Паскаль имеет вид: `array[`*TIndex*`]of` *TElem*. В качестве типа индекса обычно выступают отрезки целого типа, как в случае

```
array [-100..100] of real
```

но могут быть использованы и другие перечислимые типы, как, например,

```
array[boolean] of integer
array[char] of char
array[(red, green, blue)] of boolean
```

и т. п.

Массив является структурированным типом данных, основной операцией которого является доступ к элементу массива по заданному индексу. Так, если `A` – массив типа `array[-100..100] of real`, то `A[i]` обозначает *i*-й элемент массива. Как правило, массив реализуется сплошным отрезком памяти, в котором элементы следуют один за другим:

Если элементы массива являются записями, то может потребоваться дополнительное выравнивание. Например, если запись начинается с целого числа, то необходимо обеспечить, чтобы не только первый, но и все последующие элементы массива начинались со смещения кратного размеру целого числа.

Поскольку предполагается, что массивы *статические*, т. е. количество значений в типе индекса статически известно и равно, скажем n, а для любого типа элементов все его значения имеют представления одинакового размера, скажем s, то и количество памяти, которая отводится для хранения всех элементов массива, может быть вычислено во время трансляции – $n*s$. По известному адресу a начала массива и минимальному значению в типе индекса $i_{\min}$ адрес элемента с индексом i можно получить как

$$a + s * (i - i_{\min}).$$

Насколько бы простой ни казалась данная формула, надо отдавать себе отчет, что она вычисляется для любого обращения к элементу массива.

Важным понятием, связанным с массивами, является *контроль индексов*: при обращении к элементу массива необходимо выполнить проверку того, что индекс лежит в нужных пределах, т. е.

$$i_{\min} \leq i \leq i_{\max},$$

где $i_{\max}$ – максимальное значение в типе индекса. При нарушении этого условия исполнение программы должно прерваться с диагностикой *выхода за границы индекса*. Это приводит к дополнительным накладным расходам на доступ к элементам массива [21], но исключает одну из наиболее распространенных и опасных ошибок исполнения.

Вариации

Многомерные массивы предназначены для представления матриц. В языке Паскаль допускается более одного индекса у массива, например,

```
array[1..N, 1..M] of real
```

и если переменная `A` имеет такой тип, то обращение к элементу массива имеет вид `A[i,j]`. Как и одномерный массив, многомерный массив представляется сплошным отрезком памяти:

A[1,1]	A[1,2]	...	A[1,N]	A[2,1]	A[2,2]	...	A[M,N]

и для вычисления адреса элемента с индексами i, j можно воспользоваться формулой

$$a + s * ((i - i_{\min}) * m + (j - j_{\min})),$$

где m – размерность массива по второму измерению, а $j_{\min}$ – минимальное значение второго типа индекса. Очевидно, что с точки зрения реализации такой многомерный массив эквивалентен одномерному массиву, элементами которого снова являются массивы:

```
array[1..N] of array[1..M] of real
```

[21] Конечно, транслятор может быть достаточно «умным» и не вставлять проверку там, где в этом нет необходимости. Однако в общем случае статическое определение по тексту программы того, лежит ли значение индекса в нужных границах, является алгоритмически неразрешимой проблемой.

Некоторые диалекты языка Паскаль вообще полагают эти типы эквивалентными. Однако с семантической точки зрения во втором случае элементы массива являются самостоятельными объектами, которые можно целиком изменять, передавать в качестве параметра функциям и т. п., как, например,

```
A[i] := A[i+1]
```

заменит всю i-ю строку матрицы.

Все предыдущие рассуждения можно распространить и на массивы с большим количеством индексов.

Динамические массивы необходимы в тех случаях, когда мы не можем заранее определить размер массива. В языке Паскаль динамических массивов нет и он не допускает последовательности действий вида

```
read(n);
var x : array[1..n] of real;
```

поскольку становится невозможным определить во время трансляции размер переменной `x`.

Простое решение этой проблемы заключается в том, что мы можем предположить худший случай и завести статический массив большого размера, который должен быть специфицирован при описании массива, а реальное количество элементов будет храниться в дополнительной переменной. Недостатки такого решения очевидны: во-первых, может оказаться, что в большинстве случаев используется лишь небольшое количество первых элементов, и, во-вторых, наше предположение о максимальном размере может оказаться неверным, и памяти все равно не хватит. Кроме этого, контроль индексов при таком подходе возлагается на программиста.

Таким образом, корректная реализации динамического массива представляет его в виде структуры, содержащей его длину (либо верхнюю и нижнюю границу) и ссылку на память, в которой подряд располагаются элементы массива. Тогда размер всей такой структуры снова становится статическим. Однако поскольку элементы массива чаще всего располагаются в динамической памяти (куче), то выделение памяти и доступ к элементам может быть менее эффективным.

Рассмотренный выше случай подразумевает, что размер массива не меняется после его создания. Если же мы хотим, чтобы к массиву

можно было добавлять новые элементы, то реализация становится более сложной, поскольку в предположении, что все элементы массива расположены подряд, его удлинение может привести к тому, что весь массив придется переписать на новое место. Поэтому память выделяют с запасом согласно некоторой стратегии, чтобы несколько последующих операций добавления не приводили к переполнению. Такие массивы иногда называют *подвижными*.

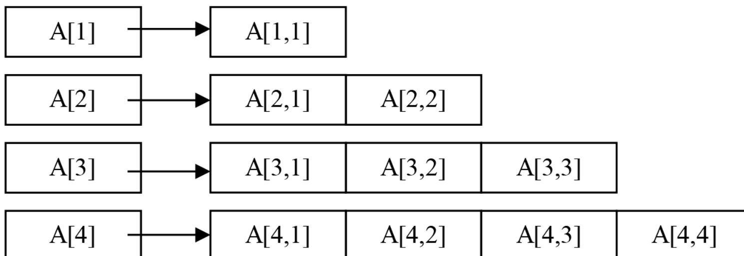
Пусть помимо добавления новых элементов в конец массива нам потребовалось вставлять или удалять элементы из середины массива. Понятно, что если настаивать на том, что все элементы массива расположены подряд, то удаление первого элемента потребует сдвига всего содержимого массива. В этом случае могут оказаться полезными списочные или древовидные структуры, которые разбивают содержимое массива на части «разумной» длины. Однако заплатить за это придется скоростью доступа к элементам массива.

Подводя итог, можно сказать, что реализация массива как последовательности элементов существенно зависит от того, какие именно операции или комбинации операций используются.

Динамические массивы имеются во многих системах программирования либо как языковые конструкции (Алгол-60, Алгол-68, Delphi), либо как часть стандартной библиотеки (Java и др.). Причем некоторые системы предоставляют несколько видов динамических массивов.

Сочетание методов реализации многомерных и динамических массивов дает возможность представления *непрямоугольных матриц*. Это может оказаться полезным, например, в вычислительных задачах линейной алгебры из соображений экономии памяти. Так, если известно, что матрица симметричная, то достаточно хранить только треугольную матрицу, сэкономив память почти в два раза.

Такую матрицу можно представить массивом из массивов, каждый из которых размещается отдельно:



Можно было бы для той же цели использовать и одномерные статические массивы, но при этом существенно усложнился бы доступ к компонентам массива. Определим массив, в котором будут храниться элементы треугольной матрицы. Если размер исходной матрицы равен N, то общее количество элементов в треугольной матрице равно $N*(N+1)/2$.

```
var B : array [1.. N*(N+1) div 2];
```

Если элементы исходной матрицы расположены в нем по строкам:

A[1,1]	A[2,1]	A[2,2]	A[3,1]	A[3,2]	A[3,3]	A[4,1]	A[4,2]	…

то элемент `A[i,j]` можно получить как `B[(i-1)*i/2+j]`.

Разнообразие особенностей массивов этим не ограничивается: можно упомянуть особые методы представления *разреженных матриц*, возникающих как в вычислительных задачах, так и, например, в дискретной математике для представления матриц смежности графов, *табличные функции*, у которых множество индексов не обязательно является целыми числами, и т. п.

Массивы, о которых шла речь выше, совмещают в себе две по существу разные операции: выделение памяти и доступ к элементам. Однако зачастую в вычислительных задачах требуется выполнение операции над подмассивом некоторого заданного массива (иногда это еще называют *вырезкой*). Для подмассива не требуется выделять собственную память, а только программно определить, как по индексу выбрать нужный элемент базового массива. Некоторые языки программирования (например, автокод Эльбрус) предлагают для этого специальные конструкции. Переводя их в паскалеподобный синтаксис, если ключевое слово `range` означает определение вырезки, то можно определить

```
var A : array [1..N, 1..N] of real;
/* транспонированная матрица */
range AT[i,j] = A[j,i];
/* первая строка матрицы */
range A1[i] = A[1,i];
/* главная диагональ */
range DiagA[i] = A[i,i];
/* побочная диагональ */
range DiagTA[i] = A[i,N-i+1];
```

Если в языке программирования заложена ориентация на вычислительные задачи, то помимо операций доступа к элементам по индексам, могут быть определены и операции линейной алгебры для массивов, представляющих векторы и матрицы. Рассмотрим пример из языка Альфа:

```
массив A[1:N,1:M], B[1:M,1:K], X, Y[1:N], Z[1:M]
вещественный C
```

т. е.

- A, B – прямоугольные матрицы размера N×M и M×K соответственно;
- X, Y – векторы длины N;
- Z – вектор длины M;
- C – вещественное число.

Тогда операции умножения и сложения понимаются следующим образом:

X*Y	Скалярное произведение
X+Y	Сумма векторов
X*C	Произведение вектора и скаляра
A*X	Произведение матрицы и вектора
Z*A	Произведение вектора и матрицы
A*B	Произведение матриц
A*C	Произведение матрицы и скаляра
A+A	Сумма матриц

С одной стороны, речь идет о перегрузке арифметических операций, которая имеется во многих языках программирования (Алгол-68, C++ и др.), что существенно повышает читаемость программ. Сравните, например,

```
(A * B) * X + (2 * Y)
```

и что-то вроде

```
sumVec(
 multMatrVec(
    multMatrMatr(A, B),
    X),
 multConstVec(2, Y))
```

С другой стороны, вынесение этих операций на уровень языка позволяет реализовать эти операции существенно надежнее и эффективнее:

- Если массивы статические, то транслятор может статически выполнить проверку соответствия размерностей, например, то, что количество строк A должно совпадать с количеством элементов X.
- Транслятор может выяснить, где нужны дополнительные массивы для хранения промежуточных результатов и их размеры. При реализации в виде функций (или операций, как в C++) мы вынуждены всегда размещать новый массив для результата, а после выполнения операции его удалять. Транслятор же может заметить, что, скажем, для присваивания

```
X := (A * B) * X + (2 * Y)
```

  промежуточные массивы вообще не нужны.
- Транслятор имеет возможность подбирать более эффективную реализацию на основе алгебраических законов или параллельных алгоритмов.

То соображение, что данные собираются в массив для того, чтобы применять массово ко всем элементам или некоторой их части одни и те же операции, доводится некоторыми языками программирования до некоего основополагающего принципа. Типичным представителем таких языков является язык APL (A Programming Language) [13]. Язык предоставляет следующие возможности:

- богатый набор операций над массивами – сдвиг, перестановка, сжатие, выбор индексов вхождений, транспонирование, упорядочение и т. п.;
- покомпонентное распространение всех операций на массивы. Таким образом, например, операция + может покомпонентно складывать не только числа, но и векторы, матрицы и т. д. Однако никакой дополнительной семантики здесь не закладывается: операция * означает именно покомпонентное перемножение, а не скалярное произведение векторов или произведение матриц;
- оператор редукции / распространяет бинарную операцию на массивы, например, `/+` и `/*` означают сумму и произведение всех элементов массива соответственно.

Эти механизмы оказываются весьма мощными, но требуют от программиста иного взгляда на решаемую задачу: вместо последо-

вательного вычисления промежуточных результатов нужно преобразовать структуру входных данных так, чтобы к ним можно было массово применять арифметические операции. Рассмотрим в качестве примера вычисление полинома степени n от x, заданного массивом коэффициентов A:

```
/+ (A * (x   ι(n+1)))
```

где стандартная операция ι порождает вектор целых чисел заданной длины, начиная с 0. Тогда

$$/+ (A * (x\ (0\ 1\ 2\ \ldots\ n)))$$
$$/+ (A * (x^0\ x^1\ x^2\ \ldots\ x^n))$$
$$/+ (A_0{*}x^0\ A_1{*}x^1\ A_2{*}x^2\ \ldots\ A_n{*}x^n)$$
$$A_0{*}x^0 + A_1{*}x^1 + A_2{*}x^2 + \ldots + A_n{*}x^n$$

Естественно, что данную программу, хотя ее и можно выполнить непосредственно в соответствии с семантикой операций, следует рассматривать как спецификацию, а эффективная реализация должна суметь не только избежать порождения промежуточных векторных значений, но и минимизировать количество умножений. К задаче вычисления полинома мы вернемся позже.

7.2.13. Строки

Строковый тип предназначен для представления последовательности символов. По существу, строка является подвижным массивом и определяется языком как отдельный тип либо ввиду своей вездесущности, либо потому, что язык не поддерживает подвижных массивов в общем виде. Так, в языке Паскаль в определении строкового типа нужно указать максимально возможную для этого типа длину. Более того, язык требует, чтобы эта длина не превосходила 255, что дает возможность реализовать ее одним байтом, а всю строку – не более, чем 256 байтами. Такое представление является весьма ограничительным с точки зрения обработки текстовой информации, поскольку не допускает длинные строки. Кроме того, реализация строк жестко привязана к однобайтовой кодировке.

Строки Паскаль поддерживают свойственный для подвижных массивов набор операций: определение текущей длины – `Length`, выборку компоненты – `s[i]`, вырезку подстроки – `Copy`, вставку – `Insert`, удаление – `Delete` и т. п.

Отличие строк от массивов заключается в особом представлении литеральных значений. В языке Паскаль для этого используются одинарные кавычки:

```
''
' '
'Hello, World!'
'A'
```

Здесь первая строка является пустой, вторая – содержит один пробел. То, что в Паскаль кавычки используются не только для строк, но и для символов, приводит к конфликту: `'A'` является как изображением строки, так и строки, и для определения типа необходимо знать контекст использования.

Одинарная кавычка, как символ внутри строки, при записи удваивается. Кроме того, допускается использование кодов символов ASCII, предваряемой символом решетки `#`:

```
''''
'A8#65#56#0#12'
```

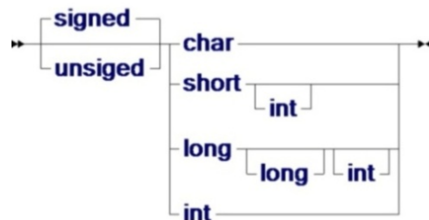
Здесь вторая строка состоит из 6 символов и заканчивается символом перевода строки, перед которым стоит символ с кодом 0, а первые два символа совпадают с последующими двумя.

7.3. Типы данных в языке С

7.3.1. Целые и символы

В языке С имеется несколько целочисленных типов, отличающихся размером и тем, допускают ли они отрицательные значения. Синтаксис целочисленного типа задается следующей диаграммой:

тип-целый:



Какие именно размеры соответствуют типам, в значительной степени зависит от конкретной реализации, что потенциально

может сказаться на переносимости программ. Следующая таблица определяет минимальные размеры для разных типов в предположении, что размер байта равен 8 бит:

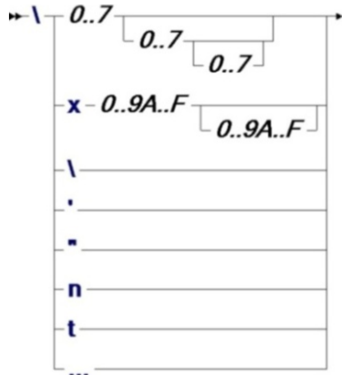
Тип	Размер (байт)	Со знаком (signed)	Без знака (unsigned)
`char`	1	[–127, +127]	[0, 255]
`short`	2	[–32767, +32767]	[0, +65535]
`long int`	4	[–2 147 483 648, +2 147 483 647]	[0, +4 294 967 295]
`long long int`	8	[–9 223 372 036 854 775 808, +9 223 372 036 854 775 807]	[0, 18 446 744 073 709 551 615]

Тип `int` без спецификатора `short` или `long` опять же в зависимости от реализации может обозначать либо короткое, либо длинное целое число.

Изображение литеральных констант задается следующей синтаксической диаграммой:

целое:

код:



Согласно этой диаграмме запись целого числа может быть основана на десятичной, восьмеричной или шестнадцатеричной системе счисления. В конце числа можно указать, является ли оно беззнаковым U, коротким H или длинным L.

Кроме того, поскольку символьный тип char тоже рассматривается как целый, и язык не делает разницы между символом и его кодом, то изображение символа в апострофах (одинарных кавычках) тоже является целым числом. Обратная косая черта \ в изображении символа используется для задания символа по его восьмеричному или шестнадцатеричному коду, либо для представления апострофа, кавычки и распространенных спецсимволов – перевода строки, табуляции и т. п.

Такой подход делает возможной символьную арифметику. Например, i-я буква латинского алфавита может быть найдена как

```
'A' + i - 1
```

что, конечно, компактнее, чем аналогичное выражение в Паскале:

```
chr(ord('A') + i – 1)
```

Однако это же делает легальными и слабоосмысленные выражения типа

```
('A' + 'B'*'2') / '3'
```

7.3.2. Логические типы

В языке С нет логического типа в явном виде. Вместо этого язык считает, что все значения, в представлении которых есть ненулевые биты, являются истинными. То же правило распространяется и на вещественные числа и указатели. Таким образом, среди целых чисел, в предположении использования дополнительного кода, единственным ложным значением является ноль.

В языке имеется несколько операций, которые работают над значениями как над логическими и выдают либо 0, либо 1

- && – конъюнкция
- || – дизъюнкция
- ! – отрицание

На самом деле конъюнкция и дизъюнкция – не совсем обычные операции, поскольку конъюнкция (дизъюнкция) в случае ложности (истинности) первого аргумента не пытается вычислять второй.

В этом смысле эти операции скорее являются разновидностью условного выполнения, о чем мы еще поговорим далее.

Примеры логических выражений:

```
!1 || 'A' && 0x12L
```

истинно, поскольку истинно `!1` и `&&` имеет больший приоритет, чем `||`, а

```
'\0' || ('A' == 'B')
```

– ложно, поскольку и `'\0'`, и `'A'=='B'` – ложны.

7.3.3. Битовые шкалы

Вместо теоретико-множественных операций язык С предлагает использовать побитовые операции над целыми числами, а точнее – над их представлениями:

- `&` – побитовая конъюнкция;
- `|` – побитовая дизъюнкция;
- `^` – побитовый XOR;
- `~` – побитовое отрицание;
- <<,>> – бинарные операции, сдвигающие битовую шкалу, представляющую первый аргумент, влево и вправо соответственно на количество разрядов, задаваемых вторым аргументом.

Например, тип `set of 0..31` языка Паскаль может быть реализован с помощью одного длинного целого числа `unsigned long int` так, что если `S1` и `S2` – множества, а `x` и `y` – числа в пределах от 0 до 31, причем `y>=x`, то

Паскаль	С
`S1 + S2`	`S1 | S2`
`S1 * S2`	`S1 & S2`
`S1 - S2`	`S1 & ~S2`
`x in S`	`(1<<x) & S`
`S1 <= S2`	`S1 & S2 == S1`
`[x..y]`	`((1<<(y-x+1))-1) << x`

Очевидно, что запись на Паскаль значительно нагляднее, хотя, возможно, язык С и обладает большей гибкостью.

Типичное использование множеств состоит в упаковке нескольких разнородных логических значений. Например, можно определить несколько характеристик человека – мужчина, вегетарианец, лысый, студент и т. п., закрепив за каждой из них некоторый бит [22]:

```
#define FLAG_MALE          1
#define FLAG_VEGETERIAN    2
#define FLAG_BALD          4
#define FLAG_STUDENT       8
```

Тогда если x – битовая шкала, представляющая набор таких характеристик для некоторого человека, то условия

```
x & (FLAG_MALE|FLAG_BALD)
x & FLAG_VEGETERIAN & FLAG_STUDENT
```

будут истинны для «мужчин или лысых» и «студентов-вегетарианцев», соответственно.

Операция сдвига зачастую используется для более эффективной реализации умножения, деления нацело и остатка от деления на степень числа 2. Пусть, например, при $m = 2^n$ и целом x:

```
x * m = x << n
x / m = x >> n
x % m = x & (m-1)
```

В случае деления нужно отдельно рассмотреть случай отрицательного x. Для того чтобы равенство имело место, необходимо обеспечить *распространение знакового бита*. Например, при восьмиразрядном знаковом целом

1 0000100 = –128+4 = –124

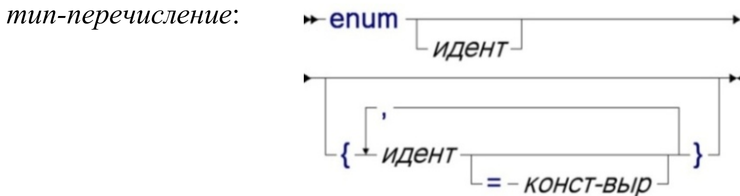
1 0000100 >> 2 = **1 11**00001 = –128 + 64 + 32 + 1 = –31 = –124/4

[22] Конечно, правильнее было бы ввести для этого соответствующий тип перечисления, а не пользоваться директивами препроцессора.

7.3.4. Перечисления

Тип перечисления в языке C служит для содержательной группировки целочисленных констант и имеет следующий синтаксис:

тип-перечисление:



Как видно, в описании элемента типа перечисления можно указать, а можно и не указывать, конкретное целочисленное значение, например,

```
enum StreetColor
{
    Red,
    Green,
    Blue
};
enum WeekDay
{
    Mon=1,
    Tue,
    Wed,
    Thu,
    Fri,
    Sat,
    Sun
};
enum PersonFlag
{
    Flag_Male        = 01,
    Flag_Vegeterian  = 02,
    Flag_Bald        = 04,
    Flag_Student     = 010
};
```

Таким образом, значения разумно задавать либо всем элементам, либо только первому, что означает начальное значение (по умолчанию равное нулю), а все последующие увеличиваются на 1.

Как уже говорилось, все элементы перечисления в языке C – просто целые числа, и поэтому будет легальным и равным 3 выражение

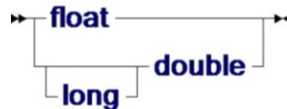
```
(Flag_Bald - Tue) | Green
```

хотя смысла в нем нет, и скорее всего такое выражение ошибочно.

7.3.5. Вещественные числа

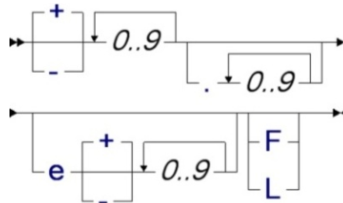
Язык C предоставляет три вещественных типа: 32-разрядный `float`, 64-разрядный `double` и `long double`, определяемых стандартом IEEE 754, о котором говорилось выше:

тип-вещественный:



Синтаксис вещественных констант задается следующей диаграммой:

вещественное:



где, подобно записи целых чисел, в конце можно указать тип константы: `F` – `float`, `L` – `double`.

Тип `long double` используется для вычислений повышенной точности и в зависимости от реализации быть 80-разрядным, либо четырехбайтным (128 разрядов), либо просто совпадать с `double`.

Заметим, что имеется лексическая неоднозначность между целыми и вещественными константами. Формально число `+99` подходит под определение вещественной константы, но будет трактоваться как целое. Кроме того, квалификатор `L` может появляться как при записи целых, так и при записи вещественных:

- `12000L` – long int
- `12e+3L` – double
- `12000.0L` – double

7.3.6. Приведение типов

Разнообразие арифметических типов приводит к необходимости преобразовывать значения одного типа в другой. Отметим, что это может быть достаточно нетривиальным действием, поскольку, скажем, представление вещественных существенно отличается от целых. Например, если арифметическая операция применяется к аргументам разных типов, то аргумент «меньшего» типа *неявно преобразуется* к большему. Для этого типы упорядочиваются по *рангу*, согласно следующему списку в порядке убывания:

- `long double`
- `double`
- `float`
- `long`
- `int`
- `char`

Заметим, что преобразование из целых в вещественные (например, `long` в `double`) может приводить к потере точности, поскольку, как мы уже отмечали, вещественные числа являются «разреженными» при больших абсолютных величинах.

Кроме этого, в зависимости от ситуации, происходит преобразование из беззнаковых целых в знаковые или наоборот. В большинстве случаев, оно определяется естественным образом, хотя в некоторых случаях может приводить к неожиданным результатам. Так, при

```
unisgned short s1 = 5;
unsigned short s2 = 10;
unsigned int i1 = 5;
unsigned int i2 = 10;
```

значение `s1-s2` будет равно –5, а значение `i1-i2` – положительным числом [23]. Подробности можно узнать в описании стандарта языка C.

---

[23] А может быть и отрицательным, если в данной реализации `int` совпадает с `short`.

Для явного преобразования типов результирующий тип указывается в скобках перед выражением, вычисляющим исходное значение. Например, при вычислении

```
(int) (3 + 0.5)
```

сначала 3 приведется к типу `float` для того, чтобы выполнить сложении с 0.5, а затем дробная часть отбросится при явном приведении, и мы снова получим целое число 3. Аналогичное (но неявное) преобразование выполняется при присваивании, инициализации и т. п., если тип источника «больше», чем тип получателя, как в

```
unsigned char x = 256;
```

где начальным значением `x` будет 0.

7.3.7. Указатели

В языке С нет отдельной синтаксической конструкции для типа указателя, который может появляться лишь в описании переменных, параметров или типов. Так, например, конструкция

```
int * p, **q, i, j;
```

описывает переменную `p` типа «указатель на целый», переменную `q` типа «указатель на указатель на целый» и две переменные `i` и `j` целого типа. Пример переменной `q` показывает, что сами указатели являются равноправными объектами, на которые можно устанавливать ссылки. Зачем это может быть нужно, мы покажем позже.

Машинное представление указателя зависит от устройства памяти. Грубо говоря, если мы занумеруем все адресуемые ячейки памяти, то реализацией указателя может быть просто целое число – номер ячейки, на которую он ссылается. Тогда если для представления указателя отводится 32 бита (4 байта), то он может указывать на 4,294,967,296 различные ячейки. Если в качестве адресуемой ячейки выступает байт, то размер адресуемой памяти будет 4 Гб.

В любом типе указателя имеется выделенное значение – *пустой указатель* `NULL`. Про него известно, что он не совпадает с адресом ни одного описанного в программе или созданного в процессе

исполнения объекта. На самом деле константа NULL определена как макрос:

```
#define NULL (void*) 0
```

и поэтому указатели можно, хотя и не рекомендуется использовать как логические значения.

Помимо присваивания, сравнения на равенство и неравенство, двумя основными операциями, связанными с указателями, являются взятие адреса (*) и разыменование (&). Рассмотрим их на следующем примере:

```
int i,j;
int *p;
p = &i;
*p = 2;
j = *p+1;
p = &j;
*p = *p+1;
```

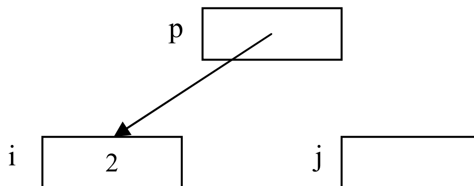
Первые две строки описывают три переменные: две целого типа и одну типа указатель на целое. Оператор

```
p = &i;
```

берет адрес переменной i и присваивает его в указатель p. Следующий оператор

```
*p = 2;
```

использует операцию разыменования для того, чтобы по указателю p «добраться» до переменной i. Таким образом, в этой точке программы *p и i означают одну и ту же ячейку памяти, которой присваивается значение 2. В результате получается следующее состояние памяти:



где стрелка, ведущая изнутри ячейки, означает, что в ней хранится адрес той ячейки, к которой ведет стрелка.

Следующий оператор

```
j = *p + 1;
```

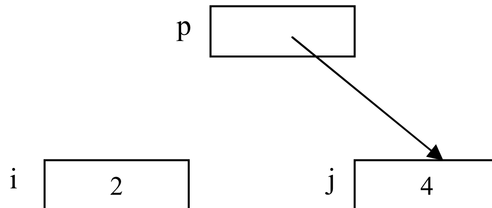
извлекает из ячейки `i` через указатель `p` значение 2, прибавляет к ней 1 и помещает результат 3 в ячейку `j`. Далее

```
p = &j;
```

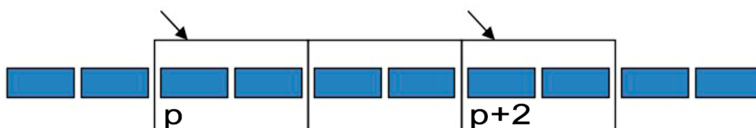
«перекидывает» стрелку из `p` на ячейку `j`: с этого места `*p` будет означать уже не `i`, а `j`. Наконец, оператор

```
*p = *p +1;
```

увеличивает значение переменной `j` на единицу, приводя к следующему состоянию памяти:



В предположении, что адреса являются целыми числами, означающими номера ячеек памяти, на указателях можно установить линейный порядок и допустить выполнение над ними арифметических операций – сложения, вычитания и сравнения. Такого рода вычисления называются в языке С *адресной арифметикой*. При этом предполагается, что указатель ссылается на один из элементов в последовательности однотипных объектов, и, например, увеличение (уменьшение) указателя на единицу дает ссылку на следующий (предыдущий) объект. Таким образом, измерения производятся не в байтах, а в объектах: если p является указателем на объект типа T, расположенного по адресу a, и k – целое (возможно отрицательное) число, то p+k будет указывать на объект типа T, расположенного по адресу a+s*k, где s – размер типа T в байтах:



Число k можно трактовать как расстояние между адресами, измеренное в объектах типа T. Аналогично к указателю можно не только прибавлять, но и вычитать такое расстояние, т. е. допустимо и p–k. Если два указателя q1 и q2 были получены из одного и того же указателя p некоторыми последовательностями операций прибавления и вычитания целых чисел, то осмысленно говорить и о сравнении этих указателей и расстоянии между ними. Так, если q1=p+5 и q2= p–5, то

- q1–q2 = 10
- q2–q1 = –10
- q1> q2 – истинно

На самом деле сравнивать можно любые два указателя, но предсказуемый результат получится только в описанной выше ситуации. Недопустимо складывать или перемножать два указателя и прибавлять к целому числу указатель.

7.3.8. Массивы

Описание массива в языке C выглядит следующим образом: если T – некоторый тип, а N – константа, то конструкция

```
T A[N];
```

отводит память под массив из N элементов типа T. Первый элемент массива имеет индекс 0, а последний – N–1. Сама переменная `A` имеет тип указателя на T, которой описание присваивает ссылку на первый (т. е. с индексом 0) элемент массива. Отличие этого описания от описания указателя

```
T * B;
```

состоит в том, что, во-первых, описание массива определяет также действия по отведению памяти и, во-вторых, переменной `A` нельзя присваивать новые значения. То есть, по существу, она является константным указателем. Во всем остальном любые указатели можно использовать и как массивы: совершенно законным будет написать `B[5]` вместо `*(B+5)`.

Сведение семантики массивов к указателям и адресной арифметике практически исключает возможность контроля индексов. Поскольку в адресной арифметике мы можем к указателю не только прибавлять, но и вычитать целые числа, то вполне корректной будет запись `A[-2]`.

С другой стороны, ни во время трансляции, ни во время исполнения не будет обнаружена ошибочность обращения `A[N]` [24].

Язык C допускает литеральные значения для массивов, но только в их инициализации: элементы массива перечисляются в фигурных скобках через запятую. При этом размер массива может определяться количеством элементов в инициализации, как например,

```
int A[] = { 5, 4, 3, 2, 1};
float A[2][2] = { {5, 4.0} , {3 , 2+2} };
```

7.3.9. Строки

Строки в языке C представляются указателем на первый символ строки. Более того, любой указатель на символ является в языке C строкой. Содержимым строки являются все символы, начиная с указываемого, вплоть до символа с кодом 0, не включая его. Длина строки нигде не хранится, и для ее определения необходимо «пробежаться» по строке, что делает эту операцию весьма неэффективной. Достоинство такого подхода состоит в том, что нет явных ограничений на длину строки, в отличие, скажем, от Паскаль.

Не следует путать строки в языке C с массивами символов, хотя мы ранее и говорили, что по существу строки являются подвижными массивами. Так, определение

```
char s[] = {'H', 'e', 'l', 'l', 'o'};
```

действительно является массивом длины 5, но не обязательно строкой длины 5, поскольку неизвестно, что следует за символом `'o'`.

Литеральные строковые значения представляются последовательностью символов (с тем же синтаксисом, что и одиночных символов), заключенной в двойные кавычки, как, например,

```
"Hello \"string"!\n"
```

Здесь за символом перевода строки `'\n'` неявно присутствует символ `'\0'`. Поэтому использование литеральной строки в описании массива

```
char s[] = "Hello";
```

[24]Хотя, казалось бы, можно было заметить, что переменная A описана именно как массив, а не просто указатель, и для нее можно было бы выполнять контроль индексов.

определит длину массива равной 6. Строки, в отличие от литеральных значений массивов, могут использоваться не только в инициализации:

```
char * s;
...
if (friendly)
    s = "Hi";
else
    s = "Hello";
```

В этом случае транслятор разместит значения констант в специальной области памяти, а присваивание просто установит значение указателя на начало одной из строк.

Сам язык С не предоставляет никаких специальных операций над строками помимо адресной арифметики. Из этого следует, в частности, что для выборки элементов строки может быть использована нотация, характерная для массивов. Например, если `s` равно `"Hello"`, то `s[0]` будет равно `'H'`, `s[4]` – `'o'`, а `s[5]` – `'\0'`.

Большое количество содержательных операций над строками предоставляется функциями стандартной библиотеки, которые описаны во включаемом файле `string.h`:

- `strlen(s)` – длина `s`
- `strcpy(s1,s2)` – копирование строки
- `strcat(s1,s2)` – конкатенация строк
- `strchr(s,c)` – указатель на первое вхождение `c` в `s`
- и т. п.

Все эти операции не выполняют никаких действий по размещению строк-результатов: предполагается, что память для этого выделена отдельно. Например, `strcpy` предполагает, что `s1` уже указывает на участок памяти, длина которого по крайней мере на 1 больше, чем длина строки `s2`, чтобы скопировать содержимое последней и дописать в конце `'\0'`. Аналогично `strcat` предполагает, что непосредственно за `s1` достаточно места для дописывания содержимого `s2`. Укоротить строку можно, присвоив в ее середину символ `'\0'`, как в случае

```
s = "Hello";
s[2] = '\0';
```

где значением `s` станет строка `"He"`.

Такая открытость и гибкость позволяет при необходимости определить и другие функции. Например, аналог функции `Copy` языка Паскаль, «вырезающий» из строки подстроку, можно реализовать следующим образом [25]:

```
char * PasCopy(char *source, int i, int l)
{
  char *dest = (unsigned char *)malloc(l+1);
  char *d = dest;
  char *s = &(source[i]);
  while ((*d++ = *s++) && l--)
    ;
  d[-1] = '\0';
  return dest;
}
```

Так же как и для массивов, язык С не обеспечивает контроля индексов для строк. Ситуация усугубляется тем, что символ `'\0'` имеет выделенное значение. Формально символ с кодом 0 разрешается и внутри литерального значения. Поэтому легально присваивание

```
s = "Hell\0o"
```

в результате которого значением s станет строка `"Hell"`.

7.3.10. Нетипизированные указатели и sizeof

Мы уже заметили, что пустой указатель `NULL` принадлежит любому типу указателя. Аналогично и некоторые операции с указателями могут по существу не зависеть от указуемого типа. Например, операция выделения памяти, которая размещает непрерывный кусок памяти и выдает указатель на его начало. Сама процедура выделения не знает, сколько значений и какого типа будут размещены в отведенной памяти, поскольку ее параметром является просто целое число, указывающее, сколько байт надо выделить. Поэтому возвращаемый указатель должен быть некоего универсального типа, или, иными словами, нетипизированным. Аналогичной является операция

[25] Этот пример демонстрирует в основном то, каким образом можно манипулировать указателями. Подробности, касающиеся управляющих структур и динамического размещения памяти (malloc) будут рассмотрены далее.

копирования одного отрезка памяти в другой, для чего требуется знать только указатели на начала отрезков и размер копируемого отрезка. В языке С такие указатели описываются с помощью псевдотипа `void`:

```
extern void * malloc(int c);
extern void * memcpy(void * dest, void * source, int count);
```

Для того чтобы этот указатель можно было типизировать, применяется приведение типов, как например, в

```
float * A =(float *) malloc(10);
```

В обратную сторону явное приведение не требуется: все указатели в случае необходимости автоматически приводятся к типу `void*`.

Основным достоинством такого подхода к реализации универсальных операций над указателями является его гибкость: программист может, пользуясь средствами языка, описать свои операции. Однако за гибкость, как часто бывает, приходится платить потерей надежности. Так, в последнем примере размер выделяемой памяти никак не связан с размером типа `float`, даже если предположить, что выделяется память не на один элемент, а на несколько. Но ни при трансляции, ни при выполнении программы проконтролировать это невозможно.

Таким образом, для обеспечения надежности и переносимости программ требуется передать подобной универсальной процедуре знание о типе аргументов. Для этой цели в языке С используется псевдофункция `sizeof`, которая возвращает размер типа аргумента. Следующая таблица приводит несколько примеров, демонстрирующих, в частности, отличие массивов от указателей и эффект выравнивания:

Описание	Размер (`sizeof`)
`char c;`	1
`char * p;`	4
`char s[] = "abc";`	4
`char a3[] = { 'a','b','c' };`	3
`struct T1 { char x; short y; char z;} S1;`	6
`struct T2 { int x; char y; } S2;`	8

Поскольку значение аргумента для этой операции неважно, то аргумент не вычисляется, и более того, в качестве аргумента

можно использовать сам тип, т. е. вместо `sizeof(p)` можно написать и `sizeof(char*)`. Ввиду того что размер любого типа определен статически, вызов `sizeof` является константным выражением и заменяется на конкретное значение во время трансляции. Таким образом, более надежным было бы пример с `malloc` записать как

```
float * A =(float *) malloc(sizeof(*A) * 10);
```

что гарантирует, что будет выделена память для 10 элементов размера типа `*A`, т. е. `float`.

Знание об природе аргумента может быть передано и с помощью функциональных параметров. Например, для реализации алгоритма сортировки массива достаточно знать размер элементов, их количество и функцию, сравнивающую два элемента, опять же специфицированных как `void *`. В любом случае язык С полагается лишь на хороший стиль и аккуратность программиста, оставляя дыру в контроле типов. Так, будет легально с помощью манипуляций с приведением указателей

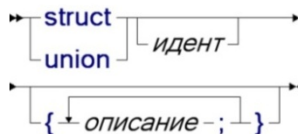
```
float f = 1.0;
int x = * (int *) (&f);
```

«взглянуть» на представление вещественного числа как на представление целого. Переменная `x` в результате может получить в зависимости от реализации, например, значение 1065353216.

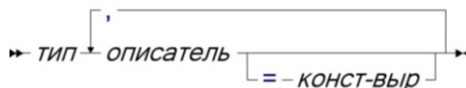
7.3.11. Описания, структуры и объединения

Тип структуры и объединения в языке С имеют почти одинаковый синтаксис, который задает перечисление полей, описание которых в свою очередь практически совпадает с описанием переменных:

тип-структура:

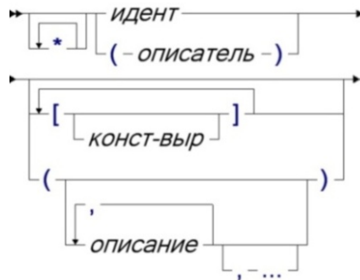


описание:



тип:

описатель:



Последняя альтернатива в диаграмме *тип* соответствует имени определяемого типа, а в диаграмме *описатель* – типу функции, которые обсуждаются ниже.

Синтаксическая категория *тип* не покрывает все возможные типы языка C: указатели, массивы и функции появляются в категории *описатель*. Это иногда приводит к неудобствам: например, не всякий тип можно непосредственно указать как аргумент операции `sizeof` и для этого приходится дополнительно описывать либо переменные, либо именованные типы. С другой стороны, в таком определении есть свои достоинства: запись описателя схожа со структурой применения операций разыменования и выборки компоненты по индексу при использовании. Это дает возможность совместить компактность записи и понимаемость даже для сложноустроенных типов:

Описание	Использование	Комментарий
`int (*p)[100];`	`(*p)[i]`	Указатель на массив целых
`int * (a[100]);`	`* (a[i])`	Массив из указателей на целые
`enum WeekDay ** ((*A)[][100]);`	`**((*A)[i][j])`	Указатель на массив из массивов из указателей на указатели на перечисление WeekDay
`long *q, n, m = 5;`	`*q, n, m`	Указатель на целое, целое и целое, равное 5

Синтаксис описания типа, который задает ему имя, отличается от описания переменной только служебным словом `typedef` в начале:

описание-типа:

Как и в описании переменной, в одной конструкции описания типа может быть определено несколько разных типов. Например,

```
typedef float Matrix[N][N], Vector[N];
Matrix A;
Vector v;
```

Основной как для типа структуры, так и для типа записи является постфиксная операция выборки поля, которая обозначается точкой, за которой следует имя поля. Для структур имеются литеральные значения, которые перечисляют значения полей в порядке их появления в описании структуры:

```
typedef struct { flaot re, im; } complex;
complex c1= {-1, 0}, c2 = {3.14,2.78}, c;
c.re = c1.re * c2.re - c1.im * c2.im;
c.im = c1.re * c2.im + c1.im * c2.re;
```

При этом в записи значения структур допускаются не только константы, что делает это особой формой выражения:

```
 c =
     {
       c1.re * c2.re - c1.im * c2.im,
       c1.re * c2.im + c1.im * c2.re
     };
```

Различные структуры могут иметь поля с одним и тем же именем. В этом смысле операция выборки является перегруженной – нужная операция однозначно определяется типом аргумента.

Очень часто используется комбинация из последовательности операций разыменования указателя на структуру с последующей выборкой поля, как, например, `(*p).next`. Для этого случая используется сокращенная форма: лексема `->`, за которой следует имя поля, т. е. последнее выражение полностью эквивалентно `p->next`.

Тип объединения реализует простое наложение альтернатив без какого-либо контроля того, какая именно альтернатива является актуальной. Как мы уже говорили, это является еще одной «дырой» в кон-

троле типов, т. е. считается вполне законной последовательность действий

```
union
{
    unsigned long l;
    unsigned char c[4];
} b4;
b4.l = 0xAABBCCDD;
b4.c[1] = 'A';
```

в результате значение поля `l` станет равным `0xAA41CCDD`.

Поэтому для «корректного» использования объедения используется его сочетание со структурой, в которой одно из полей определяет текущую альтернативу. Кроме того, в отличие от Паскаль, в языке С альтернатива не может содержать несколько полей, и поэтому полем объединения зачастую тоже является структура. Далее мы будем использовать следующий пример – структуру, описывающую вершину в дереве абстрактного синтаксиса в модельном языке выражений:

```
enum ExprCode
{
  EC_VALUE,
  EC_VAR,
  EC_UNOP,
  EC_BINOP
};
struct Expr {
  int tag;
  enum ExprCode code;
  union {
    float value;
    char name[8];
    struct {
      char op;
      struct Expr * arg;
    } unop;
    struct {
      char op;
      struct Expr *left, *right;
    } binop;
  } choice;
};
```

Здесь в каждой вершине имеется поле `tag`, хранящее вспомогательную информацию. Поле `code` определяет тип вершины, в зависимости от которого интерпретируются остальная информация:

Значение tag	Тип вершины	Поле в объединении `choice`
`EC_VALUE`	Вещественное число	`value` – значение числа
`EC_VAR`	Переменная	`name` – имя переменной
`EC_UNOP`	Применение унарной операции	`unop`, где • `op` – код операции; • `arg` – ссылка на вершину-аргумент.
`EC_BINOP`	Применение бинарной операции	`binop`, где • `op` – код операции; • `left` и `right` – ссылка на левое и правое подвыражение соответственно.

Теперь, если переменная `e` описана как

```
struct Expr * e;
```

то доступ к левому подвыражению бинарной операции будет иметь вид

```
e->choice.binop.right
```

но при этом на программисте лежит ответственность, что перед этим проверено, что значение `e->code` равно `EC_BINOP`.

7.3.12. Присваивания

Целью присваивания является изменение значения некоторого объекта. В языке С присваивание обозначается символом равенства «=» и рассматривается как специальная операция, первый аргумент которой называется *получателем* и определяет изменяемую переменную, а второй – *источником*, вычисляющим присваиваемое значение. То есть в отличие от обычных операций присваивание вычисляет не значение, задаваемое получателем, а только его адрес.

Типы получателя и источника должны быть согласованы. Если оба являются арифметическими типами, то происходит преобразование результата вычисления источника к типу получателя. При этом может происходить потеря точности. Например, при

```
int x;
x = 1.6;
```

произойдет округление вещественного значения за счет отбрасывания дробной части и x получит значение 1. Как уже говорилось ранее, потеря точности может происходить и в других случаях, преобразовании беззнаковых в знаковые, длинных целых в вещественные и т. п. Транслятор по мере возможности в таких случаях выдает предупреждение.

Массивы, поскольку они являются константными указателями, присваивать нельзя, т. е.

```
int a[3], b[3];
a = b;
```

недопустимо.

С другой стороны, структуры присваивать можно, даже если в них и имеются поля-массивы. Например,

```
struct
{
    int x;
    char y;
    int m[3];
} a, b;
a = b;
```

скопирует содержимое структуры b, т. е. отрезок памяти размера sizeof(b), включающий в том числе и все элементы поля m в структуру a.

Результатом выполнения операции присваивания является присвоенное значение. Тот факт, что присваивание в языке C является выражением, имеет как плюсы, так и минусы. К достоинствам можно отнести то, что иногда это позволяет сократить

запись. Например, если необходимо присвоить одно и то же значение нескольким переменным, то его можно указать только один раз:

```
x = y = z = 1.6;
```

Другой типичный случай возникает, когда результат присваивания нужно тут же использовать для других вычислений, как в

```
z = (x = 3) + (y = 4);
```

где x, y и z получат значения 3, 4 и 7 соответственно.

Существенным недостатком такого подхода является то, что наличие побочных эффектов в выражениях может приводить к неожиданным результатам. Рассмотрим, например, следующий фрагмент:

```
float A[N];
int  i=0, j=0;
A[i+j] = (i=1) + (j=i+1);
```

Естественно предположить, что аргументы сложения вычисляются слева направо, а источник присваивания вычисляется раньше получателя. В этом случае A[3] получит значение 3. Однако сделанное предположение **неверно** и может оказаться, что в данном конкретном случае транслятор реализует это присваивание как

```
float *t = &(A[i+j]);
j = i+1;
i = 1;
*t = i+j;
```

в результате чего A[0] получит значение 2. Очевидно, что возможны и другие варианты.

В языке С есть несколько видов присваиваний, *совмещенных* с выполнением других операций. Так, следующие два присваивания эквивалентны:

```
x = x + 2;
x += 2;
```

Сокращенная форма записи несомненно повышает наглядность и лучше отражает смысл операции: увеличить x на 2. Помимо

операции += допустимы также -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, однако, за <= и >= уже зарезервирован другой смысл – сравнение, а &&= и ||= недопустимы ввиду специфичности семантики псевдоопераций && и ||.

Если получатель присваивания является сложным выражением, то в случае совмещенного присваивания он будет вычисляться только один раз, что делает выполнение более эффективным. Однако это также может существенно изменить семантику по сравнению с несовмещенным присваиванием, как в случае

```
M[i+=1] += 2;
```

что, очевидно, **не эквивалентно**

```
M[i=i+1] = M[i=i+1] + 2;
```

хотя бы потому, что i увеличится два раза, а не один.

Среди совмещенных присваиваний особенно часто используются увеличение и уменьшение на 1, как

```
x += 1;
x -= 1;
```

Для этих случаев в языке C предусмотрены специальные унарные операции – инкремента ++ и декремента --. Таким образом, предыдущий фрагмент эквивалентен

```
x++;
x--;
```

Поскольку точно так же как простое или совмещенное присваивание, применение инкремента или декремента является выражением, то надо определить, какое значение оно вычисляет. Если следовать аналогии с присваиванием, то следует вернуть новое (т. е. присвоенное) значение. Это оказывается не всегда удобно. Вспомним, например, функцию копирования строки, где основной цикл имел вид

```
while (*p++ = *q++);
```

Здесь замысел состоял в том, чтобы присвоить то, куда указывает q, туда, куда указывает p, а затем уже сдвинуть значения p и q к следующим символам, т. е. цикл эквивалентен:

```
while (*p = *q)
{
   p++;
   q++;
}
```

причем неважно, в каком порядке изменяются p и q. Равенство в условии цикла означает не сравнение на равенство, а присваивание, значением которого будет текущее значение q, а само присваивание будет выполнено когда-то позже, но до того, как изменятся q и p. Это позволяет записать тот же цикл как

```
while (*q)
{
   *p = *q;
   p++;
   q++;
}
```

Осталось заметить, что *q в условии истинно, когда оно не равно нулю, т. е. нагляднее было бы записать его как `*q != '\0'`. Конечно, можно считать, что выбор между разными формами записи этого цикла – дело вкуса. Аргументом в пользу короткой записи является тот факт, что в ходе преобразования мы внесли дополнительные ограничения на порядок выполнения p++ и q++, а также ввели лишнюю операцию !=. Однако эти аргументы становятся неубедительными, учитывая, что современные трансляторы достаточно умные, чтобы разобраться с этими проблемами. Так что преимущество в данном случае должен иметь наиболее понятный с точки зрения программиста вариант, на что исходный код вряд ли может претендовать.

Вернемся к операции инкремента. В языке C имеется как префиксная, так и постфиксная форма этой операции:

```
++x
x++
```

соответственно. Постфиксная форма выдает исходное значение переменной, а префиксная – новое, что семантически эквивалентно

```
x += 1
(t = x, x += 1, t)
```

где t – вспомогательная переменная. В случае если результат инкремента не используется, то две эти формы эквивалентны. То же справедливо и для операции декремента --.

7.3.13. Нотационная путаница

Вероятно, стремление к краткости записи является причиной того, что одни и те же символы используются в формировании разных лексем. Это приводит к тому, что «гибкость» синтаксиса оборачивается его неустойчивостью.

Использование символа равенства «=» для обозначения присваивания противоречит естественному, привычному всем со школьного курса смыслу. Те соображения, что присваивания в программах встречаются чаще, чем сравнение на равенство, и что так было принято в языке Фортран, вряд ли можно считать достаточным обоснованием. В языках, происходящих от языка Алгол, для присваивания используется лексема :=, а в языках Кобол и Basic – вообще многословные операторы

```
MOVE X TO Y
LET Y = X
```

что уже не спутать с проверкой на равенство, но это уже явный перебор.

Возможность использовать присваивание в качестве выражения и отсутствие в языке явного типа логического и битовых шкал не позволяет своевременно диагностировать ошибки, связанные

с некорректным применением операций `&`, `&&`, `|`, `||`, <=, <<= и т. п. Рассмотрим, например, выражение

```
x=2 & y>0
```

которое, естественно, воспринимается как конъюнкция двух условий `x=2` и `y>0`, что истинно при `x`=2 и `y`=1. Без каких-либо предупреждений данное выражение будет воспринято как

```
x = ((2 & y)  > 0 )
```

что ложно, причем `x` получит новое значение 0. Желаемый результат задается внешне очень похожим выражением:

```
x == 2 && y>0
```

Стремление к краткости иногда приводит к коду, похожему на криптограмму. В языке С допустимы, например, выражения

```
a++ + ++b
a++ + +b
a+ ++b
a+ + +b
```

причем все имеют разный смысл.

8. УПРАВЛЕНИЕ

Для того чтобы действия в программе выполнялись в нужном порядке, используется управление. В простейшем случае, как, например, в машине Тьюринга, для каждой команды указывается следующая, выбор которой может зависеть от обрабатываемых данных. Такой чисто императивный способ управления является во многих случаях избыточно жестким. Ниже мы рассмотрим несколько видов управления, характерных для императивных языков программирования:

- *выражения*, главной целью которых является вычисление значений, а порядок исполнения определяется *зависимостью по данным*. Например, для того чтобы вычислить выражение `(x+y)*(x-y)`, не важно, в каком порядке будут вычислены аргументы умножения – главное, чтобы оба они были вычислены до собственно применения этой операции. В таких случаях язык задает частичный порядок на некотором подмножестве действий [26];
- *операторы*, целью которых является изменение состояния памяти. Последовательность выполнения операторов задается императивно, т. е. следующий за данным оператор задается либо однозначно, либо выбирается из двух или более альтернатив;
- *процедуры и функции* позволяют определить совокупность действий, изменяющих состояние памяти и/или вырабатывающих некоторое значение, и исполнять ее многократно, возможно, меняя некоторые параметры;
- *обработка исключительных ситуаций*, перехватывающая управление в случае, если вдруг где-то произошло событие – деление на ноль, исчерпание памяти, выход за границы индексов и т. п.

Этот перечень не охватывает все возможные способы организации управления. За рамками рассмотрения остаются параллельные, потоковые, событийно-управляемые вычисления, вычисления, основанные на сопоставлении с образцом, логическом выводе и т. п., не говоря уже про нейронные сети и квантовые вычисления.

---

[26] Понятно, что в рамках выполнения всей программы, если рассмотренное выше выражение вычисляется многократно, то вычисление аргументов может выполняться и после умножения, выполненного на предыдущей итерации.

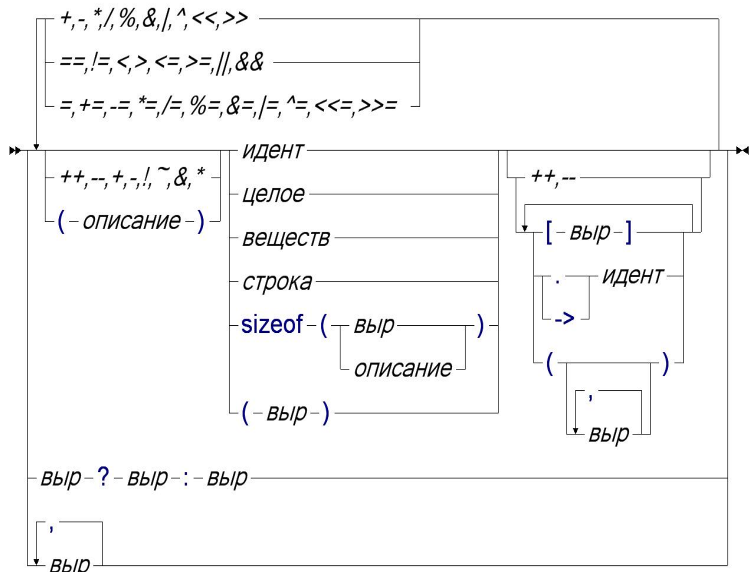
8.1. Выражения

Выражения в языке С строятся из:

- имен переменных;
- литеральных значений и имен констант;
- применения операций;
- разыменования, взятия адреса, выборки компонент массивов и структур;
- явного приведения типа и вычисления размера типа;
- группирования вычислений скобками;
- вызова функций и процедур;
- условного и последовательного выражений.

Синтаксическая диаграмма, которая сводит воедино все эти конструкции, имеет следующий вид:

выр:



Очевидно, что описанный синтаксис существенно сложнее, чем тот модельный язык выражений, который мы использовали в качестве примера, даже учитывая то, что в нем не учитывается приоритет операций, который также имеет гораздо больше уровней. В порядке возрастания приоритетов можно все операции можно разбить на

15 групп – чем выше приоритет, тем сильнее операция связывает аргументы и тем раньше она выполняется:

<table>
<tr><td>1</td><td>() [] -> . ++ --</td><td>Вызов функции, выбор поля или компоненты массива, постфиксные инкремент и декремент</td></tr>
<tr><td>2</td><td>! ~ - ++ -- + * &(тип) sizeof</td><td>Унарные, префиксные инкремент и декремент, приведение типа, sizeof</td></tr>
<tr><td>3</td><td>* / %</td><td>Умножение</td></tr>
<tr><td>4</td><td>+ -</td><td>Сложение</td></tr>
<tr><td>5</td><td><<>></td><td>Битовый сдвиг</td></tr>
<tr><td>6</td><td><<= >>=</td><td>Сравнения</td></tr>
<tr><td>7</td><td>== !=</td><td>Равенство</td></tr>
<tr><td>8</td><td>&</td><td>Побитовое AND</td></tr>
<tr><td>9</td><td>^</td><td>Побитовое XOR</td></tr>
<tr><td>10</td><td>|</td><td>Побитовое OR</td></tr>
<tr><td>11</td><td>&&</td><td>Логическое AND</td></tr>
<tr><td>12</td><td>||</td><td>Логическое OR</td></tr>
<tr><td>13</td><td>?:</td><td>Условное выражение</td></tr>
<tr><td>14</td><td>= += -= *= /= %= &= |= ^= <<= >>=</td><td>Присваивания</td></tr>
<tr><td>15</td><td>,</td><td>Последовательное выполнение</td></tr>
</table>

Приоритет операций в разных языках программирования может отличаться и не всегда соответствует интуиции. Например, следующие выражения:

```
a = b && c
a & b == c
*a[i]++
a & 2 << 3
```

будут соответственно трактоваться как

```
a = (b && c)
a & (b == c)
*(a[i]++)
a & (2 << 3)
```

Простое правило, которому надо следовать, – если у пишущего программу возникают сомнения в том, приоритет какой операции выше, то очень вероятно, что ровно такие же сомнения возникнут и у читающего программу. Поэтому в таких случаях рекомендуется использовать скобки, даже если их можно опустить.

По существу, среди всех возможных допустимых выражений мы не рассмотрели только вызовы функций, условные и последовательные выражения. Функции мы рассмотрим отдельно ниже. Что же касается условных и последовательных выражений, то они дублируют те возможности, которые реализуются на уровне операторов.

Условное выражение имеет вид:

условие ? *то-часть* **:** *иначе-часть*

Хотя условное выражение иногда и рассматривается как применение тернарной операции, оно отличается тем, что не требует вычисления всех составных частей до применения операции: при истинности *условия* вычисляется значение *то-части*, в противном случае – *иначе-части*, и полученное значение является значением всего выражения.

Естественно, для того чтобы можно было определить тип условного выражения, необходимо, чтобы тип одной из альтернатив можно было привести к типу другой. Так, например,

```
n > 0 ? 1 : &x
```

недопустимо, а значением

```
n > 0 ? 1 : 1.0
```

будет вещественное значение 1.0 вне зависимости от истинности условия.

Логические связки `&&` и `||` также по сути являются условными выражениями и реализуют так называемые конъюнкцию и дизъюнкцию по МакКарти (John McCarthy). Второй аргумент вычисляется только в случае, если первый оказался истинным и ложным, соответственно. Формально

```
A && B
A || B
```

эквивалентны соответственно

```
A ? B : 0
A ? 1 : B
```

Это позволяет использовать `&&` как *охраняющие* условия. Например,

```
(x !=0) && (1/x > 0)
(i >=0 && i < N) && (A[i] != 0)
```

не будут приводить к делению на 0 и выходу за границы индексов, что произошло бы, если бы вычислялись все подвыражения.

Управление в *последовательном выражении*

$$e_1, e_2, \dots, e_n$$

также отличается от обычного применения операций, поскольку оно требует, чтобы все «аргументы» вычислялись слева направо. Результатом всего выражения является значение последнего – e_n, а результаты всех предыдущих игнорируются. Таким образом, использование последовательного выражения осмысленно, только если все e_1, e_2, ...,e_{n-1} имеют побочные эффекты, как например,

```
c = (a=3, b=2+a, a+b);
```

что эквивалентно

```
a = 3;
b = 2 + a;
c = a+b;
```

но **не эквивалентно**

```
c = (a = 3) + (b = 2+a);
```

Формально язык не требует наличия побочных эффектов, что может приводить к неожиданным ошибкам [27]. Например,

```
A[i,j] = i+j
```

воспринимается как присваивание элементу двумерного массива, хотя на самом деле эквивалентно

```
A[j] = i+j
```

[27] Транслятор может выдать об этом предупреждение. А может и не выдать...

8.2. Операторы

Управление на уровне операторов будем описывать методом раскрутки, т. е. от простого к сложному. Набор управляющих конструкций в языке C весьма небольшой, как показывает следующая синтаксическая диаграмма:

оператор:

Начнем с первых двух альтернатив, которых, вообще говоря, вполне достаточно для реализации любой последовательности действий.

8.2.1. Базовые операторы и блоки

Самым простым является *пустой оператор*, который ничего не делает. Изображается он точкой с запятой.

Элементарным с точки зрения управления является выражение, предположительно имеющее побочный эффект, поскольку собственно результат выражения игнорируется. Типичным примером таких выражений являются присваивания и вызовы функций. Выражение превращается в оператор, когда за ним ставится точка с запятой.

Блоки нужны, во-первых, для того, чтобы из нескольких последовательно выполняющихся операторов сделать один, поскольку некоторые синтаксические конструкции требуют в качестве составной части только один оператор, и, во-вторых, дают возможность описать переменные, доступные только в этом блоке. Таким образом мы можем написать, например, следующий оператор, эффект

которого пуст, поскольку он состоит только из пустых операторов и операций с локальными переменными:

```
{  int x=1; {;} x=1; {int x; x;} }
```

Отметим, что точка с запятой именно завершает оператор, а не разделяет операторы в последовательности, как это принято, скажем, в языке Паскаль. Поэтому блок `{;}` содержит единственный пустой оператор, а после закрывающей скобки ставить точку с запятой не нужно, хотя и можно, но в этом случае она будет обозначать еще один пустой оператор.

8.2.2. Метки и goto

Каждый оператор может быть помечен одной или несколькими метками, изображаемыми идентификаторами. Метки используются для того, чтобы можно было передать на помеченный оператор управление с помощью оператора `goto`. Так, простейшая зацикливающаяся программа имеет вид

```
l: goto l;
```

Поскольку программа размещается в памяти (возможно, в специальной области), то можно считать, что у каждой команды есть адрес, а метка – не более, чем литеральное изображение адреса команды. Эти соображения позволяют рассматривать метки команд как отдельный тип данных, как это сделано, например, в языке GNU C – диалекте языка C. Типом метки считается указатель на `void`, и, следовательно, переменная такого типа может быть описана как

```
void * lab;
```

Для того чтобы по идентификатору метки получить значение типа метки, используется унарная операция `&&`. С полученным значением можно делать все, что позволяет язык: сохранять его в структурах данных, передавать в качестве параметра, использовать в альтернативах условных выражений, сравнивать с другими метками и т. п. Например, мы можем присвоить его в переменную `lab`:

```
lab = &&l;
```

В конечном итоге значение типа метки может быть использовано для передачи управления с помощью разновидности оператора `goto *`:

```
goto * lab;
```

Возможности, которые предоставляют вычисляемые метки, настолько неконтролируемы и опасны, что даже в описании языка GNU C предписывается использовать их лишь в тех случаях, когда ничто другое принципиально не подходит. В нашем случае мы ненадолго введем вычисляемые метки для описания семантики других управляющих конструкций, после чего внесем их в «черный список».

Указанных управляющих возможностей достаточно для написания следующей программы:

```
void * next[] = {&&l1, &&l2, &&l1, &&l3};
goto * next[(n>0) | ((n&1)<<1)];
l3: y*=x; goto * next[n>1];
l2: x*=x; goto * next[1 | ((n>>=1)&1)<<1];
l1:
```

в которой достаточно сложно узнать программу вычисления x^n, рассматривавшуюся ранее:

```
while (n > 0)
{
    if (n%2)
        y = y*x;
    x = x*x;
    n = n / 2;
}
```

Помимо того, что мы использовали совмещенные присваивания и использовали сдвиги вместо операций взятия по модулю и деления нацело на 2, все управление спрятано в массиве меток `next`. Младший бит индекса этого массива соответствует положительности *n*, а второй – нечетности *n*. Кроме этого, программа была оптимизирована содержательно на основе следующих наблюдений:

1. Если *n* положительно и нечетно, то после вычитания 1 оно обязательно будет четно;
2. Если *n* положительно и четно, то после деления на 2 оно обязательно будет положительно.

Таким образом, удалось избежать избыточных проверок.

Возможно, что эта программа и более эффективна, чем исходная, но читать и понимать ее совершенно невозможно. Для каждого поме-

ченного оператора необходимо просмотреть весь фрагмент, чтобы понять, откуда на него может быть передано управление. А в случае вычисляемых меток приходится решать и обратную задачу: для каждого оператора `goto` приходится выяснять, куда именно он может передавать управление. На фундаментальном уровне это выражается еще более серьезной проблемой: становится невозможным задать семантику программы путем композиции, как мы это делали при описании денотационной и аксиоматической семантики.

Переменные-метки и переходы по вычисляемым меткам не являются изобретением языка GNU C. Еще в языке Фортран были сходные конструкции, называемые переключателями. Приведем для примера пару фрагментов, реализующих одно и то же:

```
       t = X - (X/6)*6 +1
       GOTO (0,1,2,3,3,3) t
0      CONTINUE
2      X = X+2
3      X = X+1
       GO TO 100
       X = 0
100
```

и

```
       t = X - (X/6)*6
       ASSIGN 3 TO L
       IF (t .EQ.0) ASSIGN 0 TO L
       IF (t .EQ.1) ASSIGN 1 TO L
       IF (t .EQ.2) ASSIGN 2 TO L
       GOTO L, (0,1,2,3)
0      CONTINUE
2      X = X+2
3      X = X+1
       GO TO 100
1      X = 0
100
```

Смысл оператора `ASSIGN` и различных форм `GOTO` можно легко предположить. Заметим, что Фортран все-таки не допускал тех вольностей, которые допускает GNU C: для каждого оператора

goto указан список возможных меток, на которые он может передать управление, и этот список не может меняться в процессе исполнения.

Поскольку для меток используются имена, следует сказать об области действия этих имен. В языке C (по крайней мере в стандарте) нет явного объявления меток, и метка считается видимой во всей охватывающей функции. Таким образом, допускается переход внутрь блока, как, например

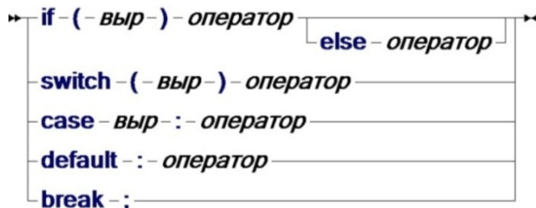
```
goto l;
for (int i=0; i<n; i++)
{
   if (i>0)
   {
        int x = 2;
        l: printf("%d", x*i);
    }
}
```

что приводит к непредсказуемым последствиям. Язык Паскаль требует объявления меток, но они также относятся к функциям и не согласуются с блочной структурой.

8.2.3. Ветвления

В языке C имеются две конструкции, реализующие выбор одной из ветвей в зависимости от значения выражения – условный if и переключатель switch. Их синтаксис задается следующим образом:

ветвления:



Условный оператор if имеется в том или ином виде во всех императивных языках программирования. Его семантика заключается в вычислении условия и выполнении одной из двух альтернатив в зависимости от истинности условия: *то-части*, следующей

непосредственно за условием, или *иначе-части*, следующей за ключевым словом `else`. Если иначе-часть не указана, то она полагается равной пустому оператору. Таким образом, оператор

```
if ( условие ) то-часть else иначе-часть
```

эквивалентен

```
goto * ( условие ? &&lThen : &&lElse ));
lThen : то-часть; goto lDone;
lElse : иначе-часть;
lDone:
```

Мы уже говорили о том, что зачастую условные операторы являются вложенными, как в

```
if ( условие1 )
        вариант1
else if ( условие2 )
        вариант2
...
else if ( условиеn )
        вариантn
else
        вариантelse
```

Конечно, это можно рассматривать как ветвление с более чем двумя альтернативами, но описанная выше семантика предписывает, что условия вычисляются последовательно до первого истинного.

Выбор из произвольного числа альтернатив реализуется переключателем `switch`. Описанный выше синтаксис говорит о том, что помечать операторы можно не только идентификаторами, но и метками вида `case` *выр* и `default`, которые допустимы только внутри переключателя. Выражение, следующее за `case`, должно быть константным. Выполнение переключателя начинается с вычисления выражения, указанного в скобках. Если значение равно *i*, то управление передается на оператор, помеченный меткой `case` *i*, а если такого оператора нет, то на оператор, помеченный меткой `default`. Если же и `default` отсутствует, то на оператор, следующий за `switch`. Оператор `break;` внутри переключателя также завершает его выполнение.

Рассмотрим, например, оператор

```
switch (x % 6)
{
  case 0 :
  case 2:
    x += 2;
  default :
    x += 1;
    break;
  case 1 :
    x = 0;
    break;
}
```

Он может быть реализован как

```
void * sw[] = { &&l0, &&l1, &&l2, &&lElse, &&lElse, &&lElse };
goto * sw[x % 6];
l0: l2: x += 2;
lElse: x+=1; goto lDone;
l1: x = 0; goto lDone;
lDone: ...
```

Отметим некоторые свойства переключателя:

- Значение выражения в переключателе (`x%6` в данном примере) вычисляется один раз;
- Выбор метки может быть реализован разными способами:
  1. В виде массива меток (как в данном примере), если известен их диапазон и он достаточно небольшой;
  2. Двоичным поиском (дихотомией), если альтернатив достаточно много и при этом имеется разброс значений;
  3. Последовательным перебором, если значений совсем немного;
  4. Оптимальным деревом поиска, если предварительно собрана статистика о частоте выполнения альтернатив и т. п.
- Завершение выполнения альтернативы, если она не заканчивается оператором `break;` не означает завершения выполнения всего переключателя. Так, в данном примере после выполнения `x+=2;` управление перейдет к `x+=1;`. На практике это свойство переключателя является источником ошибок, поскольку `break;` чаще всего отсутствует не намеренно. Конечно, можно привести при-

меры, когда такое поведение позволяет либо не дублировать код, либо избежать использования `goto`, но в целом вреда больше, чем пользы.

В общем, переключатель в языке C недалеко ушел от перехода по вычисляемой метке и позволяет писать плохо структурированные программы, включая переход внутрь блока, как в следующем примере:

```
int i = 1;
switch (i%2)
  if (i>0)
    case 0:
    {
      i++;
      if (i%2)
      case 1:
          i--;
      else
        break;
    }
```

При вложенных переключателях как метки `case` и `default`, так и оператор `break;` относятся к ближайшему охватывающему переключателю.

На практике в большинстве случаев переключатель соответствует следующему синтаксису, который принят в языке C#:

switch (*выр*) { case *выр* : | default : *оператор* }

что, видимо, можно рекомендовать и для использования в языке C.

Вариации

Условные операторы практические одинаковы во всех императивных языках программирования, если не принимать во внимание синтаксические различия, связанные с `else if` и условного с отрицанием `unless`. Редким исключением является язык Фортран, где помимо логического условного имеется также и *арифметический*, который основывается не на истинности выражения, а на

его знаке. Рассмотрим, например, фрагмент программы двоичного поиска:

```
int m = (low + high)/2;
if (A[m] == x)
   ; // нашли!
else if (A[m] > x)
   high = m-1;
else
   low = m+1;
```

Выбор одной из трех альтернатив зависит от знака выражения (A[m]-x), и в языке Фортран такое ветвление может быть записано как

```
IF (A[m]-x) 10, 20, 30
```

где метки 10, 20, 30 соответствуют случаям «меньше нуля», «равно нулю» и «больше нуля» соответственно. [28] В языке С такое ветвление можно записать как

```
switch (sign(A[m]-x))
{
   case 0  : break; // нашли!
   case 1  : high = m-1; break;
   case -1 : low = m+1; break;
}
```

что может быть несколько нагляднее, чем исходная версия, поскольку выносит в заголовок суть выбора, и эффективнее, поскольку здесь меньше обращений к массиву А.

Некоторые языки программирования допускают использование *интервалов* в качестве меток переключателя. Так, в языке Паскаль допустим следующий оператор:

```
case ch of
'A'..'Z', 'a'..'z' : WriteLn('Буква');
'0'..'9'           : WriteLn('Цифра');
'+', '-', '*', '/' : WriteLn('Операция');
else
    WriteLn('Спецсимвол')
end
```

[28] В языке Фортран метки обозначаются числами, а не идентификаторами.

Конечно, интервал можно было бы вручную «раскрыть», перечислив все его значения. Однако при этом теряется наглядность и появляется риск пропустить некоторое значение. К тому же, если в качестве типа меток используется перечисление `enum`, то при добавлении нового элемента к этому типу придется исправлять и все места, где он используется.

Язык С требует, чтобы метки переключателя были *целого типа* (включая, естественно, тип `char` и `enum`), но не допускает строки. В том же Паскале допустим следующий переключатель:

```
case lowercase(s) of
'huge', 'large', 'big' : FontSize=18;
'normal', 'medium'   : FontSize=12;
'small', 'little', 'tiny', : FontSize=8;
else
    FontSize=12;
end
```

Для такого переключателя не подойдет реализация в виде массива меток, но он также может быть реализован более эффективно, нежели последовательность сравнений строк. Например, можно на уровне трансляции построить регулярное выражение и соответствующий конечный автомат, который будет находить нужную альтернативу за один проход по строке.

В некоторых языках, например, в Visual Basic, нет требования, чтобы метки были константами. Если бы то же было и в языке С, то рассмотренный выше фрагмент программы двоичного поиска можно было бы реализовать следующим образом:

```
switch (1)
{
    case A[m]==x : break; // нашли!
    case A[m]>x  : high = m-1; break;
    case A[m]<x  : low = m+1; break;
}
```

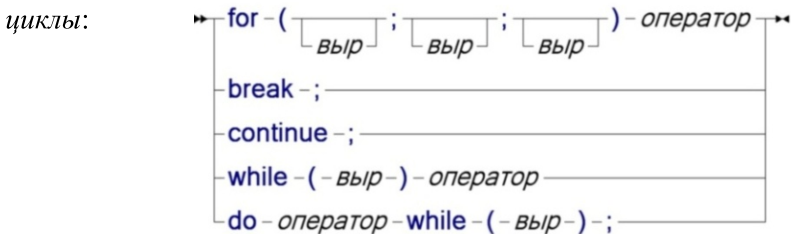
В данном примере можно гарантировать, что будет истинно ровно одно из условий, однако в общем случае невозможно ни определить диапазон значений, ни проверить то, что все метки различны. Поэтому накладывается требование, чтобы проверки осуществлялись последовательно, что семантически сводит переключатель `switch` к последовательности вложенных условных операторов.

Аналогичные проблемы возникают и в случае, если язык допускает в переключателе не только сравнения на равенство, а, например, сопоставление строки с *шаблоном* или регулярным выражением.

8.2.4. Циклы

Циклы предназначены для многократного повторения некоторой совокупности действий. В языке C есть три вида оператора цикла и два оператора, связанных с циклами, – `break;` и `continue;`, синтаксис которых определяется следующей диаграммой:

циклы:



Семантику циклов начнем описывать с оператора `for`. Если *init*, *condition* и *step* – выражения, а *body* – оператор, то конструкция

```
for (init; condition; step)
        body
```

эквивалентна

```
        init;
lLoop:
        if (!condition) goto lDone;
        body
lStep:
        step;
      goto lLoop;
lDone:;
```

Если условие *condition* отсутствует, то оно полагается тождественно истинным, т. е. условный оператор можно опустить. В этом случае тело цикла *body* будет выполняться бесконечно много раз, если только в нем не содержится оператор перехода вне цикла.

Выражения инициализации *init* и шага цикла *step* имеют смысл, только если они содержат побочный эффект. Типичное их использование состоит в задании начального значения и изменении *переменной цикла*, как например,

```
for (i=0; i<N-1; i++)
     A[i] = A[i+1];
```

В отличие от многих языков программирования, где переменная цикла, границы и шаг ее изменения задаются явными синтаксическими конструкциями, язык С предоставляет большую гибкость. Так, возможно в заголовке цикла задать несколько переменных не обязательно целого типа. Например, тот же цикл можно реализовать более эффективно:

```
for (q=(p=A)+1,i=N-1; i; p=q++,i--)
    *p = *q;
```

где переменные `p` и `q` указывают на очередной и следующий элементы массива `A` соответственно и не участвуют в условии цикла, а переменная `i` используется, чтобы обеспечить количество итераций, равное `N-1`.

Операторы `break;` и `continue;`, называемые *структурными переходами*, означают передачу управления на `lDone` и `lStep`, соответственно, т. е. `break;` завершает выполнение цикла, а `continue;` переходит к следующей итерации.

Циклы `while` и `do...while` можно рассматривать как синтаксический сахар: их легко выразить через цикл `for`. Оператор

```
while (condition) body
```

эквивалентен

```
for (;;)
{
    if (! condition) break;
        body
}
```

а оператор

```
do body while (condition);
```

эквивалентен

```
for (;;)
{
    body
    if (!condition)
        break;
}
```

То есть обе эти формы цикла выполняются, пока истинно условие, но в первом случае проверка производится перед очередной итерацией, а во втором – после.

Вариации

Во многих языках программирования есть вариант цикла, условие которого определяет не продолжение, а *завершение* цикла. Так, в Паскале цикл

```
repeat ... until condition
```

будет выполняться до тех пор, пока `x` не станет меньше 0.01. По каким-то причинам в Паскале условие завершения можно указывать только в конце цикла, а условие продолжения – только в начале. В языке Visual Basic имеется полный набор комбинаций:

```
Do While condition
    ...
Loop

Do Until condition
   ...
Loop

Do
    ...
Loop While condition

Do
    ...
Loop Until condition
```

Все эти формы цикла легко выражаются в языке С, поскольку условие завершения есть просто отрицание условия продолжения.

На практике очень часто встречается ситуация, когда надо некоторое действие повторить определенное *количество раз*. В языке Альфа-6 (диалекте Алгол-60) для этой цели введена особая форма цикла

```
N раз ...
```

Конечно, она может быть реализована в языке С как

```
for (k=N; k--;) ...
```

но менее наглядно и с необходимостью явно вводить дополнительную переменную.

Иногда необходимо выполнить цикл для *заданного множества значений* переменной цикла. Так, язык Алгол-60 позволяет оператор

```
for i=1,2,-3,4,10 do ...
```

В языке С это может быть реализовано путем введения вспомогательного массива, хранящего это множество значений:

```
{
  int values[] = {1,2,-3,4,10};
  for (k=0; i=values[k], k<5; k++)
    ...
}
```

Опять же это менее наглядно и требует дополнительного блока, чтобы расположить описание массива ближе к его единственному использованию.

Обобщением цикла по перечню значений является *цикл по структуре данных*, элементы которой могут быть перечислены: массива, списка, строки и т. п. Так, например, в языке Visual Basic для массива A можно написать цикл

```
Dim A(1 To N) As Integer
For Each x In A
   ... x ...
Next x
```

Заметим, что здесь переменная x не является синонимом очередного значения массива, и присваивание ей не будет менять состояние A. Далее обобщение цикла можно развить на основе понятия *итератора* – объекта, из которого можно получить очередной элемент обрабатываемой последовательности, если она еще не исчерпана.

По другому пути пошло обобщение в языке Алгол-60. Он рассматривает каждое из перечисленных значений как отдельный заголовок цикла, описывающий единственное значение, а в общем случае заголовком может быть и перечисление с использование while, until, step и т. п. Такое обобщение позволяет написать, например, следующий цикл с тремя заголовками:

```
for i:=1,
    4 while x>0.01,
    N step -1 until 10 do
  ...
```

где переменная i на первой итерации будет равна 1, затем выполнится какое-то количество итераций при i равном 4 до тех пор, пока x не станет меньше или равным 0.01, а затем i будет пробегать в обратном порядке значения от N до 10.

На практике весьма редко возникают ситуации, требующие несколько заголовков цикла, помимо рассмотренного выше случая, когда все заголовки являются отдельными значениями. Рассмотрим, например, цикл, в котором надо удвоить все элементы массива, за исключением k-го:

```
for i := 1 step 1 until k-1, k+1 step 1 until N do
    A[i] :=* * 2;
```

Реализация цикла со многими заголовками весьма нетривиальна. Можно заменить этот цикл несколькими, в каждом из которых будет один заголовок. Но при этом придется раскопировать тело цикла, что нежелательно, если оно достаточно сложное. Другой способ заключается в организации внешнего цикла, который будет перебирать заголовки, а переходы, организующие

внутренний цикл, заменить на переходы по вычисляемой метке. В языке C такая реализация может выглядеть как

```
void * lInit[] = { &&lInit1, &&lInit2 } ;
void * lStep[] = { &&lStep1, &&lStep2 } ;
int l;
for (l=0; l<2; l++) // цикл по заголовкам
{
  goto* lInit[l];
lInit1: i=1; goto lLoop1;
lInit2: i=k+1; goto lLoop2;
lLoop1:
  if (i>k-1)continue; // переход к следующему заголовку цикла
  goto lBody;
lLoop2:
  if (i>N)continue; // переход к следующему заголовку цикла
  goto lBody;
lBody:
  A[i] *= 2; // тело исходного цикла
  goto* lStep[l];
lStep1: i++; goto lLoop1;
lStep2: i++; goto lLoop2;
}
```

Заметим, во-первых, что здесь не пришлось копировать тело цикла, а во-вторых, что при такой реализации в каждом внутреннем цикле нет дополнительных накладных расходов, за исключением перехода по вычисляемой метке

```
goto* lStep[l];
```

Однако общая организация цикла достаточно сложна и в данном конкретном случае не дает особого выигрыша по сравнению с

```
for i := 1 step 1 until N do
    if (i^=k) A[i] :=* * 2;
```

где на каждой итерации делается одна дополнительная проверка. Кроме того, она явно уступает рассмотренной выше реализации для случая, когда все заголовки цикла – константы.

Таким образом, можно сделать вывод, что стремление к обобщению не всегда является продуктивным и лучше иметь несколько частных конструкций с эффективной реализацией, чем одну со сложной и, возможно, не самой эффективной.

Необходимо сделать несколько замечаний о *переменной и границах цикла*. Как мы уже отметили, в языке С такого понятия в явном виде нет, что, с одной стороны, предоставляет большую гибкость, а, с другой, – лишает возможности дополнительного контроля. В языке Паскаль цикл `for` явно указывает переменную цикла и ее границы, как, например, в

```
for i:=1 to Length(s) do
  if (s[i] = " ") Inc(cnt);
```

При этом гарантируется, что границы цикла будут вычисляться только один раз. Это может оказаться существенным с точки зрения как семантики, так и эффективности. Например, в аналогичном цикле на языке С

```
for (i=0; i<strlen(s); i++)
    if (s[i] == ' ') cnt++;
```

сложность будет пропорциональна не длине строки, а квадрату длины, поскольку `strlen(s)` вычисляется на каждой итерации и требует просмотра всей строки.

Очень нехорошей является идея изменять переменную и границы внутри тела цикла. Например, в Паскале до появления в нем оператора `break;` для прерывания цикла использовался следующий прием: переменной цикла присваивалось значение, большее верхней границы. Так, например, следующий цикл прерывает выполнение, если очередной элемент массива `A` оказывается нулевым:

```
for i:=1 to N do
  if A[i] = 0 then i := N+1;
```

Язык Алгол-68 вообще считает переменную цикла константой, объявленной в заголовке: присваивание ей в теле цикла приведет к синтаксической ошибке, а значение переменной цикла после его завершения не определено.

Рассмотрим теперь подробнее операторы структурного перехода: break; и continue;. С ними возникает проблема, когда они используются во вложенных циклах. Так, continue; относится к ближайшему охватывающему циклу, а break; – к циклу или оператору switch, где break; имеет свой смысл. В частности, это означает, что если телом цикла является оператор switch, то невозможно использовать break; для того, чтобы прервать цикл.

```
for (i=0; i<N-1; i++)
  switch (s)
  {
    case '0' :
        ....
        continue; // относится к for
    default :
        ....
        break;    // относится к switch
  }
```

Аналогично, при вложенности циклов невозможно прервать охватывающий цикл. Пусть, например, требуется найти первую строку матрицы с нулевым элементом. Следующий фрагмент будет делать это **неправильно**:

```
int found = -1;
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    if (A[i][j] == 0)
    {
       found = i;
       break;
    }
```

поскольку после нахождения нулевого элемента будут проигнорированы оставшиеся элементы в текущей строке, после чего управление перейдет к следующей итерации внешнего цикла. В результате будет найдена не первая, а последняя строка, содержащая нулевой элемент. Для того чтобы исправить эту ошибку, можно завести дополнительную логическую переменную, которую сделать истинной в момент прерывания внутреннего цикла,

а сразу после этого цикла прервать и внешний, если эта переменная истинна:

```
int found = -1;
for (i=0; i<N; i++)
{
  int do_break = 0;
  for (j=0; j<N; j++)
    if (A[i][j] == 0)
    {
      found = i;
      do_break = 1;
      break;
    }
  if (do_break)
    break;
}
```

Конечно, таким образом удалось сохранить структурированность программы, но за счет дополнительных накладных расходов, потери наглядности и надежности, т. е. всего того, для чего структурированность и предназначена.

Проблема, очевидно, заключается в том, что в структурном переходе `break;` невозможно указать, к какому именно оператору он относится. Некоторые языки, например, Java, решают эту проблему введением понятия *структурных меток*, которыми могут быть помечены циклы или другие составные операторы. Тогда тот же фрагмент можно вернуть практически к исходному виду:

```
int found=-1;
iloop:
for (i=0; i<n; i++)
  for (j=0;j<N;j++)
    if (A[i][j]==0)
    {
       found = i;
       break iloop;
     }
```

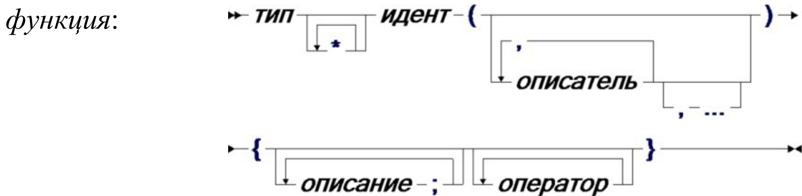
В языке C такие ситуации относятся к тем редким случаям, когда оправдано применение оператора `goto`.

8.3. Процедуры и функции

Процедуры и функции позволяют описать некоторую параметризованную совокупность действий, которую затем можно многократно вызвать в разных местах программы. Различие между функциями и процедурами состоит в том, что первые предназначены для вычисления значения результата и их вызов является выражением, а вторые – исключительно для изменения состояния памяти и, соответственно, их вызов является оператором. Поскольку в C, как и во многих других языках программирования, включая Паскаль, Алгол и т. д., функции тоже могут иметь побочный эффект, в языке оставлено только понятие функции, а процедуры считаются их частным случаем с результатом типа `void`.

8.3.1. Описание функций

Синтаксис описания функции в языке C имеет следующий вид:



В нем отражается несколько, вообще говоря, самостоятельных аспектов:

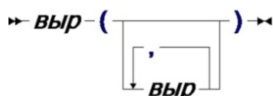
1. описание *типа функции*, которая включает тип результата, с которого начинается описание, а также типы и имена *формальных параметров*, задаваемые описателями. Если список формальных параметров завершается многоточием, то это означает, что функция может иметь дополнительные параметры;
2. *имя* функции;
3. *тело функции* – блок, определяющий последовательность выполняемых действий с указанием того, что является *результатом*.

8.3.2. Вызов функции

Функции в языке C являются значениями, хотя и не вполне равноправными. Единственная операция, которая к ним применима, – это *вызов функции*. Однако имя функции, если оно используется не в вызове, трактуется как указатель на функцию, который уже

может быть присвоен переменной соответствующей типа, передан в качестве параметра другой функции и т. п. Таким образом, вызываемая функция может быть получена не только указанием имени, но и применением разыменования к выражению, вычисляющему указатель на функцию.

Синтаксис вызова функции имеет следующий вид:



что включается в общий синтаксис выражения. Начинается вызов с выражения, вычисляющего функцию, а затем в скобках указываются выражения, вычисляющие фактические параметры. Количество фактических параметров в вызове функции должно совпадать с количеством формальных, если только это не функция с переменным количеством параметров. Типы фактических параметров должны быть приводимы, как при присваивании, к типу соответствующих формальных.

Для того чтобы из вызова функции сделать оператор, например, если эта функция является процедурой, достаточно, как и для любого другого выражения, поставить после вызова точку с запятой.

Приведем несколько примеров вызова функции:

```
ToPolar(x, y, &alpha, &ro)
* (shift ? sin : cos) (n * pi / 3)
(* F[i])(x > 0 ? 1 : x=-x , -1)
Ack(m-1, Ack(m,n-1))
WriteLn()
```

Первый вызов – пример самого распространенного случая, когда явно указывается имя вызываемой функции. Во втором вызове будет вызываться либо `sin`, либо `cos` в зависимости от значения переменной `shift`, но аргумент будет один и тот же. В третьем случае `F` – массив указателей на функции, и будет вызван один из его элементов. Четвертый вызов демонстрирует, что фактическим параметром также может быть вызов функции. Наконец, последняя строка – вызов функции `WriteLn` без пара-

метров. Типичной ошибкой является попытка вызова функций без скобок:

```
WriteLn;
```

как это принято в таких языках, как Паскаль или Visual Basic. К сожалению, в языке C это является вполне законной конструкцией, вычисляющей функцию как значение и тут же игнорирующей его, но не приводящей к ее вызову.

Вызов функции – шаги исполнения:

1. вычисляется вызываемая функция;
2. создаются локальные объекты: формальные параметры, локальные объекты тела функции;
3. вычисляются фактические параметры, значения которых копируются в соответствующие локальные объекты;
4. выполняется тело функции;
5. удаляются локальные объекты;
6. возвращается результат.

Возвращаемое функцией значение указывается в операторе `return` со следующим синтаксисом:

return [expr] ;

Выражение должно присутствовать тогда и только тогда, когда тип возвращаемого значения отличен от `void`. Кроме этого, оператор `return` завершает исполнение процедуры, т. е. переходит к шагу 4. Точнее говоря, корректное исполнения функции, которая не является процедурой, <u>обязано</u> завершаться оператором `return`.

Каждая программа должна содержать *главную функцию*, называемую `main`, следующего типа:

```
int main(int argc, char * argv[])
```

Система программирования вызывает эту функцию, подготовив все необходимое окружение. В частности, в качестве фактических параметров передается массив `argv` из аргументов командной строки и их количество `argc`.

Семантику исполнения функций мы будем демонстрировать на примере программы вычисления полинома:

```
float poly(float coef[], int n,float x)
{
   float sum = 0f;
   for (int i=0; i<=n; i++)
      sum += coef[i] * power(i,x);
   return sum;
}
float power(int n, float x)
{
  return n==0 ? 1 : x*power(n-1,x);
}
void main()
{
  float binom[] = {1,2,1};
  printf("%d", poly(binom,2,10.0));
}
```

В этой программе определены три функции:

1. `main` – главная функция:
2. `poly` – функция вычисления полинома, получающая в качестве параметров массив коэффициентов `coef`, степень полинома и значение переменной. Длина массива `coef` равна `n`+1;
3. `power` – функция вычисления `x` в степени `n`.

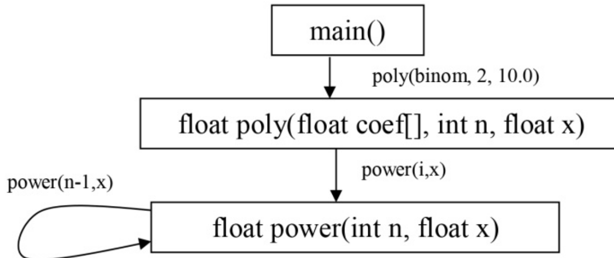
8.3.3. Рекурсия

Особенностью функции `power` в рассматриваемой программе является то, что при некоторых условиях она может вызывать сама себя, что соответствует рекуррентному математическому определению степенной функции:

$$x^n = \begin{cases} 1, & \text{если } n = 0 \\ x * x^{n-1}, & \text{иначе} \end{cases}$$

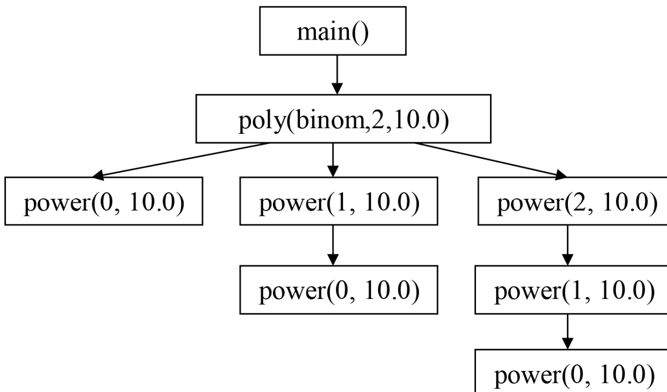
Такие функции называются *рекурсивными*. Это определение может быть истолковано разными способами и требует более формального подхода. Рассмотрим *граф вызовов* программы. Вершинами этого графа являются функции, а дуга проводится от одной функции к

другой, если в первой есть вызов второй. Дуга помечается этим вызовом. Для рассматриваемой программы граф вызовов имеет вид:



Тогда функцию можно назвать рекурсивной, если в графе вызовов через эту функцию проходит хотя бы один цикл.

Однако такое определение является статическим, поскольку опирается исключительно на текст программы и не вполне отражает реальное исполнение. Действительно, может оказаться, что при конкретных данных данный цикл в графе вызовов не реализуется. Более того, может оказаться, что ни один цикл, проходящий через данную вершину, не реализуется ни при одном исполнении. Динамическое поведение программы отражает понятие *дерева вызовов*, вершинами которого являются выполненные в ходе исполнения программы вызовы с указанием значений фактических параметров, а дуга проводится, если один вызов привел к другому. Дуги упорядочены согласно порядку исполнения. Корнем дерева является вызов главной функции main. Для рассматриваемой программы дерево вызовов имеет следующий вид:



Тогда функцию можно считать рекурсивной, если она встречается по крайней мере дважды на некотором пути от корня к листу. *Глубиной рекурсии* для вызова некоторой функции называется количество вершин в дереве вызовов, соответствующих той же функции, на пути от данного вызова до корня. Например, глубина рекурсии последнего вызова функции `power` равна 3.

Рекурсия может приводить к зацикливанию программы и бесконечному дереву вызовов. Простейшим примером является функция

```
void infinite() { infinite(); }
```

Есть несколько условий, которые позволяют избежать зацикливания в функциях:

1. В функции должен быть хотя бы один путь, не приводящий к рекурсивному вызову. В случае функции `power` такой путь обеспечивается условием `n==0`;
2. Должна быть некоторая дискретная величина, которая монотонно убывает при каждом рекурсивном вызове и ограничена снизу. В случае `power` такой величиной является значение параметра `n`.

В общем случае не только бесконечность рекурсии, но и сама рекурсивность функции в «динамическом» смысле является алгоритмически неразрешимым свойством. Что же касается «статической», т. е. основанной на графе вызовов рекурсивности, то ее легко определить, но только в случае, если вызываемые функции всегда указываются явно своими именами. Иначе для каждого выражения, вычисляющего функцию, необходимо определить множество ее возможных значений, что, во-первых, требует глобального анализа программы и, во-вторых, в общем случае можно сделать лишь приблизительно.

Если происходит рекурсивный вызов функции, т. е. функция вызывается, когда предыдущий ее вызов еще не закончился, то согласно описанному методу исполнения вызова функции создаются новые локальные объекты, когда старые еще не удалены. В результате для каждого локального объекта возникают несколько одновременно существующих *поколений*, но каждый вызов «знает», с каким именно поколением он работает.

Рекурсия в языках программирования долгое время вызывала много споров, связанных прежде всего с неэффективностью ее

реализации. Во-первых, собственно вызов функции является дорогостоящей операцией, связанной с пересылкой значений фактических параметров в локальные объекты, организацией возврата управления и т. д. Во-вторых, для рекурсивных процедур требуется дополнительная память, размер которой пропорционален высоте дерева вызовов. Наконец, рекурсия сильно затрудняет статический анализ программ, что зачастую делает невозможными многие оптимизирующие преобразования.

С другой стороны, все эти недостатки перекрываются тем, что рекурсия позволяет естественно реализовать по существу рекурсивные алгоритмы, такие как полный перебор, метод ветвей и границ, метод «разделяй и властвуй» и др.

8.3.4. Реализация функций

Семантику функций мы будем описывать, используя *трансформационный* подход, постепенно преобразуя программу ко все более простому виду. На каждом шаге будут использоваться преобразования, смысл, применимость и корректность которых очевидна, что гарантирует, что семантика программы в целом не меняется.

На **первом шаге** мы упростим выражения, разбив их на последовательность операторов, возможно, с использованием вспомогательных переменных так, чтобы любой вызов функции, не являющейся процедурой, встречался только как левая часть присваивания

```
t = f(...);
```

Для этого может потребоваться, в частности, преобразовать условные выражения в условные операторы. Так, например, оператор

```
return n==0 ? 1 : x*power(n-1,x);
```

преобразуется в

```
if (n==0)
  return 1;
else
{
  float t = power(n-1,x);
  return x * t;
}
```

На **втором шаге** преобразуем все функции в процедуры. Для этого к каждой функции добавим дополнительный формальный параметр `result` типа указателя на исходный тип результата функции. Все операторы вида

```
return e;
```

заменим на присваивание переменной, на которую указывает `result` и `return`

```
* result = e;
return;
```

Присваивание, в котором правой частью является вызов функции

```
t = f(e1,...,en);
```

заменим на вызов процедуры, последним параметром которой будет адрес получателя исходного присваивания:

```
f(e1,...,en, &t);
```

В результате этого преобразования функция `power` примет следующий вид:

```
void power(int n, float x, float * result)
{
  if (n==0)
    *result = 1;
  else
  {
    float t;
    power(n-1,x,&t);
    *result = x * t;
  }
}
```

На **третьем шаге** мы добьемся того, что все процедуры будут иметь единственный параметр – указатель на структуру, называемую *фреймом*, в которой собраны все формальные параметры исходной процедуры и локальные переменные тела функции. Для каждой функции будет свой тип фрейма. Позже мы в эту структуру добавим дополнительную информацию. Фрейм создается перед вызовом процедуры и в него присваиваются значения, соответст-

вующие фактическим параметрам. В теле процедуры обращения к локальным объектам заменяются на обращения к полям фрейма. Наконец, по завершении вызова процедуры фрейм удаляется.

Для создания и удаления фрейма будем использовать специальные конструкции `frame_new` и `frame_dispose`, реализацию которых обсудим ниже.

Таким образом, процедура `power` приобретает следующий вид:

```
struct frame_power
{
   int n;
   float x;
   float * result;
   float t;
}
void power(struct frame_power *f)
{
   if (f->n==0)
      *(f->res) = 1;
   else
   {
      struct frame_power *a;
      frame_new(a);
      a->n = f->n-1;
      a->x = f->x;
      a->result = &(f->t);
      power(a);
      frame_dispose(a);
      *(f->result) = f->x * f->t;
   }
}
```

Головную функцию `main` оставим вне рассмотрения – она требует специальной обработки, поскольку мы не можем менять ее тип.

На **четвертом шаге** мы избавимся от параметров вообще. Заметим, что к этому моменту у каждой процедуры есть доступ только к одному фрейму. Заведем глобальную переменную `f`, которая будет хранить

ссылку на текущий фрейм. Поскольку фреймы бывают разных типов, то переменная `f` будет описана просто как указатель

```
void * f;
```

а чтобы при каждом использовании `f` не делать явного приведения типов, заведем также глобальные типизированные указательные переменные для каждого типа фрейма – `f_power`, `f_poly` и т. д., – в которые будем присваивать значение `f` в начале соответствующей процедуры.

В каждом фрейме будем сохранять указатель на фрейм вызвавшей процедуры, который тоже должен быть описан как нетипизированный, поскольку данная процедура может вызываться из разных мест. Переменной `f` присваивается новое значение перед вызовом процедуры и восстанавливается старое после вызова

```
struct frame_power
{
  void * parent;
  int n;
  float x;
  loat * result;
  float t;
} *f_power;

void power()
{
  f_power = f;
  if (f_power->n==0)
    *( f_power->res) = 1;
  else
  {
    struct frame_power *a;
    frame_new(a);
    a->n = (f_power->n)-1;
    a->x = f_power->x;
    a->res = &(f_power->t);
    a->parent = f;
    f = a;
    power();
    f = a->parent;
    frame_dispose(a);
    *(f_power->result) = f_power->x * f_power->t;
  }
}
```

На последнем, **пятом шаге,** мы избавимся от процедур вообще. В результате получится одна большая процедура main, в которую будут включены тела всех процедур. К настоящему моменту из всех действий, связанных с вызовом процедуры, остался только переход к выполнению тела процедуры, что может быть реализовано оператором goto на метку, которую поставим в начале тела процедуры.

Проблема только в реализации возврата, т. е. в том, куда передать управление при достижении конца процедуры или оператора return. Для этого поставим после каждого вызова уникальную метку и запомним перед переходом к выполнению тела процедуры указатель на нее в еще одном дополнительном поле фрейма. Тогда оператор return можно заметить на переход по вычисляемой метке goto*.

```
void * f;

struct frame_power
{
   void * f_parent;
   void * return_label;
   int n;
   float x;
   float * res;
   float t;
} * f_power;

struct frame_poly
{
   ...
} * f_poly;

void main()
{
  ...
poly:
  f_poly = f;
  ...
```

```
  for (...)
  {
    ...
    struct frame_power *a;
    frame_new(a);
    a->n = f_poly->i;
    a->x = f_poly->x;
    a->res = &(f_poly->t);
    a->parent = f;
    a->return_label = &&l_power1;
    f = a;
    goto power;
l_power1:
    (f_poly)->sum += (f_poly->coef)[i] * f_poly->t;
  }
  ...

power:
  f_power = f;
  if (f->n==0)
    *(f->res) = 1;
  else
  {
    struct frame_power *a;
    frame_new(a);
    a->n = (f_power->n)-1;
    a->x = f_power->x;
    a->res = &(f_power->t);
    a->parent = f;
    a->return_label = && l_power2;
    f = a;
    goto power;
l_power2:
    f = a->parent;
    frame_dispose(a);
    * (f_power->res) = f_power->x * f_power->t;
  }
  goto * (f_power->return_label);
}
```

Если у процедуры есть единственный вызов (и других никогда не будет), то можно не хранить метку и переход по вычисляемой метке заменить не безусловный переход в точку возврата.

Если теперь нужно вообще избавиться от переходов по вычисляемой метке и вернуться к стандарту С, то можно пронумеровать все возможные точки возврата:

```
enum return_power
{
    e_power1,
    e_power2,
}
```

во фрейме вместо указателя на метку хранить значение этого типа,

```
struct frame_power
{
   ...
   enum return_power return_label;
...
}
...
    a->return_label = e_power1;
...
    a->return_label = e_power2;
...
```

а переход по вычисляемой метке заменить на переключатель:

```
switch (f_power->return_label)
{
  case e_power1: goto l_power1;
  case e_power1: goto l_power1;
}
```

Таким образом мы преобразовали рекурсивную программу в итеративную, сведя тем самым семантику процедур к базовым управляющим конструкциям. Отметим следующие аспекты проделанных преобразований:

1. Все преобразования носят универсальный характер, т. е. не используют знаний о том, что именно делает программа. Это значит, что мы можем проделать то же самое с любой другой программой.
2. Полученная программа гораздо сложнее воспринимается и анализируется, нежели исходная. Но именно так нам пришлось бы программировать, если бы в языке не было бы рекурсивных процедур,

а решаемая задача была бы существенно [29] рекурсивной, что еще раз свидетельствует о достоинствах рекурсивных процедур, по крайней мере для определенного класса задач.

8.3.5. Хвостовая рекурсия

Преобразования не являются оптимизирующими, т. е. полученная программа выполняет те же действия, что и исходная. Более того, вполне возможно, что умный транслятор сможет реализовать исходную программу даже более эффективно, чем полученную, за счет использования специальных машинных инструкций для доступа к локальным объектам и возврата из процедур, а также ввиду неспособности анализировать сложное управление с использованием переходов по вычисляемым меткам.

Некоторые программы могут быть переведены в итеративную форму, не требующую создания многих поколений локальных объектов. В частности, это возможно, если все процедуры используют только так называемую *хвостовую рекурсию*, при которой любой вызов процедуры является последним действием вызывающей процедуры, не имеющим доступа к локальным объектам вызывающей процедуры. В некоторых случаях процедура может быть преобразована к хвостовой рекурсии путем введения *накопительных* параметров. Рассмотрим этот прием на примере той же функции возведения в степень, которая уже была предварительно преобразована в процедуру:

```
void power(int n, float x, float * result)
{
  if (n==0)
    *result = 1;
  else
  {
    float t;
    power(n-1,x,&t);
    *result = x * t;
  }
}
```

[29] Существенность здесь понимается неформально, в том смысле, что исходная задача гораздо проще формулируется с помощью рекуррентных соотношений, но не как невозможность реализовать задачу без использования рекурсии.

Заметим, что она не удовлетворяет требованию хвостовой рекурсии, поскольку, во-первых, после вызова `power` еще выполняется присваивание результату, и, во-вторых, вызову передается ссылка на локальную переменную `t`, которая внутри вызова может быть изменена и на самом деле меняется присваиванием `*result`.

Заведем дополнительный параметр `total`, который будет накапливать то значение, которое нужно выдать в качестве результата по завершению рекурсии. При первом вызове в качестве этого параметра должно быть передано значение 1. После этого станет ненужной локальная переменная `t`, поскольку результат сразу можно записывать по назначению:

```
void power(int n, float x, float total, float * result)
{
  if (n==0)
    *result = total;
  else
    power(n-1,x, x*total, result);
}
```

Свойства хвостовой рекурсии обеспечивают возможность не создавать новый фрейм перед рекурсивным вызовом, а использовать текущий, изменив в нем поля, соответствующие формальным параметрам, на значения фактических, и передать управление на начало процедуры. Кроме этого, формальный параметр `total` можно сделать локальной переменной с соответствующей инициализацией:

```
void power(int n, float x, float * result)
{
  float total = 1;
entry:
  if (n==0)
    *result = total;
  else
  {
    n = n-1;
    total = x * total;
    goto entry;
  }
}
```

Легко заметить, что тело цикла можно преобразовать в цикл for:

```
void power(int n, float x, float * result)
{
  float total;
  for (float = 1; n != 0; total*=x, n--);
  *result = total;
}
```

8.3.6. Вложенные процедуры

Вернемся к программе вычисления полинома. Можно заметить, что параметр x, передаваемый в вызове power в функции poly дальше передается без изменений в рекурсивные вызовы. В результате все поколения переменной x имеют одно и то же неизменяемое значение. Оно копируется в каждом фрейме, что требует как времени, так и памяти. Однако функция power не может обратиться непосредственно к параметру x функции poly ввиду ограничения на видимость объектов. Конечно, можно было бы сделать переменную x глобальной, но тогда она будет существовать все время выполнения программы, и ее смогут изменить и другие процедуры. Выход из этого положения дают *вложенные процедуры*, которые имеются во многих языках программирования, например, в Паскале и в некоторых расширениях языка С. В данном примере описание функции power помещается внутрь описания функции poly:

```
float poly(float coef[], n,float x)
{
  float power(int n)
  {
    return n==0?1:x*power(n-1);
  }
  float sum = 0f;
  for (int i=0; i<=n; i++)
    sum += coef[i]*power(i);
  return sum;
}
```

Согласно правилам видимости теперь использование x внутри power обращается объекту в ближайшем охватывающем блоке, в котором описано x, т. е. к параметру функции poly. Конечно, в результате функция power становится недоступной вне функции poly.

На первый взгляд, мы добились желаемого результата, однако побочным эффектом такого преобразования является то, что становится неверным предположение о том, что любая функция имеет доступ только к глобальным переменным и своим локальным объектам. Теперь при обращении к `x` внутри `power` необходим текущий фрейм функции `poly`. Это можно сделать разными способами. Например, можно в каждом фрейме в качестве первого поля завести признак, определяющий функцию, к которой он относится, и в тот момент, когда потребовалась переменная или параметр охватывающей процедуры, пробежаться по ссылкам на охватывающий фрейм до тех пор, пока не будет найден требуемый. Понятно, что сложность доступа при такой реализации может быть пропорциональна глубине рекурсии, что во многих случаях неприемлемо. Другой подход заключается в том, чтобы для каждой процедуры поддерживать ссылку на фрейм, содержащий последнее поколение локальных объектов. Так или иначе, вложенные процедуры облегчают собственно вызов процедуры, но достаточно существенно затрудняют доступ к локальным объектам.

К достоинствам вложенных процедур можно отнести и то, что они облегчают процесс *свертки*. Допустим, мы обнаружили, что некоторый фрагмент кода нужно выполнить еще в нескольких местах и для этого его нужно оформить в виде процедуры. Если язык не поддерживает вложенных процедур, то все локальные объекты, которые используются в данном фрагменте кода, придется сделать формальными параметрами и передавать их при вызове, а если их достаточно много, то это опять же скажется на эффективности. Большинство современных языков программирования в том или ином виде поддерживают вложенность процедур и функций.

8.3.7. Оптимизации

Если уж мы заговорили об оптимизации, то рассмотрим наш пример на основе анализа сложности вычислений. Использованные ниже методы достаточно характерны при оптимизации программ (подробнее см. Раздел 11). Будем оценивать сложность в количестве выполненных операций умножения. В тексте она встречается дважды: один раз в функции `poly` и один – в `power`. Поскольку все управление в обеих функциях полностью определяется параметром `n`, мы можем сопоставить каждой функции другую функцию, которая вычисляет сложность: T_{poly} и T_{power}.

Для исходной программы с рекурсивной реализацией `power` несложно показать, что

$$T_{\text{poly}}(n) = \sum_{i=0}^{n-1} (1 + T_{\text{power}}(i))$$

$$T_{\text{power}}(n) = \begin{cases} 0, & \text{если } n = 0 \\ 1 + T_{\text{power}}(n), & \text{иначе} \end{cases}.$$

Очевидно, что $T_{\text{power}}(n) = n$ и, следовательно,

$$T_{\text{poly}}(n) = \sum_{i=0}^{n-1} (1 + i) = \frac{n(n+1)}{2}.$$

Что касается емкостной сложности, то она определяется максимальной глубиной в дереве вызова, которая равна n. Преобразование функции `power` путем приведения ее к хвостовой рекурсии не изменяет количество операций, но уменьшает емкостную сложность до константы.

Большинство умножений происходит в функции `power`. Следуя принципу оптимизации наиболее горячих точек, займемся ею в первую очередь. Можно попытаться реализовать функцию `power` за счет использования стандартных функций, например, как

```
float power(int n, float x)
{
  return exp(log(x)*i);
}
```

но так мы просто прячем от себя сложность, оставляя в функции единственное умножение, игнорируя при этом неизвестное и явно не малое количество умножений, выполняемых при вычислении экспоненты и логарифма.

Мы уже рассматривали способ возведения в степень, который может быть записан как

```
float power(int n, float x)
{
  float y = 1;
  while (n) n&1 ? (y*=x,--n) : (x*=x,n/=2);
  return y;
}
```

Оставим в качестве упражнения доказательство с использованием аксиоматической семантики того, что количество умножений здесь не превосходит 2*log(n+1), и тогда

$$T_{\text{poly}}(n) \le \sum_{i=0}^{n-1} (1 + 2\log(i+1)).$$

Сумму логарифмов можно оценить, используя формулу Стирлинга

$$n! \sim \sqrt[2]{2\pi n}\left(\frac{n}{e}\right)^{n},$$

то есть

$$\log n! \sim \log \sqrt[2]{2\pi n}\left(\frac{n}{e}\right)^{n} = \frac{\log 2 + \log \pi + \log n}{2} + n(\log n - \log e)$$

и асимптотически

$$T_{\text{poly}}(n) \le 2\, n \log n + 1.$$

Теперь заметим, что на каждой итерации цикла в функции `poly` результат функции `power` отличается от предыдущего в x раз. Это дает возможность вообще исключить функцию `power`, введя вместо нее вспомогательную переменную:

```
float poly(float coef[], int n, float x)
{
   float sum = 0f;
   float power;
   int i;
   for (i=0,power=1f; i<=n; power*=x,i++)
      sum += coef[i] * power;
   return sum;
}
```

Это приводит к тому, что количество умножений становится равно

$$T_{\text{poly}}(n) = 2\, n.$$

Наконец, вспоминая из школьной программы схему Горнера, мы можем реализовать вычисление полинома как

```
float poly(float coef[], int n, float x)
{
   float sum = coef[n];
   for (i=n-1; i>=0; i--)
      sum = sum * x + coef[i];
   return sum;
}
```

где требуется лишь n умножений.

Таким образом, наибольший эффект достигается использованием эффективных методов. Однако и систематическая оптимизация, такая как нахождение индуктивных переменных, замена рекурсивных программ итеративными и т. д. может привести к существенному повышению эффективности. Нельзя рассчитывать на то, что все это за нас сделает транслятор.

8.3.8. Функциональные значения

Как мы уже говорили, указатели на функции являются равноправными объектами в языке С. В частности, их можно передавать параметром другим функциям. В таком случае передаваемые функции называются *функциями обратного вызова* (*call-back*). Продемонстрируем это понятие на примере алгоритма сортировки массива. В стандартной библиотеке языка С имеется функция `qsort`, описанная в стандартном включаемом файле `stdlib.h` как

```
void qsort(void * base,
           size_t num, size_t size,
           int (*cmp) (void *,void *));
```

Реализованный в ней алгоритм сортировки не зависит природы элементов. Первые три параметра задают сортируемый массив: указатель на начало массива `base`, количество элементов `num` и их размер `size`. Последний параметр `cmp` задает функцию сравнения элементов, которая согласно принятым соглашениям вырабатывает положительное значение, если первый аргумент больше второго, отрицательное – если меньше, и ноль – если они равны.

Пусть, например, задан двумерный массив размера N×N:

```
#define N 1000
float M[N][N];
```

и нам требуется отсортировать его строки довольно специфическим образом, сравнивая их по длине задаваемых ими N-мерных векторов.

Для этого достаточно описать функцию veccmp следующим образом:

```
float veclen(float x[])
{
  float sum = 0;
  for (int i=0; i<N; i++)
    sum += x[i] * x[i];
  return sqrt(sum);
}
int veccmp(void *x; void *y)
{
  float v = veclen((float*) x) - veclen((float*) y);
  return (v==0 ? 0 : v>0 ? 1 : -1);
}
```

Здесь нам пришлось преодолеть типовый контроль, во-первых, описав параметры функции как void*, чтобы ее тип совпал с тем, который требует qsort, и, во-вторых, внутри функции привести универсальные указатели к конкретному типу float*.

Если теперь мы вызовем

```
qsort(M, N, N*sizeof(float), &veccmp);
```

то в ходе своего выполнения qsort (многократно) вызовет нашу функцию veccmp. Отсюда и название – обратный вызов.

Функции можно не только передавать в качестве параметров, но и выдавать в качестве результата. В следующем примере функция get_binop преобразует код операции в указатель на реализующую ее функцию:

```
float sum(float x, float y) { return x+y; }
float sub(float x, float y) { return x-y; }
float mult(float x, float y) { return x*y; }
float div(float x, float y) { return x/y; }
typedef float (operation *)(float,float);
operation get_binop(char code)
{
  switch (code)
  {
    case '+' : return &sum;
    case '-' : return &sub;
    case '*' : return &mult;
    case '/' : return &div;
  }
}
```

и может быть применена как

```
((*get_binop)('+'))(7,8)
```

Если возвращаемое функциональное значение ссылается на вложенную функцию, то может возникнуть проблема с несоответствием времени жизни объектов. Рассмотрим, например, функцию

```
typedef float (* unary_operation)(float);
unary_operation get_incr(float k)
{
  float plus(float x)
  {
    return x + k;
  }
  return &plus;
}
```

которая, будучи применена к аргументу `k`, должна возвращать функцию, которая, будучи применена к `x`, возвращает `x+k`. т. е. вызывать мы ее должны как, например,

```
(*get_incr(7))(8)
```

Проблема здесь заключается в том, что локальный объект, соответствующий формальному параметру `k`, удалится после завершения вызова процедуры `get_incr`, и когда дело дойдет до вызова функции `plus`, он уже не будет существовать, хотя и используется внутри `plus`. Поэтому чаще всего языки, допускающие вложенные процедуры, считают такие трюки некорректными. Обобщая, можно сказать, что некорректным будет возвращение в качестве результата либо присваивание в нелокальную переменную любой ссылки на локальный объект, будь то переменная, процедура или метка.

8.3.9. Подстановка параметров

В этом разделе мы рассмотрим несколько вариаций на тему подстановки параметров. В языке C единственным способом подстановки параметров является *подстановка по значению*, которая заключается в том, что значения фактических параметров копируются в локальные объекты, над которыми выполняется тело функции. Такая семантика может иногда приводить к неожиданным результа-

там. Рассмотрим, например, процедуру, которая обменивает значениями две вещественные переменные:

```
void  swap(float x, float y)
{
   x += y; y = x - y; x -= y;
}
```

Здесь мы проявили смекалку и оптимизировали процедуру [30], сэкономив вспомогательную переменную, по сравнению с

```
float tmp = x; x = y; y = tmp;
```

Ожидается, что если имеются переменные A и B с текущими значениям 1 и 2 соответственно,

```
float A = 1, B = 2;
```

то после вызова

```
swap(A,B);
```

значениями станут 2 и 1. Однако на самом деле A и B останутся неизменными именно ввиду того, что все операции внутри тела swap выполнялись над локальными объектами x и y, которые после того, как им были присвоены начальные значения, не имеют никакого отношения к A и B.

На первый взгляд получается, что при передаче параметров по значению единственным способом осуществить побочный эффект является изменение глобальных переменных. Нам необходимо, чтобы в ходе выполнения swap переменные x и y означали A и B соответственно, и любое присваивание, скажем, x означало присваивание A. Такой способ передачи параметров называется *по ссылке*. В языке С он может быть реализован следующим образом:

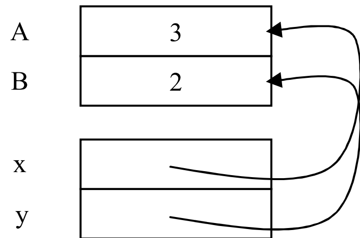
```
void  swap(float * x, float * y)
{
  *x += *y; *y = *x-*y; *x -= *y;
}
```

[30] Это пример очень плохой оптимизации, и так делать **не следует**. Во-первых, мы использовали три дополнительные операции типа сложения, а, во-вторых, что более существенно, эта процедура будет работать неправильно в случае потери точности при арифметических операциях, например, когда x – очень большое, а y – очень маленькое.

а при вызове <u>по значению передавать указатель</u> на объект:

```
swap(&A,&B);
```

Состояние после `*x += *y;` отражено на следующем рисунке:



Указанный метод практически в точности отражает семантику передачи параметров по ссылке, за исключением того факта, что значение формального параметра, скажем `x`, может быть изменено в процессе выполнения функции, и он будет указывать уже на другой объект. Кроме того, обилие операций разыменования явно не улучшает читаемость программы.

Некоторые языки программирования – Паскаль, Visual Basic, C++ и др. – поддерживают как передачу параметров по значению, так и по ссылке, причем последний способ является умолчательным. В языке Фортран параметры всегда передаются по ссылке, поскольку в нем типична передача в качестве параметров больших массивов [31]. При передаче по значению это потребовало бы больших накладных расходов как по памяти, так и по времени.

Вернемся к примеру с функцией `swap`. Рассмотрим случай, когда в качестве параметра передается один и тот же объект:

```
swap(&A, &A)
```

Естественно ожидать, что значение `A` после вызова должно остаться без изменения. Однако на самом деле оно станет равным нулю, поскольку присваивание

```
*y = *y - *x;
```

при таком вызове эквивалентно

```
A = A - A;
```

поскольку `*x`, `*y` и `A` обозначают один и тот же объект, т. е. являются *синонимами*. Конечно, можно возразить, что это является особым случаем, и ни в одной реальной программе такого не будет, однако

[31] Именно массивов, а не ссылок на первый элемент массива, как в языке С.

проблема не так проста. Например, если M – массив вещественных, то при вызове

```
swap(&(M[i]), &(M[j]));
```

необходимо показать, что i не равно j при любом исполнении программы, что в общем случае алгоритмически неразрешимо. Наличие синонимов значительно усложняет понимание программы и может приводить к труднообнаруживаемым ошибкам. Также это может влиять на эффективность программ, поскольку транслятору приходится предполагать, что параметр может указывать куда угодно, и, как следствие, ограничивать применение некоторых оптимизаций.

Избежать синонимичности позволяет передача параметров *по значению-результату*, которая является своего рода комбинацией передачи параметров по значению и по ссылке. Точно так же, как при передаче параметров по ссылке, процедуре передаются адреса объектов, но их значения тут же копируются в локальные объекты, а по завершению процедуры значения из локальных объектов копируются обратно по указанным адресам. Это может быть реализовано на языке C следующим образом:

```
void swap(float * _x, float * _y)
{
   float x = * _x, y = * _y;
   x += y; y = x-y; x -= y;
   *_x = x; *y = _y;
}
```

Таким образом, в ходе выполнения процедуры локальные объекты не являются синонимами объектов, переданных в качестве фактического параметра. Очевидным недостатком передачи параметров по значению-результату является использование дополнительной памяти и пересылок в начале и конце исполнения вызова.

Рассмотрим теперь другой аспект подстановки параметров, связанный с тем, что значения фактических параметров всегда вычисляются до исполнения тела процедуры. На самом деле это не всегда удобно, поскольку некоторые из полученных значений могут быть востребованы только при определенных условиях, которые в общем случае зависят от других параметров.

Рассмотрим, например, функцию, которая вычисляет квадрат либо второго, либо третьего аргумента в зависимости от неко-

торого условия, значение которого передается первым аргументом:

```
float if_sqr(int cond, float t, float f)
{
  if (cond)
    return t * t;
  else
    return f * f;
}
```

Теперь эта функция может быть вызвана как

```
if_sqr(x==0, 0, 1/x)
```

Ожидается, что при `x` равном нулю функция вернет ноль, однако это не так, поскольку согласно методу передачи параметров по значению сначала вычисляются все три значения фактических параметров, включая `1/x`, что приведет к исключительной ситуации.

Отложить вычисление можно, если передавать не значение, а функцию, вычисляющую это значение. Такой способ называется передачей параметров *по необходимости*. Сразу скажем, что это имеет смысл только для так называемых *нестрогих* параметров, т. е. тех, значения которых могут не потребоваться. В противном случае параметр называется *строгим*. В приведенном выше примере параметр `cond` является строгим, а параметры `t` и `f` – нестрогие. В языке С он может быть реализован следующим образом:

```
float if_sqr(int cond, float *t(), float *f())
{
  if (cond)
    return (*t)() * (*t)();
  else
    return (*f)() * (*f)();
}
float t_thunk() {  return 0; }
float f_thunk() { return 1/x; }
```

Теперь при вызове

```
if_sqr(x==0, &t_thunk, &f_thunk)
```

вычисление последних двух параметров не будет приводить к вызову, а только к выдаче указателей на функции. Функция

без параметров, используемая для этих целей, называется *thunk* [32].

Фундаментальным достоинством передачи параметров по необходимости является то, что такой вызов всегда выдает ответ, если он существует, в отличие от передачи параметров по значению, где вычисление «ненужного» параметра может привести к зацикливанию или исключительной ситуации.

Что касается эффективности, то она зависит от конкретного случая. Передача параметров по необходимости может повысить эффективность, если фактическим параметром является сложное выражение, которое будет вычисляться только тогда, когда его значение используется при данном вызове. И наоборот, если фактический параметр простой, то эффективность ухудшится, поскольку вместо простого обращения к формальному параметру будет происходить вызов функции, что является дорогостоящей операцией. Ситуация может ухудшится кардинально, если к формальному параметру обращаются многократно. Рассмотрим, например, функцию, которая возводит число в куб:

```
float power3(x)
{
  return x * x * x;
}
```

и ее вызов

```
power3(power3(power3(sqrt(a)))
```

Очевидно, что при передаче параметров по значению функция `sqrt` будет вызвана один раз. Если же параметр `x` передавать по необходимости, то реализацией функции будет

```
float power(float * x())
{
  return (*x)() * (*x)() * (*x)();
}
```

а вызов будет иметь вид

```
power3(t2_thunk)
```

[32] Слово thunk является субстандартной или диалектной совершенной формой глагола think (думать).

при определении дополнительных функций

```
float sqrt_thunk() { return sqrt(x*x+1); }
float t1_thunk() { return power3(sqrt_thunk); }
float t2_thunk() { return power3(t1_thunk); }
```

для каждого фактического параметра. Теперь при головном вызове `power3` трижды вызовется функция `t2_thunk`, каждый вызов которой трижды вызовет `t1_thunk`, каждый вызов которой в свою очередь трижды вызовет `sqrt_thunk`. В итоге `sqrt` вызовется 27 раз! Дублирование вычислений, происходящее при передаче параметров по необходимости, может приводить к экспоненциальному снижению эффективности. Это особенно неприятно, если вычисление фактического параметра имеет побочный эффект.

В языке Алгол-60 помимо передачи параметров по значению имеется передача параметров *по имени*, которая на словах очень проста: тело функции выполняется так, как если бы везде вместо формального параметра был написан текст фактического параметра, т. е. (в нотации языка C) если задана функция

```
float sum_i(float x)
{
  float sum = 0;
  for (i=0; i<N; i++)
    sum += x;
  return sum;
}
```

то вызов

```
sum_i(A[i]*B[i])
```

должен быть эквивалентен вызову функции

```
float sum_i_AiBi()
{
  float sum = 0;
  for (i=0; i<N; i++)
    sum += A[i]*B[i];
  return sum;
}
```

Такой способ реализации очень похож на макрообработку, но в отличие от нее лучше согласован с синтаксисом и семантикой языка. Копировать тело процедуры для каждого вызова – чрезвычайно расточительно. Реализация передачи параметров по имени в Алгол-60 сходна с рассмотренной выше передачей параметров по необходимости: для каждого фактического параметра транслятор порождает процедуру без параметров, передаваемую в качестве параметра. Разница заключается в том, что при передаче параметров по имени фактический параметр попадает в другой контекст. В приведенном выше примере все использованные в вызове переменные – `A`, `B` и `i` – идентифицируются не в месте вызова, а в месте вхождения формального параметра в теле процедуры, что достаточно сложно выразить в терминах языка С [33].

Все рассмотренные выше способы передачи параметров предполагают, что количества формальных и фактических параметров совпадают. Исключением из этого правила являются функции с переменным числом параметров. Описание параметров таких процедур заканчивается многоточием. Типичным примером является стандартная функция `printf`, специфицированная как

```
int printf(char *format, ...)
```

вызов которой может иметь вид

```
printf("%d %c %d = %d",
        x, op, y,
        get_binop(op)(x,y));
```

[33] Надо признать, что аналогичная проблема возникает и при передаче параметров по необходимости: формировать thunk нужно ровно в том месте, где находится фактический параметр, поскольку в противном случае он не будет иметь доступа к использованным локальным переменным. Приведенные в обсуждении примеры корректны только в предположении, что в фактическом параметре встречаются только глобальные переменные.

Возможность передавать произвольное количество параметров не укладывается в ту схему реализации процедур, которую мы рассматривали выше, поскольку неизвестен размер фрейма и типы передаваемых дополнительных параметров. Ее можно развить, формируя фрейм не по описанию процедуры, а по конкретному вызову, что приведет к тому, что для разных вызовов структура фрейма будет различна. Общее в них только то, что все дополнительные параметры размещаются последовательно, возможно с выравниванием, вслед за последним явно указанным параметром. Следовательно, и доступ внутри процедуры к дополнительным параметрам можно осуществить только с помощью адресной арифметики и приведения типов.

Для корректной работы с указателями во фрейме используется тип `va_list` и набор макросов, определенных во включаемом файле `stdarg.h`. Реализация `va_list` может существенно меняться в зависимости от конкретной системы пограммирования. Можно предположить, что она помимо прочего содержит указатель на текущий дополнительный параметр. Инициализация переменной типа `va_list` производится с помощью макроса `va_start`, которому должен быть передан последний из явно указанных формальных параметров. Зная тип этого параметра и то, как именно размещаются параметры во фрейме, `va_start` устанавливает указатель на первый дополнительный параметр. Выбор значения текущего параметра и переход к следующему параметру совмещен в макросе `va_arg`, которому надо указать переменную типа `va_list` и тип параметра. Наконец, когда обработка всех дополнительных параметров закончена, нужно вызвать макрос `va_end` на тот случай, если `va_start` запросил ресурсы, которые теперь требуется освободить.

Рассмотрим это на примере реализации упрощенной версии `printf`, которую назовем `my_printf`. Будем считать, что уже реализованы процедуры печати строки и чисел:

```
void print_string(string s);
void print_int(int i);
void print_float(float v);
```

Тогда функцию `my_printf` можно реализовать следующим образом:

```
#include <stdarg.h>
void my_printf(char * format, ...)
{
  va_list    argptr;
  char * f; // указатель на очередной символ в формате.
  va_start(argptr, format);
  for (f = format; *f; f++)
    if (f[0]=='%'&& f[1])
    {
      switch(f[1])
      {
        case 's' :
          print_string(va_arg(argptr,char *));
          break;
        case 'd' :
          print_int(va_arg(argptr,int));
          break;
        case 'f' :
          print_float(va_arg(argptr,float));
          break;
        default :
          print_char(f[1]);
      }
      f++;
    }
    else
      print_char(f[0]);
    va_end(argptr);
}
```

и вызвать ее, например, как

```
my_printf("%d: Hello, %s!", cnt++, UserName);
```

Использование макросов, адресной арифметики и приведения типов делает возможными все характерные для них ошибки, которые мы уже обсуждали ранее. В данном примере это проявляется следую-

щим образом. Во-первых, невозможно проверить, что количество формальных параметров соответствует формату. То есть при вызове

```
my_printf("%f + %f = %f", x, y);
```

при обработке последнего %f произойдет обращение к памяти, находящейся вне фрейма и распечатается непредсказуемое значение. Во-вторых, невозможно проверить, что элементам формата соответствуют параметры нужного типа. Например, при вызове

```
my_printf("%s + %s = ?", UserName, 0.7L);
```

обработка второго %s приведет вещественное значение 0.7L к типу указателя и попытается применить к нему функцию print_string опять же с непредсказуемым результатом. И проблема здесь не в том, что мы как-то плохо реализовали нашу функцию – стандартный printf страдает теми же проблемами. На этом фоне жалоба на то, что невозможно описать функцию с переменным числом параметров, если у нее нет хотя бы одного явного параметра, кажется капризом.

В некоторых языках программирования эти проблемы решаются тем, что все дополнительные параметры собираются в массив. Так, например, на языке C# можно описать функцию, вычисляющую среднее переданных вещественных параметров, следующим образом:

```
float average(params float[] A)
{
  if (A.length == 0)
    return 0.0;
  float sum = 0.0;
  for (int i =0; i<A.length; i++)
    sum += A[i];
  return sum/A.length;
}
```

Эта функция может быть вызвана как

```
average()
average(1, 2.5, 3.7)
```

причем транслятор позаботится о том, чтобы выполнить приведение фактических параметров к нужному типу, например, 1 к 1.0.

Этого не вполне достаточно, чтобы описать `printf`, где параметры могут иметь разный тип, но оставшиеся проблемы уже не связаны собственно с передачей параметров.

Рассмотрение вопросов, связанных с передачей параметров, завершим *именованными* и *необязательными* фактическими параметрами, которых **нет** в языке С. Пусть нам нужно описать процедуру, рисующую на экране прямоугольник. Очевидно, что достаточно знать его ширину `witdh` и высоту `height`, а также координаты его левого верхнего угла (`left`, `top`). Таким образом, процедура может быть определена как

```
void draw_box(int left, int top, int width,
int height);
```

При более детальном анализе оказывается, что этих параметров недостаточно, и в общем случае требуется также указать:

- `x_offset` и `y_offset` – смещение окна (экрана) относительно логической плоскости рисования;
- `border_width` и `border_color` – ширина границы прямоугольника и ее цвет;
- `fill` и `fill_color` – признак того, что надо закрашивать внутренность прямоугольника, и цвет заливки;
- `transparency` – степень прозрачности при рисовании.

В результате спецификация процедуры становится следующей:

```
void draw_box(int left,
              int top,
              int width,
              int height,
              int x_offset,
              int y_offset,
              int border_width,
              int border_color,
              int fill,
              int fill_color,
              int transparency);
```

Теперь типичный вызов этой процедуры будет выглядеть так:

```
draw_box(100, 200, 50, 100, 0, 0, 1, 0, 1, 16777215, 50);
```

Глядя на такой вызов, достаточно трудно сразу понять, чему соответствует, например, первый параметр, равный 1. Конечно, современные системы программирования могут показать подсказку, если подвести курсор к нужному месту, но это требует дополнительных усилий.

Некоторые языки программирования (например, C#) решают эту проблему, позволяя задать умолчательные значения для некоторых параметров, которые будут использоваться в случае, если они не указаны в вызове. Если бы такая возможность была в языке C, то можно было бы описать процедуру как

```
void draw_box(int left,
              int top,
              int width,
              int height,
              int x_offset = 0,
              int y_offset = 0,
              int border_width = 1,
              int border_color = 0,
              int fill = 0,
              int fill_color = 0xFFF,
              int transparency = 0);
```

и в большинстве случаев указывать только четыре первых параметра. Недостатком такого подхода является то, что все параметры должны стоять на соответствующих позициях. Так что в данном случае, если нужно задать прозрачность, то придется указать и все остальные параметры.

Кардинальным решением является указание имен в вызове. Опять же, если бы такое было возможно в C, то по аналогии с C# можно было бы вызвать `draw_box` как

```
draw_box(width:50, height:100, left:100, top:200, transparency:128);
```

причем транслятору не составит труда расставить параметры в нужном порядке и проверить, заданы ли все параметры, для

которых нет умолчаний. Конечно, именованные фактические параметры тоже не всегда уместны и бывает удобнее позиционная запись либо комбинация этих способов.

8.4. Обработка исключительных ситуаций

Мы уже приводили пример, связанный с циклами, когда необходимо «экстренно» завершить выполнение нескольких вложенных конструкций и передать управление в точку продолжения. Проблема становится более трудной при наличии процедур и рекурсии. Продемонстрируем это на примере функции вычисления выражений. Напомним, что мы определили структуру внутреннего представления для простого языка выражений следующим образом:

```
struct Expr {
  int tag;
  enum ExprCode code;
  union {
    float value;
    char name[8];
    struct {
      char op;
      struct Expr * arg;
    } unop;
    struct {
      char op;
      struct Expr *left, *right;
    } binop;
  } choice;
};
```

Оставим пока в стороне вопрос о том, как порождается такая структура. Будем считать, что уже написана процедура

```
struct Expr * parse_expr(char * s);
```

преобразующая текстовое представление выражения в его внутреннее представление.

Процедура вычисления выражений

```
float eval_expr(struct Expr * e, Memory m);
```

получает на вход ссылку `e` на такую структуру и некоторый объект `m`, представаляющий память, над которой вычисляется выражение [34], и возвращает значение выражения.

Имея две такие функции, мы можем просто написать процедуру, реализующую *цикл «читать-вычислять-печатать»*, который по очереди вводит выражения и выдает их значения:

```
void read_eval_print(Memory m)
{
  char s[256];
  fputs("Привет!",stderr);
  for (;;)
  {
    fputc('>',stderr);
    fgets(s,stderr);
    if (s[0]=='.')
       break;
    fprintf(stderr, "%d\n",
            eval_expr(parse_expr(s), m));
    error_exit: ;
  }
  fputs("Пока.,stderr);
}
```

Все кажется просто до тех пор, пока не оказывается, что при вычислении выражения (как, впрочем, и при его разборе), может произойти ошибка. Например, при вычислении выражения

```
(1 + (2 * ((3/(2-(1+1))) - 4)))
```

произойдет деление 3/0. Чтобы наша программа вообще не прервалась, мы должны предусмотреть обработку исключительных

[34] Для дальнейших рассуждений нам неважно, как именно устроен этот объект и как, используя его, получить значение переменной, входящей в выражение.

ситуаций. Рассмотрим ветвь реализации `eval_expr`, связанную с делением:

```
float eval_expr(struct Expr * e, Memory m)
{
  swith (e->code)
  {
    ...
    case EC_BINOP:
    {
      float v1 = eval_expr(e->choice.binop.left, m);
      float v2 = eval_expr(e->choice.binop.right, m);
      switch (e->choice.binop.op)
      {
        case '/':
        {
          if (v2 == 0)
          {
            fputs("Деление на ноль.",stderr);
            goto error_exit;
          }
          return v1/v2;
        }
      }
      ...
    }
    ...
  }
}
```

Здесь перед выполнением деления мы проверяем, не равен ли делитель нулю, и если этот так, то выдаем сообщение и передаем управление в конец цикла «читать-вычислять-печатать», где предусмотрительно была поставлена метка `error_exit`.

Однако такой метод **некорректен**, поскольку язык C не позволяет передать управление на метку, находящуюся в другой функции.

Решение может быть связано с использованием вложенных процедур: мы могли бы поместить описания `parse_expr` и `eval_expr` внутрь функции `read_eval_print`. Однако это не всегда возможно, например, если эти процедуры описаны в другом файле или вообще не нами. К тому же в используемом диалекте языка C может не быть вложенных процедур. В таком случае нам придется переделать реализацию `eval_expr` так, чтобы она возвращала не только значение, но и признак того, что произошла ошибка. Поскольку ошибки бывают разных типов, то уместно завести тип перечисления, в котором они все указаны:

```
enum ErrorType
{
   OK = 0,
   ERR_SYNTAX,
   ERR_DIV0,
   ERR_OVERFLOW,
   ERR_UNDEFINED_VARIABLE,
   ...
}
```

Для функций, выдающих код завершения, принято, что 0 означает нормальное завершение, а ненулевые значения – коды ошибок. Это позволяет рассматривать их как логические значения.

Теперь функция eval_expr будет возвращать код завершения, а собственно вычисленное значение передаваться через параметр res:

```
enum ErrorType Eval(struct Expr * e, Memory m, float * res)
{
  swith (e->code)
  {
    ...
    case EC_BINOP:
    {
      float v1;
      float v2;
      enum ErrorType ec;
      if ((ec=eval_expr(e->choice.binop.left, m, &v1)) != OK)
        return ec;
      if ((ec=eval_expr(e->choice.binop.right, m, &v2)) != OK)
        return ec;
      switch (e->choice.binop.op)
      {
        case '/':
        {
          if (v2 == 0)
           return ERR_DIV0;
          * res = v1/v2;
        }
      }
      ...
    }
    ...
  }
  return OK;
}
```

Как видно, функция вообще не переходит к применению бинарной операции, если при вычислении одного из подвыражений произошла ошибка, а сразу возвращает ее код. Кроме того, мы вынесли из функции печать текста сообщения, чтобы делать это централизованно в месте вызова, что является хорошим стилем с точки зрения принципа отделения логики вычислений

от ввода/вывода. Функцию `read_eval_print` теперь следует переписать следующим образом:

```
void read_eval_print()
{
  char s[256];
  float res;
  fputs("Привет!",stderr);
  for (;;)
  {
    fputc('>', stderr);
    fgets(s, stderr);
    if (s[0]=='.')
      break;
    switch (eval_expr(parse_expr(s), m, &res))
    {
      case 0 :
        fprintf(stderr,"%d\n", res);
        break;
      case ERR_DIV0 :
        fputs("Деление на ноль",stderr);
        break;
      case ERR_OVERFLOW :
        ...
    }
  }
   fputs(“Пока.”,stderr);
}
```

Недостатком такого метода является то, что пришлось существенно усложнить реализацию: кроме того, что приходится передавать дополнительные параметры, каждый вызов функции приходится обрамлять проверкой того, какой код она вернула, хотя всё, что нам требовалось – передать управление в нужную точку в случае возникновения ошибки. Стандартная библиотека языка C предоставляет для этой цели так называемые *нелокальные переходы*, специфицированные во включаемом файле `setjmp.h`. Она определяет тип `jmp_buf`, содержащий точку, в которую надо передать управление, а также всю необходимую информацию для корректного завершения (в том числе и рекурсивных) функций.

Функция

```
int setjmp(jmp_buf env);
```

запоминает в переданном параметре[35] обстановку вычислений и возвращает 0. Функция

```
void longjmp(jmp_buf env, int val);
```

восстанавливает запомненную `setjmp` обстановку вычислений, возвращается в то место, где `setjmp` собирался вернуть 0, но вместо этого заставляет выдать `val`. Таким образом, в следующей реализации

```
#include<setjmp.h>
jmp_bufenv;
void read_eval_print(Memory m)
{
  char s[256];
  int res;
  fputs("Привет!",stderr);
  for (;;)
  {
    fputc('>', stderr);
    fgets(s, stderr);
    if (s[0]=='.')
      break;
    switch (setjmp(env))
    {
      case 0 :
        fprintf(stderr,"%d\n",
          eval_expr(parse_expr(s), m));
        break;
      case ERR_DIV0 :
        fputs("Деление на ноль",stderr);
        break;
      case ERR_OVERFLOW :
        ...
    }
  }
  fputs("Пока.",stderr);
}
```

[35] На самом деле setjmp является макросом, поскольку в противном случае она не могла бы изменить параметр, переданный по значению.

при вызове setjmp управление будет передано на альтернативу case 0, и если ничего не случится, то будет распечатано вычисленное значение выражения.

Функцию eval_expr вернем практически к ее начальному виду, только вместо перехода goto используем longjmp, у которой первым параметром будет запомненная setjmp в глобальной переменной обстановка вычислений, а вторым – код ошибки:

```
float eval_expr(struct Expr * e, Memory m)
{
  swith (e->code)
  {
    ...
    case EC_BINOP:
    {
      float v1 = eval_expr(e->choice.binop.left, m);
      float v2 = eval_expr(e->choice.binop.right, m);
      switch (e->choice.binop.op)
      {
        case '/':
        {
          if (v2 == 0)
            longjmp(jmp_buf, ERR_DIV0);
          return v1/v2;
        }
      }
      ...
    }
    ...
  }
}
```

Выполнение longjmp в этой функции вернет управление в заголовок переключателя switch в функции read_eval_print, а после этого – к альтернативе case ERR_DIV0.

Таким образом, мы отделили обработку исключительных ситуаций от логики вычисления выражений. Обстановку вычислений следовало бы хранить не в глобальной переменной, а передавать дополнительным параметром, что позволило бы вызывать eval_expr из разных мест.

Как и переходы по вычисляемой метке, нелокальные переходы в языке C являются средством очень низкого уровня. С одной стороны, это позволяет использовать их не только для обработки исключительных ситуаций, но и, скажем, для реализации со-программ, в которых управление может быть многократно передано из одной процедуры в другую и обратно, что моделирует их (псевдо)параллельное выполнение.

С другой стороны, при неаккуратном использовании нелокальные переходы приводят к очень тяжелым ошибкам, типичными примерами которых являются использование `longjmp` прежде, чем была вызвана `setjmp`, либо случай, когда та процедура, в которой была вызвана `setjmp`, уже закончила выполнение. В современных языках есть специальные средства обработки исключительных ситуаций – try-блоки и исключения, которые в большинстве случаев позволяют избежать таких ситуаций. Так, в нашем примере оператор `switch` в процедуре `read_eval_print` является примером типичного шаблона, где первая альтернатива является охраняемым фрагментом, в котором может произойти исключительная ситуация, а остальные альтернативы – реакциями на исключения.

9. РАСПРЕДЕЛЕНИЕ ПАМЯТИ

Все объекты программы можно классифицировать в зависимости от способа их создания и времени существования на глобальные, локальные (или автоматические) и динамические.

Глобальные объекты существуют все время исполнения программы. Обычно они описываются на самом верхнем уровне. Достоинством глобальных объектов является то, что их адрес может быть вычислен статически во время сборки программы, и, следовательно, доступ к ним эффективен. Недостатком, естественно, является то, что даже если такой объект реально нужен только в течение ограниченного периода времени, он будет занимать память, пока программа не закончится.

Автоматическими называются локальные объекты процедур и функций. Как мы уже обсуждали, они появляются только при вызове процедуры, а после ее завершения удаляются, причем для этого не требуется ничего дополнительно указывать. Отсюда и название. Достоинством является, в частности, то, что если процедура никогда не вызывается, то и память под локальные объекты не отводится. Однако, если она вызывается многократно, то проявятся накладные расходы на создание и удаление. Кроме того, одна и та же локальная переменная или формальный параметр может при разных вызовах размещаться по разным адресам, что делает доступ к ним менее эффективным, чем к глобальным переменным.

Поскольку выделение памяти для локальных объектов соответствует *дисциплине FIFO* (first-in-first-out), означающей в данном случае, что первым будет удален тот фрейм, который был создан последним, то для реализации может быть использован стек. Все фреймы будем хранить в одном байтовом массиве `stack`. Переменная `sp` (stack pointer) будет указывать на первый свободный байт:

```
char stack[10000];
char * sp = stack;
```

Тогда процедуры выделения и освобождения памяти в стеке могут быть реализованы следующим образом:

```
void * stack_alloc(int size)
{
    void * f =  (void *) sp;
    sp += size;
    return f;
}
void stack_free(int size)
{
    sp -= size;
}
```

Осталось определить макросы

```
#define frame_new(p) p=stack_alloc(sizeof(*p))
#define frame_dispose(p) stack_free(sizeof(*p))
```

Рассмотренная реализация стека очень упрощенная: она отводит под стек массив фиксированной длины, не рассматривает случай переполнения стека, игнорирует выравнивание и т. п. Возможны и принципиально другие методы организации хранения фреймов. Например, техника *мелкого стека* определяет для каждой функции (или даже для каждого параметра) свой стек, что, в частности, решает проблему с вложенными процедурами.

Динамические объекты появляются по специальному запросу в так называемой *куче*. Время их жизни явно не связано процедурами и функциями, в которых они были созданы. Стандартная библиотека языка С предоставляет следующие функции для создания и удаления динамических объектов, определенные в стандартном файле `alloc.h` (или `malloc.h`):

```
void * malloc(unsigned int size);
void free(void * ptr);
```

Функция `malloc` находит в куче свободное место размера `size` и выдает указатель на него, а процедура `free` отмечает указанное место как свободное. Заметим, что функции `free` не требуется передавать размер освобождаемой памяти, что означает, что куча организована таким образом, что в ней запоминается размер, запрошенный при вызове `malloc`. Ни та, ни другая процедура не

знает тип объекта, для которого запрашивается память, а только его размер.

Выделение динамической памяти весьма сложная процедура, и существуют разные методы ее реализации. Например, можно организовать списки для каждого запрашиваемого размера (или по крайней мере для наиболее распространенных значений). При этом, конечно, придется решать проблему, когда имеется много свободных фрагментов одного размера, а запрашивается другой. Во избежание *фрагментации*, т. е. ситуации, когда свободная память есть, но слишком мелкими фрагментами, можно соединять последовательно расположенные свободные фрагменты в один большего размера и т. п. Так или иначе для организации кучи требуется дополнительная память, и если запросить 1 байт, то вполне возможно, что это займет намного больше.

Для некоторой оптимизации в стандартной библиотеке определены также функции

```
void * calloc(unsigned int count, unsigned int size);
void * realloc(void * ptr, unsigned int size);
```

Функция `calloc` предназначена для динамического размещения массивов, но по сути

```
calloc(count, size)
```

эквивалентно

```
malloc(count * size)
```

с той разницей, что `calloc` обнуляет выдаваемой фрагмент памяти и может сделать это более эффективно за счет использования специальных машинных команд.

Вызов функции

```
realloc(p, size)
```

функционально эквивалентен последовательности

```
(free(p), malloc(size))
```

но `realloc` может сделать (а может и не делать) это более эффективно, например, в случае, если размер `size` меньше, чем текущий размер фрагмента, на который указывает указатель `p`.

Низкий уровень процедур работы с динамической памятью является источником большого количества ошибок. Рассмотрим наиболее типичные из них. При размещении объекта объем выделяемой памяти может быть недостаточен для его представления. Например,

```
int * p = malloc(2);
*p = 123;
```

может вполне хорошо работать, если размер целого в данной системе программирования равен 2, но будет приводить к непредсказуемым последствиям, если он равен 4. Такие ошибки не возникают, скажем, в языке Паскаль, где размещение памяти выполняется псевдопроцедурой

```
new(p)
```

которая «знает» о типе, на который указывает `p`. В языке C это можно реализовать с помощью макроса

```
#define new(p) p = malloc(sizeof(*p))
```

Аналогично попытка копирования строки

```
char * dest = strcpy((char *) malloc(strlen(source)),
source);
```

приведет к ошибкам, поскольку при выделении памяти не учтен 0, завершающей строку `source`.

Каждый размещенный фрагмент памяти должен быть освобожден. Если указатель на выделенный участок памяти потерян, то теряется и возможность освободить его. В простейшем случае при выполнении последовательности

```
new(p);
new(p);
```

первый из выделенных фрагментов станет «мусором». Если в ходе выполнения программы это происходит многократно, то происходит так называемая *утечка памяти*. Как мы увидим позже, определить причины и устранить утечки памяти бывает весьма непросто.

Еще больше ошибок связано с процедурой освобождения памяти free. Повторное освобождение указателя, как в случае

```
new(p);
q = p;
free(p);
free(q);
```

может привести к разрушению структуры кучи, а ошибка проявится лишь в одном из последующих malloc. Аналогичный эффект может иметь и освобождение памяти через указатель, который не был получен путем выделения памяти. Например, это случится при

```
p = calloc(10, sizeof(*p));
p++;
free(p);
```

хотя указатель p и указывает в область кучи.

Обращение к ранее выделенной динамической памяти, которая к данному моменту уже была освобождена, приводит к непредсказуемым последствиям. Так, последнее условие в следующем фрагменте:

```
new(p);
p->a = 5;
free(p);
if (p->a == 5)
   ...
```

может оказаться ложным, поскольку вполне возможно, что free изменило память, на которую указывает p.

Ошибки, связанные с распределением памяти, относятся к самым трудным, поскольку:

- проявляются далеко от места ошибки и внешне могут показаться не связанными с распределением памяти;
- могут возникать только при переносе программы в другую систему программирования;
- попытки обнаружения этих ошибок путем внесения в программу отладочных действий могут скрыть их или перебросить в другое место.

Поскольку большинство указанных ошибок связано с освобождением памяти, то кардинальным решением проблемы является *автоматическая сборка мусора*. В этом случае имеются только операции создания объектов, а удаление происходит автоматически. С логической точки зрения можно считать, что размещенный объект существует до конца исполнения программы. Система поддержки исполнения может удалить объект, если на него больше не существует ссылок. В простейшей реализации, как это сделано, скажем, в языке Visual Basic, с каждым объектом связывается счетчик ссылок, который увеличивается или уменьшается при присваивании указательных переменных. Когда же счетчик становится равным 0, объект можно удалять.

Такая реализация оказывается недостаточной в случае, если появляется множество объектов, которые ссылаются друг на друга, но ни одна переменная программы не ссылается ни на один из них. Тогда это множество объектов может быть удалено только все целиком.

По многим причинам – наличия адресной арифметики, возможности некорректного приведения типов и др. – автоматическая сборка мусора принципиально невозможна в языке С. Впервые она была применена в 1959 году в языке Lisp и сейчас доступна во многих языках программирования, таких как Java, C# и др., особенно там, где надежность ставится выше, чем эффективность. Главным аргументом против автоматической сборки мусора является то, что она происходит в непредсказуемые моменты времени и достаточно сложна. Это не позволяет использовать такие языки для задач реального времени, где требуется гарантированное время отклика. Однако методы автоматической сборки мусора постоянно совершенствуются и область их использования расширяется.

В качестве примера размещения динамической структуры данных рассмотрим реализацию процедуры разбора выражения `parse_expr`. Напомним, что грамматика выражения задана следующим образом:

выр ::= *прост-выр* [(= | < | <= | <>) *прост-выр*]
прост-выр ::= [+ | -] *слаг* ((+ | -) *слаг*)*
слаг ::= *множ* ((* | /) *множ*)*
множ ::= (*перем* | *конст* | (*выр*))

Оставим в качестве упражнения реализацию лексического анализа и будем считать, что задан тип, в котором перечислены все типы лексем:

```
enum TokenCode
{
    TC_EOF,      // конец входной строки
    TC_VALUE,    // число
    TC_VAR,      // имя переменной
    TC_LPAR,     // (
    TC_RPAR,     // )
    TC_PLUS,     // +
    TC_MINUS,    // -
    TC_MULT,     // *
    TC_DIV,      // /
    ...
};
```

и определена структура, представляющая лексему, в которой, если необходимо, помимо ее кода указана дополнительная информация о значении или имени:

```
struct Token {
    enum TokenCode code;
    union {
        float value;
        char name[8];
    } choice;
};
```

Также будем считать, что определен тип `Lexer`, для которого реализована функция, вырабатывающая ссылку на очередную считанную лексему:

```
struct Token * get_token(Lexer lexer);
```

Поскольку нам потребуется «заглядывать» на шаг вперед, то нужна и функция, которая возвращает обратно лексему:

```
void unget_token(Lexer lexer, struct Token * token);
```

Разбор выражения будем осуществлять методом рекурсивного спуска, сопоставив каждому нетерминалу грамматики по отдельной функции, каждая из которых будет возвращать считанное выражение:

```
struct Expr * parse_expr(Lexer lexer);
struct Expr * parse_simple(Lexer lexer);
struct Expr * parse_term(Lexer lexer);
struct Expr * parse_factor(Lexer lexer);
```

Для экономии места приведем реализацию только двух последних функций, оставив первые две в качестве простого упражнения:

```
#include <malloc.h>
#include <string.h>

struct Expr * parse_term(Lexer lexer)
{
 struct Expr* res = parse_factor(lexer);
 for (;;)
 {
  struct Token * t = get_token(lexer);
  switch (t->code)
  {
  case TC_MULT:
  case TC_DIV:
      struct Expr* left = res;
      struct Expr * right = parse_factor(lexer);
      res = (struct Expr*) malloc(sizeof(*res));
      res->code = EC_BINOP;
      res->choice.binop.op =
         (t->code == TC_MULT ? '*' : '/');
      res->choice.binop.left = left;
      res->choice.binop.right = right;
      break;
  default:
      unget_token(lexer, t);
      return res;
  }
 }
}
```

```
struct Expr* parse_factor(Lexer lexer)
{
  struct Expr* res = NULL;
  struct Token* t = get_token(lexer);
  switch (t->code)
  {
  case TC_VALUE:
      res = (struct Expr*) malloc(sizeof(*res));
      res->code = EC_VALUE;
      res->choice.value = t->choice.value;
      break;
  case TC_VAR:
      res = (struct Expr*) malloc(sizeof(*res));
      res->code = EC_VAR;
      strcpy(res->choice.name, t->choice.name);
      break;
  case TC_LPAR:
      res = parse_expr(lexer);
      if (get_token(lexer)->code != TC_RPAR)
         longjmp(jmp_buf, ERR_SYNTAX);
  default:
      longjmp(jmp_buf, ERR_SYNTAX);;
  }
  return res;
}
```

В каждом из отмеченных вызовов функции `malloc` создается новая вершина дерева разбора. В случае бинарной операции в поля `left` и `right` присваиваются ссылки на уже построенные поддеревья. Отметим, что во всех случаях память для вершины выделяется по максимуму, хотя для вершины-числа ее можно было бы выделять меньше, чем для вершины-операции.

В случае синтаксической ошибки нелокальный переход `longjmp` возвращает управление в цикл «читать-вычислять-печатать» с соответствующим кодом ответа.

После вычисления и печати значения полученного выражения его необходимо удалить, чтобы избежать утечки памяти. Для удаления рекурсивной структуры данных, какой является `Expr`, потре-

буется рекурсивная процедура `free_expr`, которая обходит дерево разбора и удаляет вершины, начиная с листьев:

```
void free_expr(struct Expr* e)
{
    switch (e->code) {
    case EC_VALUE:
    case EC_VAR:
        break;
    case EC_UNOP:
        free_expr(e->choice.unop.arg);
        break;
     case EC_BINOP:
        free_expr(e->choice.binop.left);
        free_expr(e->choice.binop.right);
        break;
     }
     free(e);
}
```

Тело цикла «читать-вычислять-печатать» теперь надо переделать следующим образом:

```
    ...
    switch (setjmp(env))
    {
      case 0 :
        struct Expr * e;
        fprintf(stderr,"%d\n",
                e = eval_expr(parse_expr(s), m));
        free_expr(e);
        break;
      case ERR_DIV0 :
        fputs("Деление на ноль",stderr);
        break;
      case ERR_OVERFLOW :
        ...
    }
    ...
```

К сожалению, исправленная таким образом программа не исключает полностью утечек памяти, которые могут возникнуть в случае обнаружения синтаксической ошибки. В этом случае уже созданные объекты, соответствующие успешно разобранным частям выражения, останутся недостижимым мусором. Оставляем в качестве упражнения устранение этой ошибки.

10. ВВОД-ВЫВОД

Практически любая программа вводит входные данные и выводит полученные результаты. Для этой цели используются *файлы*. Если большинство рассмотренных выше конструкций (за исключением, может быть, распределения памяти) оперировали исключительно с объектами программы, то ввод-вывод обращается к объектам операционной системы.

Операционная система работает с *физическими файлами*, представляющими последовательность байтов, скрывая при этом, что именно за ней стоит. Это может быть хранящийся на диске файл, принтер, клавиатура, экран дисплея, сетевой ресурс и т. п.

Идентификация файлов осуществляется на основе понятия *пути*, которое может существенно зависеть от конкретной операционной системы. В некоторых системах путь к файлу может начинаться с указания устройства или места в локальной сети, за которым следует последовательность имен вложенных друг в друга директорий (папок, каталогов) и собственно имя файла. В других системах путь начинается с указания имени пользователя, которому принадлежит файл. Некоторые системы могут поддерживать несколько версий файла, и номер версии требуется указать в пути. Естественно, и способ записи пути различается в разных операционных системах. Кроме того, большинство операционных систем имеют средства ограничения прав доступа к файлам и, опять же, делают это каждая по-своему. В системе MS Windows пути могут выглядеть как

```
\\crimson\Users\user3891\Documents\photo.jpg
C:\НГУ\Программирование\2020\Пересдача.txt
```

С другой стороны, программа оперирует *логическими файлами*, которые являются объектами программы, т. е. переменными специального типа данных. Система программирования должна предоставлять возможность установить связь между логическим файлом и физическим. При этом в разные моменты исполнения с одним и тем же логическим файлом могут быть связаны разные физические файлы, а может быть и не связан никакой файл. После установления связи через переменную-файл можно что-то считать из физического файла или записать в него.

Стандартная библиотека языка С предоставляет следующие функции *низкоуровневого ввода-вывода*, определенные в стандартном включаемом файле `fcntl.h`:

```
// создание файла
int creat(char *filename, int permission);
// открытие файла
int open(char *filename, int access, int permission);
// чтение из файла в буфер
int read(int handle, void *buffer, int nbyte );
// запись из буфера в файл
int write(int handle, void *buffer, int nbyte );
// установка текущей позиции
long lseek(int handle, long offset, int whence);
// закрытие файла – освобождение ресурсов
int close(int handle );
// удаление файла
int unlink(char *filename );
```

Продемонстрируем использование этих функций на примере:

```
#include <fcntl.h>
...
int fd;
char buffer[10];
fd=open("C:\\НГУ\\Программирование\\2020\\Пересдача.txt",
        O_RDONLY | O_TEXT );
lseek(fd,4,SEEK_SET);
read(fd, buffer, 10);
close(fd);
...
```

В качестве типа, представляющего логический файл, используется просто целое число. На самом деле это индекс элемента в таблице, представляющей все файлы программы. Эта таблица формируется операционной системой перед запуском программы и является связующим звеном между логическими и физическими файлами. При открытии (`open`) или создании (`creat`) [36] файла система поддержки

[36] Здесь нет опечатки, хотя «создать» по-английски будет «create». Авторы библиотеки заявляли о желании переименовать эту функцию, которое, к сожалению, невыполнимо по причинам обратной совместимости.

исполнения выбирает свободный элемент в этой таблице, заполняет его, устанавливая связь с физическим файлом, и выдает индекс элемента. Размер этой таблицы определяет ограничение на количество одновременно открытых файлов. Для того чтобы освободить элемент таблицы, необходимо вызвать функцию `close`.

Первые три элемента таблицы файлов формируются автоматически при запуске программы и означают 0 – стандартный ввод, 1 – стандартный вывод, 2 – файл ошибок. Стандартный ввод по умолчанию связывается с клавиатурой, а другие два файла – с выводом на дисплей. Они могут быть перенаправлены. Например, при запуске из командной строки в системе MS Windows

```
MyProg.exe <StudentData.txt > Report.txt
```

в качестве стандартного ввода откроется файл `StudentData.txt`, а стандартного вывода – `Report.txt`.

Для каждого открытого файла известна текущая позиция, изначально устанавливаемая в начало файла. Ее можно явно переместить, используя функцию `lseek`. Функция `read` считывает с текущей позиции файла указанное количество байтов и помещает их по указанному адресу, перемещая при этом текущую позицию. Функция `write` осуществляет запись в файл аналогичным образом.

В принципе, этих средств достаточно для реализации обмена. Однако надо учитывать, что вызов любой из этих функций является обращением к операционной системе, что является очень дорогостоящей операцией. Так, считывание по очереди тысячи байтов потребует во много раз больше времени, чем их считывание за одно обращение к `read`.

Для минимизации количества обращений к операционной системе используется *буферизованный ввод-вывод*, типы и функции которого определены в стандартном включаемом файле `stdio.h`. Идея заключается в том, что с каждым файлом связывается буфер – достаточно большой байтовый массив, размер которого может определяться характеристиками физических устройств. Например, память на жестком диске разбивается на блоки фиксированного размера и блок всегда считывается целиком. Тогда размер буфера файла разумно сделать кратным размеру блока на диске. При использовании буферизованного ввода-вывода данные сначала перемещаются с физического файла в буфер, а лишь затем из буфера по конечному назначению.

Если при последующем чтении требуемые данные уже находятся в буфере, то обращения к физическому файлу не происходит.

Определенная в stdio.h структура FILE содержит номер файла, буфер и дополнительную информацию, необходимую для реализации описанной выше схемы обмена. Для стандартного ввода, вывода и файла ошибок определены переменные stdin, stdout и stderr, соответственно.

Перечень функций буферизованного вывода практически дублирует функции низкоуровнего ввода-вывода:

```
// открытие файла
FILE *fopen(char *filename, char *mode);
                            //mode == "r" - чтение
                            //mode == "w" - запись
                            //mode == "a" - дозапись
// чтение из файла count элементов размера size
long fread(void* ptr, long size, long count, FILE * stream);
// запись в файл count элементов размера size
long fwrite(void* ptr, long size, long count, FILE * stream);
// установка текущей позиции
int fseek(FILE * stream, long offset, int origin);
// установка текущей позиции
long ftell(FILE * stream);
// закрытие файла - освобождение ресурсов
int fclose(FILE * stream);
```

Использования буферизованного и низкоуровневого ввода-вывода также очень похожи:

```
FILE * f;
char bname[8],  bmarks[6];
f = fopen("C:\\НГУ\\Программирование\\2020\\Пересдача.txt","r");
fread(bname,7,1,f);
fread(bmarks,6,1,f);
fclose(f);
```

но здесь второй вызов fread уже не будет обращаться к операционной системе.

Поскольку как низкоуровневый, так и буферизованный ввод-вывод входят в стандартные библиотеки, то не всегда понятно, какой из двух следует использовать. Понятно, что буферизованный вывод, несмотря на описанные выше преимущества, тоже не бесплатный. Во-первых, он требует памяти для буфера, и, во-вторых, все данные будут перемещаться дважды: из файла в буфер, а затем – из буфера по назначению. В случае если требуется считать файл целиком и для этого достаточно памяти, то низкоуровневый ввод-вывод будет эффективнее.

Как низкоуровневый, так и буферизованный ввод-вывод может приводить к ошибкам, подобным ошибкам при работе с указателями:

- чтение из закрытого или неоткрытого файла, как и повторное закрытие файла могут сразу прервать выполнение программы, поскольку в таблице файлов имеется соответствующая информация;
- незакрытие файла, которое может привести к исчерпанию таблицы свободных файлов;
- несоответствие размера запрашиваемых данных и размера буфера – наиболее тяжелая ошибка, приводящая к непредсказуемым последствиям.

Все рассмотренные выше процедуры чтения и записи работают только с байтовыми массивами. Это значит, что вызов

```
int x = 1234;
fwrite(&i, sizeof(int), 1, f);
```

запишет в файл `f` бинарное представление числа `x`, а не текст «`1234`».

Для преобразования данных в текст используется *форматный ввод-вывод*. Мы уже рассматривали функцию `printf`, которая делает это в языке С, а также проблемы, с ней связанные. Например, при вызове

```
fprintf(f, "%6.2f + %6.2f = %7.2f\n", x, y, x+y);
```

транслятор не может проверить, что элемент формата `%7.2` соответствует параметру `x+y`, если в него не заложены специфические знания о функции `fprintf`. И даже если это так, то строка-формат

может не быть константой, а формироваться динамически, например, считываться из файла.

В языке C имеются достаточно развитые средства форматирования, позволяющие печатать десятичные, восьмеричные и шестнадцатеричные числа, вещественные числа в различном виде, вставлять дополнительные пробелы по левому или правому краю поля вывода и т. д. Подробнее об этом можно узнать в стандарте языка C. Однако набор этих средств ограничен и нет возможности его расширить. Например, если возникнет потребность печатать числа в двоичном виде или выражения, заданные указателем на рассмотренную выше структуру `struct Expr`, то встроить эту функциональность в `printf` не удастся.

Простым решением проблемы является использование отдельных функций ввода-вывода для каждого типа данных. Например, в языке Modula-2 те же действия можно выполнить с помощью последовательности операторов:

```
FWriteFloat(f, x, 6, 2);
FWriteString(f, ' + ');
FWriteFloat(f, y, 6, 2);
FWriteString(f, ' = ');
FWriteFloat(f, y, 7, 2);
FWriteLn(f);
```

что замечательно с точки зрения статического контроля, но гораздо менее наглядно и, вероятно, менее эффективно.

Многие языки программирования вводят для форматного ввода-вывода специальные конструкции. В языке Паскаль для вывода используются стандартные псевдопроцедуры `Write` и `WriteLn`:

```
WriteLn(f, x:6:2, ' + ', y:6:2, ' = ', x+y:7:2);
```

Эти псевдопроцедуры, в отличие от обычных процедур, могут иметь произвольное количество параметров, а также в параметрах можно указывать способ форматирования, специфичный для конкретного типа. То есть процедурами они называются только для удобства восприятия, а на самом деле являются специальными синтаксическими конструкциями.

В некоторых языках операторы ввода-вывода имеют весьма развитый синтаксис. Например, в языке Фортран оператор

```
      READ (f,2) (X(I), I=1,100)
2     FORMAT (16F5,1)
```

содержит внутри цикл, который считывает 100 элементов массива X. Отметим, что в данной конструкции действия, связанные с преобразованием числа в текст, отделены от собственно ввода и задаются специальной конструкцией FORMAT. Это дает возможность использовать один и тот же формат в нескольких командах ввода-вывода.

В языке C# отделение форматирования от ввода-вывода привело к понятию *интерполяции строк*, которая позволяет вставить в строковую константу форматированные параметры. Например, интерполированная строка

```
$"{x,6:f2} + {y,6:f2} = {x+y,7:f2}"
```

по существу представляет собой достаточно сложную последовательность вызовов операций форматирования, которую порождает и может полностью проконтролировать транслятор.

11. ОЦЕНКА РЕСУРСОЕМКОСТИ ПРОГРАММ

Человечество издавна использует оценочные категории, такие как «хорошо/плохо» или «лучше/хуже» и т. д., для сравнения объектов, явлений и процессов окружающего мира. Одни из них обладают изрядной долей субъективизма, другие общеприняты и широко разделяемы, в том числе формализованы некоторым способом. Естественным образом такие вопросы возникают и в отношении программ: какая из двух программ, делающих одно и то же, делает это лучше. Какие принципы могут быть положены в основу такого рода сравнений? В данной главе мы рассмотрим сравнение программ в смысле потребляемых ими при исполнении ресурсов, таких как, например, время и память. Чрезвычайно полезным и вдохновляющим на новые программные решения является чтение книг, где излагаются математические основы разработки и анализа алгоритмов [2, 6, 10, 19, 26, 30, 31, 32, 33, 39, 40, 60].

11.1. Элементы теории сложности

Перед тем как перейти к более практическим аспектам оценивания ресурсоемкости программ, отметим следующее. Раздел математики, изучающий формализованные свойства вычислительных процедур, которые нас интересуют в данном контексте, – это теория алгоритмов. Поэтому следует установить связь между алгоритмами и программами. Кроме того, как и теория вычислимости («экзистенциальная» часть теории алгоритмов, т. е. отвечающая на вопрос о принципиальном существовании алгоритма для некоторой задачи и смежные вопросы), так и теория сложности («метрическая» часть теории алгоритмов, т. е. отвечающая на вопрос о количественных характеристиках существующего алгоритма для некоторой задачи или общие алгоритмические свойства этой задачи, которые не связаны с конкретным алгоритмом; далее нами будет рассматриваться только эта часть теории алгоритмов) может развиваться в двух направлениях.

Рассматривая вычислительные процедуры как вычисление функций, можно развивать так называемую *машинно-независимую теорию сложности*. В этом случае понятие сложности (меры сложности) вводится аксиоматически. В рамках этой теории было получено много интересных результатов, демонстрирующих, насколько сложно могут

быть устроены алгоритмы, насколько сложно может быть их поведение. Упомянем лишь несколько из них.

- Существуют сколь угодно сложновычислимых функций.
- Существуют функции, вычисление которых можно как угодно ускорить.
- Не существует (рекурсивного) отношения между величиной функции и сложностью ее вычисления.
- Использование условного оператора позволяет давать более короткие описания (примитивно рекурсивным) функциям.

Конечно же, данная теория мало что может сказать о свойствах алгоритмов, встречающихся в реальном мире и используемых для решения практических задач. Это связано с тем, что при рассмотрении вычислительной процедуры в такой форме игнорируются ограничения, которые, несомненно, имеются при организации вычислений в реальном мире. Поэтому, оставаясь важным теоретическим инструментом исследований, машинно-независимая теория сложности не является приоритетной в практической работе программистов.

Программы, которые пишут программисты, исполняются некоторыми (физическими) вычислительными устройствами. Таким образом, свойства программ, описывающие, как сложно устроен соответствующий вычислительный процесс, зависят от свойств этих вычислительных устройств. В рамках *машинно-зависимой теории сложности* рассматриваются формализованные описания – модели – вычислительных устройств (машины Тьюринга, машины с произвольным доступом к памяти, клеточные автоматы и т. д.), характеристики сложности, применимые к процессам вычислений, которые можно организовать с помощью этих устройств (например, такие как «время» и «память»), а также соотношение, в смысле вычислительных возможностей, между этими устройствами (задачи трансляции одной модели в другую).

11.2. Понятия сложности в «простой» вычислительной модели (машина Тьюринга)

В начале необходимо дать важные пояснения. Во введении к этой главе мы уже упоминали понятия «задача» и «алгоритмические свойства задачи». Пример из обыденной жизни: есть блюдо «жареная картошка», есть десятки рецептов ее приготовления, а есть еще сотни особенностей реализации этих рецептов. Этот

пример неформально соответственно иллюстрирует понятия (алгоритмической) задачи (например, сортировка массива), известных алгоритмов ее решения (многочисленные алгоритмы сортировок, появляющиеся до сих пор) и вариантов их реализации на языках программирования (нетрадиционная реализация алгоритма быстрой сортировки в библиотеке Java 8) для исполнения на конкретных компьютерах [37]. Эти понятия следует четко различать. Например, рассуждая о таком алгоритмическом свойстве некоторой задачи, как минимальная сложность. Это утверждение о том, что не существует алгоритма, решающего задачу с меньшими затратами, а не утверждение о затратах какого-то конкретного алгоритма. Иными словами, утверждение об алгоритмических свойствах некоторой задачи – это утверждение, следующее из свойств всевозможных алгоритмов, решающих эту задачу, в том числе неизвестных.

Необходимые сведения о машинах Тьюринга читатель может почерпнуть в [26, 39, 53]. Время $t_{\mathrm{M}}(w)$ – это длина протокола (количество исполненных команд, которые привели к смене конфигурации МТ) работы машины M на входном слове w. Временная сложность алгоритма в худшем T_{wst}:

$$T_{\mathrm{wst}}(n) = \max_{\mathrm{w \in D(f_M)}}\{t_{\mathrm{M}}(w) : \ |w| \leq n\}.$$

Неформально – это описание «пиковой» сложности алгоритма. Мы можем быть уверены, что вычисление для любого входа потребует не больше шагов, чем описывается сложностью в худшем. Обычно точные значения функций сложности (т. е. для каждого n есть входы, на которых достигается максимум) вывести чрезвычайно трудно, поэтому приводят оценки сложности. Представляют интерес оценки как можно более близкие к точным. Для некоторых задач получены алгоритмы с асимптотически точными оценками (т. е. совпадают с точными в терминах O-большого). Оценка консервативна, если ни на одном входе она не достигается.

[37] Вопрос, можно ли считать всякое описание алгоритма программой на некотором языке программирования, интересный, но несколько уводящий нас в сторону. Поэтому проще разделить эти сущности.

Требуемая «память» – емкостная сложность – машины M на входе w:

$$s_{\mathrm{M}}(w) = \max_{\mathrm{w}}\{p_{\mathrm{i}}\colon (q, S, p)$$

– конфигурация из протокола работы M на слове w}

Неформально говоря, емкостная сложность – это максимальный номер ячейки, в которую заглядывала машина M при работе над словом w. Емкостная сложность алгоритма в худшем S_{wst}:

$$S_{\mathrm{wst}}(n) = \max_{\mathrm{w \in D(f_M)}}\{s_{\mathrm{M}}(w) : |w| \leq n\}.$$

Сложность алгоритма в среднем T_{ave} – математическое ожидание «времени его работы». Если $\mathbb{P}(n, w)$ – вероятность появления слова w среди всех входов длины n, то (временная) сложность алгоритма в среднем:

$$T_{\mathrm{ave}}(n) = \sum_{|\mathrm{w}|=\mathrm{n}} t_{\mathrm{M}}(w) \cdot \mathbb{P}(n, w).$$

Неформально – это (усредненная) оценка сложности алгоритма для наиболее часто встречающихся входных данных. Обычно ввиду недостатка сведений о распределении данных предполагают их равновероятность. В этом случае для временной сложности это обычное среднеарифметическое длин протоколов.

Для сложных вычислительных моделей не всегда возможно точно учесть все исполняемые операции. Рассматривается:

- агрегация элементарных операций ВМ в более крупные команды;
- вычленение наиболее значимых для понимания временных затрат команд.

Таким образом, моделей сложности одного и того же алгоритма может быть несколько.

11.3. Пример: подсчет количества бит

Рассмотрим задачу подсчета количества ненулевых бит в натуральных числах, длина которых не превосходит n бит: $0 \leq N \leq 2^{\mathrm{n}} - 1$. В качестве меры сложности алгоритмов, решающих задачу, выберем количество итераций.

В первом алгоритме биты перебираются по очереди от младшего к старшему ненулевому и ненулевые биты подсчитываются:

```
unsigned bits_count1(unsigned N) {
unsigned count=0;
   while (N) {
      count+=N%2;
      N/=2;
   }
   return count;
}
```

Очевидно, что для любого положительного N количество итераций цикла (для 0 их ноль) определяется позицией максимального ненулевого бита, а значит временная сложность алгоритма (отметим, что данная величина является точной для всех положительных N и ее можно считать достижимой оценкой в худшем):

$$T_{\mathrm{wst}}(N) = \lfloor \log_2 N \rfloor + 1,$$

где $\lfloor \alpha \rfloor$ – максимальное целое число, которое не превосходит α.

Оценка временной сложности алгоритма в среднем – это предыдущая оценка, ранжированная для всех чисел $0 \le N \le 2^{\mathrm{n}} - 1$:

$$T_{\mathrm{ave}}(n) = \frac{1}{2^{\mathrm{n}}}\left(0 + \sum_{\mathrm{N}=1}^{2^{\mathrm{n}}-1} (\lfloor \log_2 N \rfloor + 1)\right).$$

Эту сумму можно легко переписать, избавившись от «неудобной» функции $\lfloor \alpha \rfloor$. Если максимальная единица находится на k-й позиции, то потребуется в точности k итераций цикла, причем таких чисел, с максимальной единицей в k-й позиции, ровно $2^{\mathrm{k}-1}$:

$$T_{\mathrm{ave}}(n) = \frac{1}{2^{n}}\left(0 + \sum_{\mathrm{k}=1}^{\mathrm{n}} k2^{\mathrm{k}-1}\right) = n - 1 + \frac{1}{2^{\mathrm{n}}}.$$

Таким образом, временная сложность этого алгоритма в среднем ненамного лучше, чем сложность в худшем.

Поиски другого алгоритма инспирированы наблюдением, что даже для чисел, содержащих «мало» бит, но которые располагаются «далеко», предыдущий алгоритм будет работать не очень быстро. Собственно, его оценка сложности в среднем подтверждает эти не-

формальные рассуждения. Идея ускорения состоит в том, чтобы научиться каким-либо образом «быстро» отыскивать и исключать в точности ненулевые биты. На очередной итерации таким битом мог бы быть, например, наименьший ненулевой бит. Оказывается, это возможно сделать с помощью одной арифметической и одной побитовой операций. Действительно, двоичные записи чисел N и $N-1$ совпадают «выше» наименьшего ненулевого бита и являются дополнительными друг к другу от этого бита и «ниже».

Во втором алгоритме перебор осуществляется в точности по ненулевым битам, а значит, количество итераций совпадает с их количеством:

```
unsigned bits_count2(unsigned N) {
unsigned count=0;
   while (N) {
      ++count;
      N&=N-1;
   }
   return count;
}
```

Таким образом, временная сложность алгоритма (отметим, что данная величина также является достижимой):

$$T_{\mathrm{wst}}(N) = bits(N),$$

где $bits(N)$ – количество ненулевых бит в двоичном представлении N. Оценим среднюю сложность этого алгоритма:

$$T_{\mathrm{ave}}(n) = \frac{1}{2^{\mathrm{n}}} \sum_{\mathrm{N}=0}^{2^{\mathrm{n}}-1} bits(N).$$

Последнюю сумму, использующую «неочевидную» функцию $bits(N)$, легко выразить через более привычные величины:

$$\sum_{\mathrm{N}=0}^{2^{\mathrm{n}}-1} bits(N) = \sum_{\mathrm{k}=0}^{\mathrm{n}} k \binom{n}{k}.$$

Однако в данном случае эту сумму можно вычислить проще, если заметить, что для каждого числа в интервале от 0 до $2^{\mathrm{n}}-1$ существует однозначно определенное число из того же самого

интервала, двоичная запись которого является дополнением исходного числа. Так как общее количество ненулевых бит в каждой такой паре, которых $\frac{1}{2}2^{\mathrm{n}}$, равно n, то

$$T_{\mathrm{ave}}(n) = \frac{n}{2}.$$

Сравнивая поведение этих двух алгоритмов в среднем, можно отметить, что второй имеет видимое преимущество в скорости работы перед первым.

Первый алгоритм имеет очевидное обобщение. Оно не является слишком интересным с практической точки зрения, однако может служить полезной (и довольно простой) иллюстрацией некоторого важного алгоритмического приема. В исходном алгоритме мы очень ловко заметили, что общее количество ненулевых бит можно подсчитать, «перегоняя» их последовательно в наименьший (нулевой) бит и проверяя, равно ли единице его значение. Возникает идея попробовать использовать несколько наименьших бит. Действительно, их легко выделить и, потом, удалить и единственной проблемой становится быстрый подсчет количества ненулевых бит среди выделенных. Конечно же, этот подсчет должен быть реализован каким-либо эффективным способом, так как иначе желаемое ускорение не будет достигнуто. Если рассматривается небольшое количество бит, например, два наименьших, то подсчет можно организовать следующим образом:

```
unsigned bits_count3(unsigned N) {
   unsigned count=0;
   while (N) {
      switch (N%4) {
         case 3: ++count;
         case 2: ;
         case 1: ++count;
      }
      N /= 4;
   }
   return count;
}
```

Но очевидно, что кроме того, что этот код сомнителен в части понимаемости, он плохо «масштабируем» относительно количества выделяемых бит. Если мы захотим их увеличить (с целью ускорения вычислений) и продолжить реализацию в том же сти-

ле, то получим гигантский оператор выбора. Для большого количества выделяемых бит более целесообразно, неформально говоря, «перенести» управление в данные. А именно, для выбранного количества бит следует предвычислить (любым способом) массив, который позволяет по индексу получить количество бит в этом индексе. В приведенном ниже примере выбрано 8 бит (один байт):

```
unsigned bits_count4(unsigned N) {
char bits[]={
0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,4,5,5,6,5,6,6,7,5,6,6,7,6,7,7,8}
;
unsigned count=0;
   while (N) {
      count+=bits[N%256];
      N/=256;
   }
   return count;
}
```

Отметим, что количество итераций в худшем оценивается как

$$T_{\text{wst}}(N) = \lfloor \log_{256} N \rfloor + 1,$$

а значит, мы в восемь раз ускорили оригинальный алгоритм. Такое же ускорение будет и в среднем.

Полезный урок, который можно извлечь из данной реализации, состоит в том, что программу можно ускорить, если суметь предвычислить некоторые данные, к которым можно эффективно обращаться в процессе «основной» работы. Таким образом, часть вычислений, которые мы заранее знаем, как делать и которые бы пришлось выполнить (возможно многократно) в процессе работы, выносится на более ранние стадии. Она выполняется едино-

жды и результаты сохраняются в хранилище, выборка данных из которого осуществляется быстрее, чем прямые вычисления этих данных. В рассмотренном примере предвычисления были сделаны один раз и навсегда некоторым сторонним образом, при этом они не зависели от возможных входных данных. Более распространенный случай, когда предвычисления интегрированы в саму программу, могут, например, зависеть от конкретных входных данных и сами по себе представлять нетривиальные вычисления.

Важно найти оптимальную границу, до которой предвычисленные данные будут приносить желаемый эффект ускорения и потреблять разумные ресурсы, а за которой от этого будут «убытки». В нашей последней программе подсчета бит можно было бы достичь ускорения и в, например, 16 раз. В этом случае пришлось бы хранить массив размера $2 \cdot 2^{16} Б \approx 130\, кБ$[38]. Не самые страшные размеры, тем более, что этот массив вычисляется единожды. Однако такие затраты памяти для столь частной задачи не выглядят рациональными. Более сложная ситуация в случае, когда сложные предвычисления предваряют «основную» программу. Вспомнив неравенство $T_А \geq S_А$ легко понять, что если, желая получить как можно большее ускорение, мы решим предвычислить очень много данных, то сложность этого предварительного этапа может превзойти сложность основной части. Вообще говоря, не всегда это является криминалом: получается, что вычисления были просто реорганизованы по-другому. Но может случиться и другая ситуация, когда предвычисленных данных много, на них были затрачены значительные временные ресурсы на предварительной стадии, они занимают много памяти, скорость их выборки падает (чем больше хранилище, тем меньше скорость доступа для него), а эффекта ускорения не наблюдается.

---

[38] Конечно, можно заняться «компрессией» имеющихся данных, заметив, что для их хранения достаточно 4 бит, а не 8 (один байт), как это сделано сейчас. Но, во-первых, улучшения по памяти не будут радикальны, а во-вторых, пришлось бы выполнять дополнительные действия, связанные с «декомпрессией», что снижало бы производительность.

11.4. Реализация, пессимизация, оптимизация

Рассмотрим еще одну вычислительную задачу. Прямолинейная реализация вычисления полинома степени *n*, заданного массивом коэффициентов *coef*, в заданной точке x:

```
float power1(int n, float x) {
   return n==0 ? 1.0 : x*power(n-1,x);
}
float poly1(float coef[], int n, float x) {
float sum = 0.0;
   for(int i=0; i<=n; i++)
      sum += coef[i] * power1(i,x);
   return sum;
}
```

Отметим свойства:

- одно умножение на каждой из n+1 итерации цикл **for**;
- глубина рекурсии power равна n;
- i умножений на каждой итерации цикла **for**;
- i фреймов для хранения локальных объектов.

Даже если заменить рекурсию (это типичный пример хвостовой рекурсии) на цикл, производительность возведения в степень оставляет желать лучшего.

Пристальный взгляд на процесс возведения в степень позволяет получить для функции x^{n} несколько более длинное рекурсивное определение, но требующее меньшее количество операций для вычисления определяемой функции. Вместо

$$x^{\mathrm{n}} = \begin{cases} 1, & \text{если } n = 0, \\ x \cdot x^{\mathrm{n}-1}, & \text{иначе,} \end{cases}$$

можно использовать

$$x^{n} = \begin{cases} 1, & \text{если } n = 0, \\ x \cdot x^{\mathrm{n}-1}, & \text{если } n - \text{нечетное}, \\ \left(x^{\frac{\mathrm{n}}{2}}\right)^{2}, & \text{если } n - \text{четное}. \end{cases}$$

Имеем (ВАЖНО: внимательно разберите реализацию тела цикла):

```
float power2(int n, float x) {
float y = 1.0;
   while (n)
     n&1 ? (y*=x,--n) : (x*=x,n/=2);
   return y;
}
```

Свойства:

- одно умножение на каждой из n+1 итерации цикл **for;**
- максимальное количество итераций цикла **while** равно $2\log n$;
- $4\log i$ операций умножения на каждой итерации **for;**
- память – константа.

Заметим, что этот алгоритм можно еще несколько ускорить в среднем, если продолжить его специализацию относительно степени n (т. е. рассмотреть другие возможные случаи для n).

Следует отметить, что довольно часто при реализации функции возведения в целочисленную степень можно наблюдать явление, которое можно было бы назвать *пессимизацией*. Действительно, вспомнив такое математическое свойство $x^n = e^{n\log x}$, студенты пишут что-нибудь вроде (главное обоснование – так короче):

```
float power3(int n, float x) {
   return exp(n*log(x));
}
```

Этот вариант является пессимизированным в том смысле, что в предыдущих вариантах мы имели точно оцениваемое представление о том, какое количество элементарных операций (сложений, умножений и т. д. для целых и вещественных чисел) необходимо для вычисления результата. Сколько требуется элементарных операций для вычисления с требуемой точностью функций exp, log, sin, arctan и так далее, предоставляемых программными библиотеками, предсказать сложно, в частности потому, что реализации библиотек различны. Более существенно следующее. В данном случае игнорируется такая важная информация, как целочисленность n. Это довольно типичная ошибка начинающих программистов – и распространенный пример пессимизации – когда без реальной на то необходимости выполняется переход от целочисленных вычислений

к вещественным. Помимо практически неизбежных потерь в производительности, требуется большая аккуратность при преобразованиях значений из целых в вещественные, а особенно обратно. В заключение отметим, что версии функции pow из стандартной библиотеки C так же игнорируют эту информацию, принимая в качестве обоих параметров вещественные значения, и только в стандартной библиотеке C++ появились версии, использующие целочисленность n и реализующие, по сути дела, вариант, который соответствует `power2`.

Вернемся к первоначальной версии вычисления значения полинома. Легко видеть, что принципиальным недостатком этого варианта является независимое вычисление степени x на каждой итерации цикла, соответствующей вычислению значения очередного монома. Решение – использовать на очередной итерации значение степени, вычисленной на предыдущей. На каждой итерации значение `power` увеличивается в x раз:

```
float poly4(float coef[], int n, float x) {
float sum = 0.0, power = 0.0;
int i;
   for (i=0; i<=n; power*=x,i++)
      sum += coef[i] * power;
   return sum;
}
```

В каком-то смысле обобщением этой идеи является широко известная схема Горнера вычисления полинома:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = ((\ldots(a_n x + a_{n-1})x + \ldots + a_2)x + a_1)x + a_0,$$

здесь скобки определяют порядок вычислений. Ее программная реализация выглядит очень естественно и компактно:

```
float poly5(float coef[], int n, float x) {
float sum = coef[n];
   for (i=n; i>=1; i--)
      sum = sum*x + coef[i];
   return sum;
}
```

и требует наименьшее количество операций среди рассмотренных алгоритмов вычисления значения полинома: n умножений и сложений.

Сравнение различных реализаций вычисления полинома продемонстрировано в данной таблице:

	Время (количество умножений)	Память
Рекурсивная power	$\sum_{i=0}^{n}(1+i)=\frac{n(n+1)}{2}$	O(n)
Итеративная power	$\sum_{i=0}^{n}(1+2\log_2 i) \le (n+1)(1+2\log_2 n)$	константа
$\exp(\log(x)i)$	$\sum_{i=0}^{n}(2+T_{\log}(i)+T_{\exp}(x\log(i)))$	$\max_i(\max(S_{\exp}(x*\log(i)),S_{\log}(i))$
Накопление power	$\sum_{i=0}^{n}2=2n+2$	константа
схема Горнера	$\sum_{i=1}^{n}1=n$	константа

Отметим, что одним из первых нетривиальных результатов алгебраической теории сложности о нижних оценках является положительный ответ, данный в 1966 году В.Я. Пан на гипотезу А.М. Островского, выдвинутую в 1954 году, где было предположено, что схема Горнера является оптимальным по количеству нескалярных умножений способом вычисления значения полинома. Позднее А. Бородин показал, что это в определенном смысле единственная оптимальная процедура для рассматриваемой задачи.

11.5. Организация информационных множеств

Сложность алгоритмов чрезвычайно обширная тема, различным аспектам которой посвящено большое количество исследований, изложенных в литературе [2, 10, 19, 31, 32, 33]. Мы не ставим перед собой задачу осветить эту тему хоть сколь-нибудь широко, но одна область алгоритмики кажется нам очень важной. С одной стороны, она связывает математическую теорию с непосредственной практикой программирования. А с другой стороны, среди практикующих

«реальных» программистов, для которых эта математическая теория скрыта API библиотек, сложилось устойчивое мнение, что это не является предметом, стоящим внимания при разработке программ. Как, впрочем, и при обучении программированию. Между тем данная тема выделена в отдельный курс в международных рекомендациях по обучению компьютерным наукам и программной инженерии (Computing Curricula for Computer Science and Software Engeneering).

11.5.1. Общие понятия

Есть универсум объектов некоторой природы, конечный или бесконечный. Эти информационные[39] объекты/элементы могут появляться и исчезать в программе. Перед разработчиком стоит задача, помимо других, организации их хранения – организация *информационного множества* (также называемого информационным хранилищем; будут использоваться как синонимы), а также манипуляции с ними в рамках этого множества. Как и в других случаях, интерес представляют не произвольные, а эффективные способы решения этой задачи. Здесь под эффективностью подразумевается, что затраты на организацию хранения и скорость манипулирования являются разумными.

Природа объектов универсума, обычно, не имеет значения для организации информационного хранилища, так как предполагается, что для всякого объекта хранения существует его выделенная характеристика-свойство, традиционно называемое *ключом* (key). Это свойство может быть как статической составной частью объекта, так и (легко) динамически вычислимой по объекту величиной. Бывают ситуации, когда ключ полностью совпадает с самим объектом хранения. Примерами ключей для информационных объектов могут служить:

- целые числа (ключ совпадает с самим информационным объектом);

[39] Авторы не считают прилагательное «информационный» удачным в данном контексте, но такова сложившаяся терминология. Конечно же, здесь под информационным объектом подразумеваются данные, представимые в программе и программно обрабатываемые.

- строка символов (ключ совпадает с самим объектом; отметим, что строка может рассматриваться как (длинное) целое число);
- структура, описывающая персональные данные студента некоторого учебного заведения (ключом можно выбрать часть объекта, уникально его характеризующая, например, номер зачетной книжки);
- в предыдущем примере всю область памяти, занимаемую структурой, можно рассматривать как последовательность байт, т. е. как строку символов.

Обязательным требованием является возможность сравнения ключей на равенство. В общем случае не требуется, чтобы на ключах объектов универсума был определен *линейный порядок*, т. е. чтобы для любой пары ключей можно было сказать, какой из них меньше, а какой больше. Имеются примеры информационных множеств как требующие линейного порядка на ключах, так и обходящиеся только сравнением на равенство.

Рассмотрим, какие манипуляции возможны с информационным множеством. Забегая вперед, следует отметить, что выбор допустимых операций очень сильно влияет на способ организации хранилища и поэтому требует тщательной проработки, основывающейся на анализе предметной области.

Очевидно, что всякая работа с информационным множеством как структурой данных начинается с его *инициализации*, результатом которой необязательно является «пустое», т. е. не содержащее элементов, информационное множество. Имеет смысл выделить специальный вид инициализации. Информационные множества, которые будучи единожды проинициализированными набором элементов, далее не изменяющие свое состояние, называются *недеструктивными* [40]. Множества, элементный состав которых может меняться в процессе работы, называются *деструктивными*. Боль-

[40] Можно еще тоньше специфицировать это понятие и различать «физическую»и «логическую» недеструктивность. Физическая подразумевает, что не только набор элементов, но и организация информационного множества не изменяется. Логическая предполагает только неизменность набора элементов множества, а организация его может и меняться, например, для повышения эффективности операций.

шинство примеров, рассматриваемых ниже, являются примерами деструктивных множеств.

Для того чтобы в информационном множестве появились элементы, для него должна быть определена *операция добавления/вставки элемента*. Отметим, что эта операция является деструктивной. Также деструктивной является *операция удаления элемента* из множества.

Примером недеструктивной и, в определенном смысле, самой фундаментальной операцией является *проверка принадлежности элемента* множеству. Эта операция является единственной из базовых применимой к недеструктивным информационным множествам. Она фундаментальна в том смысле, что для информационного множества могут быть определены только две операции – инициализация и проверка на принадлежность.

Таким образом, можно выстроить следующую последовательность операций, постепенно расширяющих функциональность информационного множества:

- инициализация и проверка множества на пустоту;
- проверка на принадлежность;
- добавление элемента;
- удаление элемента.

То, что последняя операция не всегда реализуется даже при необходимости удалять элементы, связано с тем, что она может быть очень дорогостоящей и поэтому проще «бросить» ненужные элементы во множестве, а не удалять их.

Среди других операций, которые можно рассматривать для информационных множеств, укажем:

- подсчет элементов множества;
- отыскание k-го наименьшего/наибольшего элемента множества (над ключами должен быть определен линейный порядок);
- средства перебора элементов множества (итерации над множеством);
- теоретико-множественные операции, такие как объединение, пересечение или симметрическая разность.

Более сложные операции предполагают более сложные методы реализации (и, кстати, требуют более сложной теоретической

базы; примером может служить система управления реляционными базами данных), что неминуемо сказывается на эффективности. А именно эффективность – важнейшее требование к задачам, где используются информационные множества, для которых бедный набор операций компенсируется высочайшей скоростью. Следуя этой практике, мы также не будем рассматривать подобные операции.

Отметим, что дополнительными соображениями, принимаемыми во внимание при реализации информационных множеств, являются:

- дополнительные расходы памяти для поддержания структуры ИМ;
- возможность выделения больших объемов памяти (непрерывный кусок);
- скорость выделения динамической памяти.

Переходя к конкретным примерам информационных множеств, рассмотрим еще один вид анализа сложности алгоритмов, который возникает как раз в связи с исследованием свойств этих множеств. Традиционно понятия «оценка в худшем» и «оценка в среднем» для операций принадлежности/вставки/удаления связаны с отдельно взятой операцией. Однако обычно в процессе исполнения программы используется не одна операция, а серия операций, которые меняют множество. Хотя каждую операцию можно оценить отдельно, имеет смысл оценить «расходы» на серию из них, так как отдельно взятая операция может быть очень затратной, но сложность операций в серии может быть различной, в том числе и существенно меньшей, чем в худшем случае. Например, операция вставки «оптимизирует» организацию множества так, что затраты на последующие операции (возможно не все) уменьшаются. Этот вид анализа называется *амортизационным*. Если такие оценки имеют смысл для рассматриваемых информационных множеств, они будут указываться.

11.5.2. Битовые шкалы

Рассмотрим условия, при которых применим такой широко известный пример организации информационных множеств как *битовые шкалы*. Отметим, что с математической точки зрения,

битовые шкалы – это характеристические векторы конечных множеств. Битовые шкалы применимы, если:

- информационные объекты суть целые числа,
- все они лежат в интервале значений от L до U, причем величина $U–L+1$ (длина битовой шкалы) является «разумной» в рамках разрабатываемого приложения (отметим, что бывают изощренные реализации битовых шкал, учитывающих возможную разреженность представляемого множества, см. ниже).

Разумность длины битовой шкалы определяет разработчик, приняв во внимание, что для реализации потребуется массив длиной $(U–L+1)/8$ байт.

Возможна следующая реализация:

```
#define UINT_BITS 32
typedef struct {
    int       lb;
    unsigned  sz;
    unsigned *bs;
} bitscale;

void init(bitscale *b, int L, int U) {
  //пропущен контроль размера множества
  b->lb=L;
  b->sz=U-L+1;
  b->bs=(unsigned*)calloc(b->sz/UINT_BITS+(b->sz%UINT_BITS?1:0),
                         sizeof(unsigned));
}
int member(int n, bitscale *b) {
  //пропущен контроль диапазона
  return (1<<(n-b->lb)%UINT_BITS) & b->bs[(n-b->lb)/UINT_BITS];
}
void insert(int n, bitscale *b) {
  //пропущен контроль диапазона
  b->bs[(n-b->lb)/UINT_BITS] |= 1<<(n-b->lb)%UINT_BITS;
}
void delete(int n, bitscale *b) {
  //пропущен контроль диапазона
  b->bs[(n-b->lb)/UINT_BITS] &= ~(1<<(n-b->lb)%UINT_BITS);
}
```

Легко видеть, что операции вставки/удаления/принадлежности требуют константное (независящее от размера битовой шкалы) число операций – $O(1)$.

Недостатками этого вида информационных множеств являются ограничение на тип хранимых объектов и потенциальная возможность неэффективного расходования памяти в случае, если объекты-числа, с которыми манипулирует программа, распределены по битовой шкале неравномерно и имеются длинные промежутки, в которых объекты никогда не появляются. Если для очень длинных шкал кластеризация такого рода встречается систематически, то, возможно, имеет смысл рассмотреть иерархическую организацию множества: массив указателей на одинаковые куски исходной шкалы, которые порождаются динамически только тогда, когда появляется первый элемент из данного куска. При данной реализации удаление и проверка принадлежности по-прежнему требуют константного числа операций. Сложность операции вставки зависит от того, является ли она первой для данного куска-промежутка шкалы или нет.

11.5.3. Массивы

Неэффективные расходы памяти в битовых шкалах могут быть исключены в информационном множестве, организованном в виде массива, что, кстати, позволяет ликвидировать и второй недостаток – появляется возможность хранить не только целые числа, но и объекты произвольной структуры. Конечно же, эта экономия не дается бесплатно. Рассмотрим возможные решения и их следствия. Предположим, что при исполнении программы ожидается N объектов хранения (для простоты – различных) и для них выделен «традиционный» массив соответствующей длины. Отметим, что ограничение на длину массива можно интерпретировать по-разному:

А – это общее количество объектов, появившихся в программе, или

B – это максимальное количество объектов, которые могут существовать в программе в отдельный момент времени.

Это вносит определенную специфику в организацию работы с множеством.

Первое простое решение состоит в том, что объекты помещаются в массив последовательно один за другим по мере поступления. Очевидно, сложность такой вставки константна. Так как мы не располагаем никакой информацией о взаимном расположении элементов в информационном множестве-массиве, искать в нем можно, лишь поочередно проверяя элементы и сравнивая их с рассматриваемым ключом. Если на данный момент массив содержит n (может быть равно N) элементов, то при наличии искомого элемента в множестве потребуется не более n шагов, а при его отсутствии там не более n+1. Среднюю сложность поиска тоже посчитать несложно, если предположить, что элемент может находиться в любой k-й ячейке массива с равной вероятностью (всего исходов, вместе с неудачным, n+1):

$$\mathrm{T_{ave}}(n) = \sum_{\mathrm{k=1}}^{\mathrm{n+1}} \frac{1}{n+1} k = \frac{1}{n+1} \sum_{\mathrm{k=1}}^{\mathrm{n+1}} k = \frac{1}{n+1} \frac{(n+2)(n+1)}{2} = \frac{n}{2} + 1.$$

Операцию удаления можно реализовывать разными способами. Отметим, что в любом случае удаляемый элемент необходимо сначала найти, а значит, общая сложность операции удаления будет включать сложность операции поиска. Если имеет место Предположение **А**, то с помощью введения некоторого выделенного фиктивного элемента, помещаемого на место удаляемого, операция удаления может быть организованна за константное время (плюс время поиска), т. е. сложность удаления отличается от сложности поиска на константу. Однако если имеет место Предположение **В**, то удаляемый элемент должен быть фактически удален из множества. Все ячейки массива, следующие за ячейкой с удаляемым элементом, должны быть скопированы в предыдущие ячейки массива. Таким образом, операция удаления при наличии элемента в массиве потребует n, а при его отсутствии – n+1. Итак, для данной реализации имеется высокая скорость вставки и сравнительно низкие скорости поиска и удаления элемента.

Не во всех случаях такая производительность является удовлетворительной. Если для разрабатываемой программы скорость проверки наличия элемента в множестве важнее скорости вставки, то имеет смысл озаботиться организацией множества, более подходящей для такого способа его использования. Например, при

исполнении программы первая операция встречается намного чаще, чем вторая (элементы редко добавляются в множество). Тогда, если такая дилемма существует, эффективность реализации проверки элемента имеет приоритет над эффективностью вставки, так как (амортизационная) сложность достаточно длинной серии операций будет определяться именно сложностью проверки принадлежности. Следующее невеликое усложнение алгоритма вставки позволяет значительно ускорить проверку принадлежности. В предыдущем случае вставка происходила в первую свободную ячейку, а значит, порядок элементов в массиве мог оказаться каким угодно. Новая идея, служащая достижению поставленной цели, состоит в поддержании массива в упорядоченном состоянии (например, по возрастанию ключей; отметим, что это уже требует наличия линейного порядка). Таким образом, следует, во-первых, отыскать подходящее место для вставки (все предыдущие элементы меньше заданного), а затем, если оно не является свободным, освободить его, скопировав хвост массива (в нем все элементы больше заданного) в последующие ячейки. Операция вставки включает поиск заданного элемента, поэтому для начала обсудим сложность поиска.

Хотя можно по-прежнему искать элемент с помощью последовательного просмотра всех элементов с самого начала, однако лучше поступить по-другому. Легко видеть, что задача поиска в упорядоченном массиве эквивалентна задаче отыскания методом дихотомии корня уравнения $F(x)=0$ для некоторой функции, заданной таблично, на интервале длины n, с точностью $\varepsilon=1$. Таким образом, сложность алгоритма поиска составляет $O(\log n)$. Особенностью процесса дихотомии для табличной функции по сравнению с непрерывной функцией является то, что элемент может отсутствовать в массиве. рис. 11.1 демонстрирует поиск числа 3 в заданном массиве, рис. 11.2 демонстрирует поиск числа 9, которое отсутствует в массиве.

Так как после отыскания места вставки следует сдвинуть хвост массива для освобождения ячейки для нового элемента, то общая сложность алгоритма в худшем составляет $O(n)+O(\log n)=O(n)$. Формально сложность вставки в худшем увеличилась. Однако фактически во временном выражении скорость вставки может улучшиться за счет сокращения количества сравнений, сложность которых мы не учитываем, но которые также требуют некоторого времени. Слож-

ность удаления элемента в худшем для Предположения **A** равна $O(\log n)$, для Предположения **B** – $O(n)+O(\log n)=O(n)$.

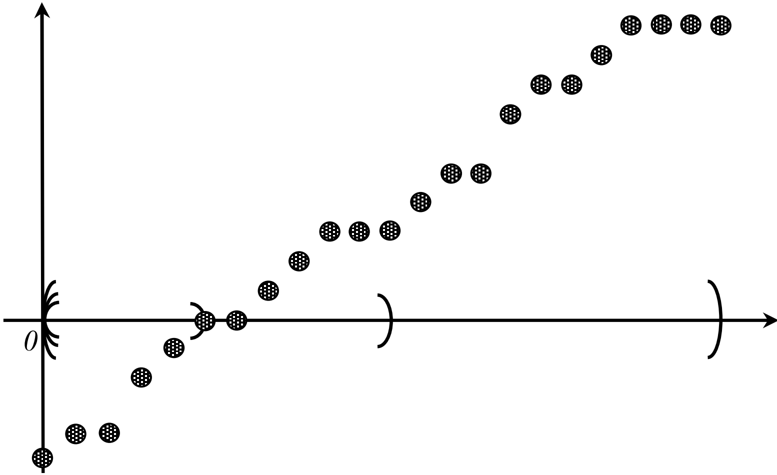


Рис. 11.1. Схематическое изображение
функции A[i]-3, где функция A[i] задана таблично:
A:[0..22]→{-2,-1,-1,1,2,3,3,4,5,6,6,6,7,8,8,10,11,11,12,13,13,13,13}

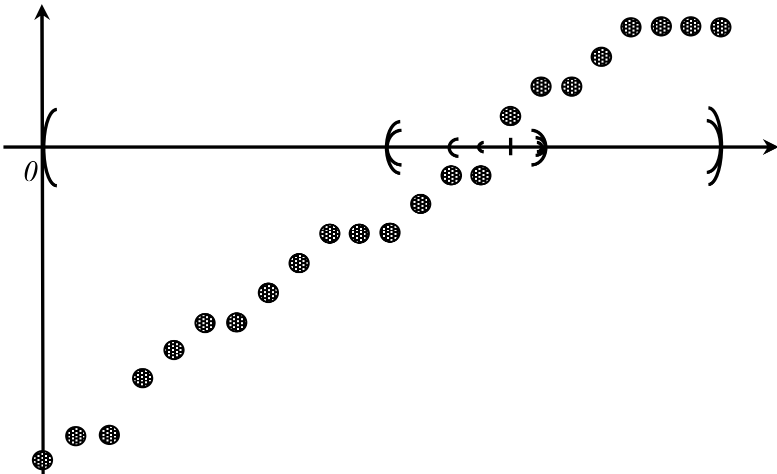


Рис. 11.2. Схематическое изображение функции A[i]-9,
где функция A[i] задана, как и на предыдущем рисунке

До сих пор при рассмотрении информационного множества в виде массива предполагалось, что операции вставки/проверки/удаления элементов следуют в произвольном порядке. Однако бывают специальные случаи использования информационных множеств. Например, сначала в множество элементы только добавляют, потом идут только операции проверки на принадлежность элементов множеству (и это самая массовая операция с множеством), а затем множество очищается. Можно считать, что используется недеструктивное информационное множество – его инициализируют некоторым специальным способом, а затем оно остается неизменным вплоть до своей «смерти». Как мы знаем, поиск в упорядоченном массиве имеет сложность $O(\log n)$. Таким образом, если инициализация массива будет состоять в придании его элементам определенного порядка, то далее самую массовую операцию проверки на принадлежность можно будет выполнять довольно эффективно. Задаче сортировки массивов посвящена обширная литература. Укажем известный факт – существуют алгоритмы сортировки, основанные на сравнениях и достигающие нижней оценки сложности для данной задачи $O(n \log n)$.

Как мы уже упоминали, задачей разработчика является комплексная оценка достоинств и недостатков реализации информационных множеств, используемой в программе. Гарантирующая эффективность поиска возможность прямой адресации в упорядоченных массивах обеспечивается представлением массива в виде непрерывного куска памяти, что может быть обременительно для больших множеств. Если мы можем пожертвовать эффективностью поиска, то проблема решается заменой неупорядоченных массивов на списки. Они могут быть упорядоченными или нет, но и упорядоченность списка не влияет радикально на сложность поиска. Отметим, что в этом случае возникают накладные расходы (как по памяти, так и по времени), связанные с использованием динамической памяти.

11.5.4. Деревья поиска

Итак, упорядоченные массивы (предположим, в возрастающем порядке), вкупе с прямой адресацией, обеспечивают хорошие показатели по времени поиска элементов за счет «неявного»

знания о том, что для любого элемента массива все элементы с меньшими индексами меньше рассматриваемого элемента, а с большими – больше. Знание «неявное» в том смысле, что оно не выражено явно средствами описания структуры данных и использует знание о «традиционной» реализации массивов. Возникает идея – выразить знание о «меньше-больше» для элементов информационного множества в явном виде при описании реализующей его структуры данных (конечно же, с дополнительными расходами памяти для этого). При этом хочется воспроизвести другую важную идею упорядоченных массивов – для всякого выбранного на очередном шаге поиска элемента, элементы, в которых достаточно продолжить поиск (отметим, что это необязательно все оставшиеся непросмотренные элементы), разбиваются на два примерно равных по мощности подмножества, состоящих из элементов, которые меньше и больше, соответственно, выбранного элемента.

Это приводит нас к широко известной структуре данных, носящей название *дерево поиска*, связанного ссылками множества узлов, содержащих интересующую нас информацию. Ниже дается типичное описание узла/вершины дерева (обратите внимание на рекурсивность в описании структуры):

```
Struct TreeNode {
   int key;
   //другие поля, относящиеся к описанию элемента ИМ
   struct TreeNode* left;
   struct TreeNode* right;
};
```

Для упорядоченных деревьев, т. е. для деревьев, где на ключах узлов определен линейный порядок, традиционно предполагается, что для всех узлов узлы с меньшими значениями находятся в поддереве, на которое указывает указатель `left`, а с большими – `right`. Как мы видим, явное описание меньших и больших элементов требует дополнительной памяти размером в два указателя, которая добавляется к каждому элементу ИМ.

Отметим другие признаки таких деревьев. Они бинарные в том смысле, что максимальное число наследников всякой вершины

не больше двух (это явно прописано посредством указателей `left` и `right`). Они ориентированные, так как невозможно перейти от наследника к предшественнику. Наконец, это корневые деревья, так как есть выделенный указатель на узел без предшественников. Это подсказывает нам, что представлением пустого дерева является нулевой указатель. Такие деревья еще называют деревьями *двоичного/бинарного поиска*.

Узлы, для которых `left` и `right` пусты (их значения равны `NULL`), называются *листьями дерева* (сравните с терминальными вершинами в деревьях вывода). Длина пути от корня к некоторому узлу дерева T называется *высотой этого узла*. Узлы с одинаковой высотой образуют уровень, или слой дерева. Максимум среди высот всех листьев дерева называется *высотой дерева* $h(T)$. Таким образом, сложность поиска в дереве (и, в общем, манипуляций с деревом) в худшем случае определяется его высотой, если на каждый узел вдоль этого максимального пути выполняется константное количество действий [41]. Отметим, что сложность поиска в среднем определяется средней высотой узлов дерева (средней длиной путей до вершин).

Условие равномерного распределения вершин по поддеревьям дает логарифмическую скорость поиска во всем дереве, так как на каждом шаге отсекается примерно половина из оставшихся к текущему шагу элементов, где заведомо поиск продолжать не нужно. Это соответствует тому, что высота – количество шагов до листа с максимальной высотой, а значит и до любого другого узла дерева, сравнимая с логарифмом от числа узлов во всем дереве. Легко видеть, что идеальными в этом смысле деревьями (идеально сбалансированными) являются деревья, в которых все слои, за исключением, быть может, последнего, максимально заполнены: количество элементов на k слое, $0 \leq k < h(T)$, равно 2^k (корень находится на нулевом слое).

Довольно просто написать какую-нибудь процедуру добавления элементов в информационное множество, представленное в виде

[41] Далее в этой главе рассмотрены деревья с вычислением штрафа, в которых сложность некоторых операций может зависеть от количества вершин в дереве как $O(n)$, хотя высота его может составлять $O(\log n)$.

бинарного дерева. Ниже приведен рекурсивный вариант (для простоты, пусть элементы ИМ это и есть целые числа):

```
struct TreeNode* Insert(struct TreeNode* t, int k)
{
 if (t == NULL) {
    t = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    t->key = k;
    t->left = t->right = NULL;
 }
 else if (t->key < k)
   t->right =Insert(t->right, k);
 else if (t->key > k)
   t->left =Insert(t->left, k);
 return t;
};

struct TreeNode* tree=NULL;
...
tree = Insert(tree, 10);
...
```

(напишите сами процедуру удаления элемента из дерева).

Однако легко видеть, что такой простой подход не дает желаемых результатов по равномерному распределению вершин по поддеревьям: возрастающая/убывающая (и много других вариантов) последовательность целых чисел, при вставке в заданном порядке, даст дерево, мало отличающееся от упорядоченного списка.

Отметим следующий факт. Если рассмотреть все деревья бинарного поиска на n вершинах, порожденных некоторой «регулярной» процедурой, то наиболее часто встречаемой (почти всегда) величиной высоты деревьев будет величина порядка $4.311 \log n$. То есть почти всегда случайное дерево на n вершинах всего лишь примерно в 4.311 раза хуже, чем идеальное. Как мы увидим дальше, результат не так уж и плох. Однако зачастую мы не можем полагаться на случай и должны гарантировать хорошую высоту дерева при наших манипуляциях – поддерживать деревья в состоянии, близком к сбалансированному, при вставках элементов и при их удалении. Было предложено большое количество способов поддержания сбалансированности деревьев. Мы рассмотрим лишь некоторые из них.

АВЛ-деревья

Исторически это был первый метод балансировки, предложенный Адельсон-Вельским и Ландисом в 1962 году. Этот метод балансировки принадлежит классу методов, использующих информацию о сбалансированности, привязанную локально к каждой вершине дерева, а именно, разность высот левого и правого поддерева вершины (см. рис. 11.3). Оказывается, если потребовать, чтобы для каждой вершины в дереве эта разность не превосходила некоторого фиксированного параметра, то высота дерева будет лишь в константное число раз хуже идеального дерева (чем больше параметр, тем больше константа).

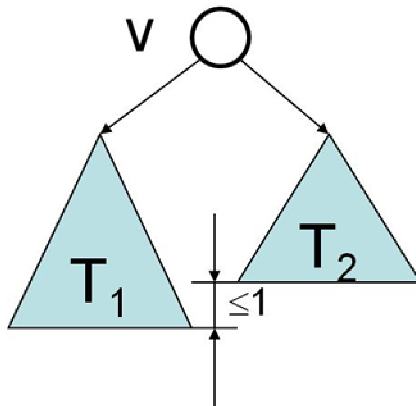


Рис. 11.3. Свойства вершин АВЛ-дерева

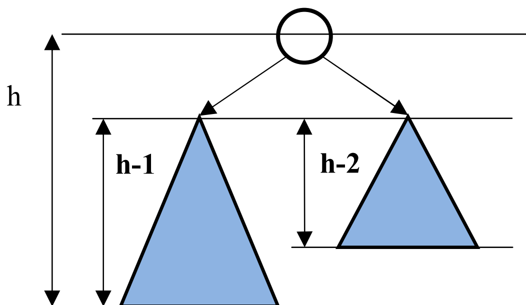


Рис. 11.4. Наиболее асимметричное АВЛ-дерево

Минимальный параметр – это единица. Конструкция самого «плохого» АВЛ-дерева для этого случая приведена на рис. 11.4. Таким образом, количество вершин N в плохом АВЛ-дереве высоты h описывается следующим рекуррентным уравнением:

$$N(h) = N(h-1) + N(h-2) + 1, \qquad N(0) = 0, \qquad N(1) = 1$$

(начальные условия $N(0)$ и $N(1)$ – это количество вершин в тривиальных деревьях, пустом и одновершинном). Решение данного уравнения:

$$N(h) = \frac{1}{\sqrt{5}} \frac{\sqrt{5}+1}{\sqrt{5}-1} \left(\frac{\sqrt{5}+1}{2}\right)^{\mathrm{h}} - 1 + o(1)$$

Отбросив ограниченные константой слагаемые, мы можем выразить высоту дерева через количество вершин в нем следующим образом:

$$h < \log_{\frac{\sqrt{5}+1}{2}} N \approx 1.44 \; \log_2 N.$$

Таким образом, высота худшего АВЛ-дерева всего лишь на 44 % хуже идеального дерева. Средняя длина путей (сложность поиска в среднем) в произвольных АВЛ-деревьях неизвестна, однако в худших она составляет $1.04 \log_2 n$.

Итак, поддерживая при вставке и удалении требуемую структуру АВЛ-дерева, мы можем гарантировать сложность поиска $O(\log n)$. Однако если поддержание этой структуры является само по себе сложной задачей, то это сведет на нет все достоинства логарифмического поиска. К счастью, эти манипуляции так же можно реализовать за логарифмическое время в худшем. Для этого нам понадобится снабдить каждую вершину дерева информацией о ее текущем балансе (разности высот левого и правого дерева). Случай минимального параметра единицы простейший – значениями баланса могут быть –1, 0, 1 (левое выше, высоты одинаковы, правое выше, соответственно), т. е. достаточно 2-х бит на вершину. И операции, требующиеся для поддержания правильной структуры, тоже очень простые.

Рассмотрим вставку нового элемента в дерево. После появления нового листа, мы движемся вверх по предшественникам, меняя подходящим образом балансы (так как для всех вершин на пути к корню одно из поддеревьев выросло), пока не встретится вершина с таким балансом, что его изменение приводило бы к нарушению

условия – разность не более единицы. Возможны две ситуации, которые по способу решения называются одинарным и двойным вращением (рис. 11.5, 11.6). Они отличаются тем, что во втором случае восстановлению баланса мешают две вершины наследника, тогда как в первом – только одна, поэтому второй случай чуть более сложный, чем первый. Но идея у них общая – надо перестроить константную окрестность вершины с плохим балансом, чтобы получился правильный. Легко видеть, что на это потребуется константное число операций. Попробуйте сами, закрыв правую часть картинки, найти эти решения.

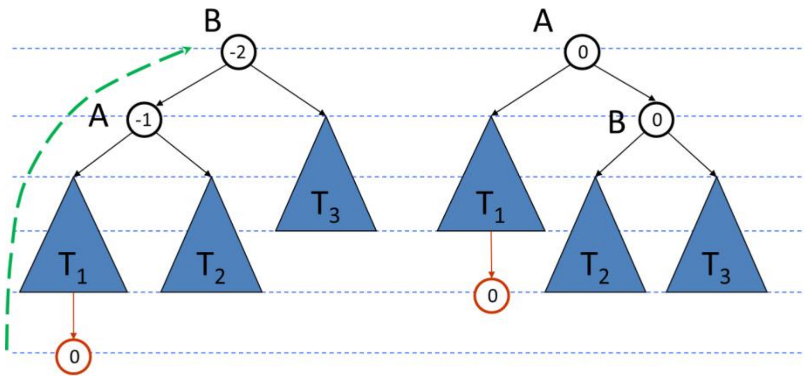


Рис. 11.5. Одинарное вращение

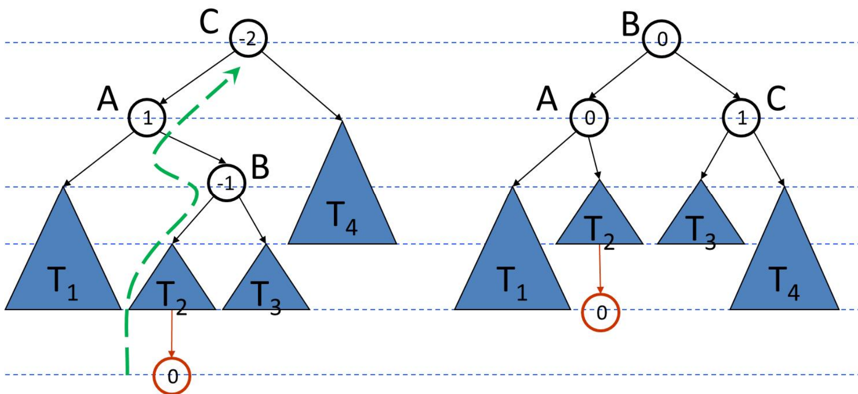


Рис. 11.6. Двойное вращение

Таким образом, двигаясь назад вдоль пути длины $O(\log n)$ (перед вставкой дерево было правильным) и выполняя пусть и в каждой вершине этого пути $O(1)$ число действий по восстановлению баланса, в итоге мы будем иметь общую логарифмическую сложность вставки элемента дерева. Более того, можно показать, что только не более чем для половины вершин в пути от нового листа до корня могут потребоваться операции балансировки. Удаление элемента выполняется аналогично и имеет ту же сложность.

В заключение укажем некоторые данные об эффективности АВЛ-деревьев, которые были получены в результате обширного тестирования. В среднем 1 вращение требуется на 2 вставки и 5 удалений. Общая рекомендация по сравнению с аналогичными деревьями: АВЛ-деревья показывают лучшую производительность, если операция поиска встречается в 5 раз чаще, чем операции вставки и удаления.

WB[α]-деревья, или сбалансированные по весу деревья

Если в предыдущем случае сбалансированность дерева достигалась за счет ограничения на высоты поддеревьев, в сбалансированных по весу деревьях, предложенных Нивергельтом и Рейнгольдом 1973 году, напрямую выражена изначальная идея – хорошее по высоте дерево должно иметь для каждой своей вершины примерно одинаковое количество вершин в левом и правом поддереве. Для всякой вершины v в α-сбалансированном дереве верно:

$$\alpha \le balance(v) \le 1 - \alpha, \qquad \alpha \in \left(0..\frac{1}{2}\right],$$

где

$$balance(v) = \frac{N(T_\mathrm{L})+1}{N(T_\mathrm{L}) + N(T_\mathrm{R})+1},$$

а $N(T_L)$ и $N(T_R)$ – количество вершин в левом и правом поддереве вершины v соответственно (рис. 11.7). Отметим, что деревья для всех значений $\alpha \in [\frac{1}{3}..\frac{1}{2}]$ – это идеально сбалансированные деревья.

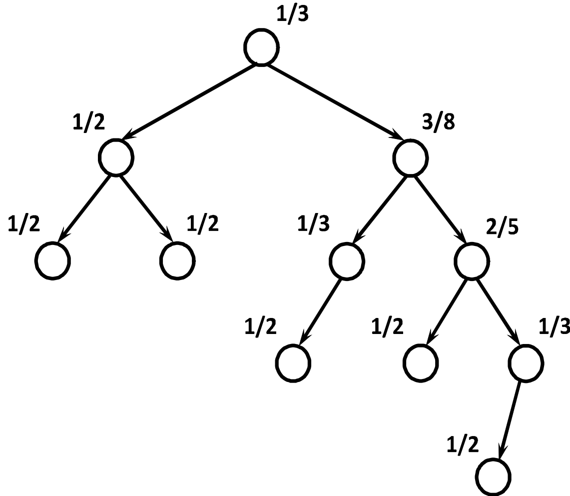


Рис. 11.7. Пример сбалансированного дерева с параметром 1/3

Как видно из определения, для поддержания структуры WB[α]-дерева в каждой вершине нужно хранить информацию о количестве вершин в дереве с корнем в этой вершине (величины $N(T_{\mathrm{L}})$ и $N(T_{\mathrm{R}})$ вычисляются в процессе возвращения из соответствующего поддерева, где новая вершина была вставлена). В целом процедура вставки элемента в дерево выглядит аналогично процедуре вставки в АВЛ-дерево: вставляем, поднимаемся к корню, проверяя баланс, если баланс нарушен, корректируем его, чтобы удовлетворить условию α-сбалансированности. Более того, для $\alpha \in [\frac{2}{11}..1 - \frac{1}{\sqrt{2}}]$, $1 - \frac{1}{\sqrt{2}} \approx 0.29289$, операции балансировки совпадают с таковыми для АВЛ-деревьев. Для других значений α операции балансировки могут быть сконструированы исходя из условия, что в итоге должна получиться «окрестность» вершины, удовлетворяющая условию α-сбалансированности.

Наиболее асимметричное WB[α]-дерево – для всех вершин дерева доля вершин, например, в левых поддеревьях достигает величины α, а в правых, соответственно, 1– α. Высота такого дерева:

$$h_{\alpha} < \log_{\frac{1}{1-\alpha}} n = \frac{1}{\log_2 \frac{1}{1-\alpha}} \log_2 n = \mathrm{C}_{\alpha} \log_2 n.$$

Описание поведения WB[α]-деревьев в среднем требует привлечения функции энтропии $\mathcal{H}(\alpha) = \alpha \log_2 \frac{1}{\alpha} + (1-\alpha) \log_2 \frac{1}{1-\alpha}$. Средняя высота вершины равна

$$h_\alpha < \frac{1}{\mathcal{H}(\alpha)} \log_2 n = \mathrm{C}_\alpha \log_2 n.$$

Ниже приведены параметры значений деревьев для различных значений α:

α	0.29289	0.25	0.2	0.1	0.01
C_α в худшем	2	2.41	3.11	6.58	68.97
C_α в среднем	1.15	1.23	1.385	2.13	12.38

Бóльшая асимметричность (т. е. уменьшение параметра α) ведет к росту высоты дерева, однако существенно по сравнению с АВЛ-деревьями снижает необходимость в выполнении операций балансировки при вставке и удалении элементов. Поэтому если вставка и удаление выполняются так же часто, как и поиск элементов, то WB[α]-деревья начинают выигрывать по производительности у АВЛ-деревьев. Эти деревья используются для реализации множеств в таких языках, как Scheme и Haskell.

Другие примеры сбалансированных деревьев

Два предыдущих примера демонстрируют, что поддержание сбалансированности требует хранения дополнительной информации в каждом узле дерева – два бита в случае АВЛ-деревьев и целое число, способное хранить значения, которые соответствуют максимальному размеру ИМ, в случае WB[α]-деревьев. Возникает вопрос: а можно ли меньше или вообще обойтись без этого?

Можно предположить, что если требовать $O(\log n)$ – сложности операций поиска/вставки/удаления в худшем, то минимальная дополнительная информация для поддержания сбалансированности может быть представлена одним битом. Такой вариант, так называемые *красно-черные деревья*, был предложен Гуибас и

Сэджвик в 1978 году. Именно для представления цвета вершины и требуется один бит. Определение следующее:

1. Вершина может быть либо красной, либо черной.
2. Корень черный.
3. Пустые поддеревья считаются фиктивными черными вершинами без данных (например, у всякого действительного листа будет два фиктивных черных потомка).
4. Оба потомка каждого красного узла черные.
5. Все пути из любой вершины до фиктивных черных вершин содержат одинаковое количество черных вершин.

Пример красно-черного дерева приведен на рис. 11.8.

Условия 4–5 гарантируют, что высота красно-черного дерева не превосходит $2\log(n+1)$ (эти условия заставляют дерево сильно «ветвиться», укорачивая пути от корня к листьям). Поддержание структуры ЧБ-дерева требует шести типов операций перестройки дерева и здесь они будут опущены.

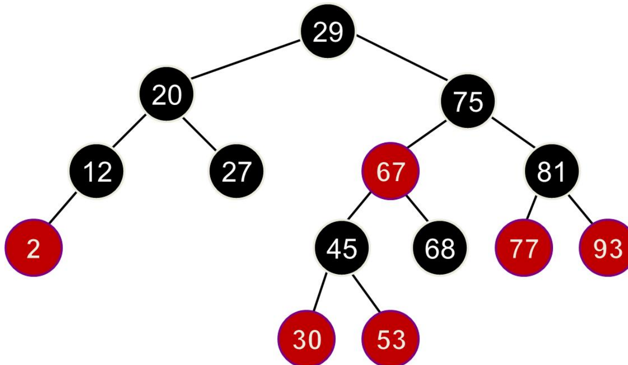


Рис. 11.8. Результат вставки в КЧ-дерево последовательности чисел 12, 81, 20, 75, 27, 67, 29, 45, 68, 77, 93, 30, 2, 53

По сравнению с АВЛ-деревья ЧБ-деревья сбалансированы несколько хуже. Однако имеют преимущества по количеству операций балансировки. Поэтому выбор между ними делается на основе таких же соображений, как и в случае АВЛ- vs WB[α]-деревья. Что касается выбора между WB[α]- и КЧ-деревьями, то, за исключением очевидных различий в объеме дополнительной информации о сбалансированности, они могут вести себя по-разному в приложениях. В этом случае

выбор может определяться дополнительными соображениями, например, дополнительными операциями над ИМ, которые требуются в приложении. КЧ-деревья используются в таких реализациях, как map и set в C++, TreeMap и TreeSet в Java, в планировщике ядра ОС Linux.

Судя по всему, полностью избавиться от дополнительной нформации, распределенной по всему дереву и используемой для поддержания сбалансированности, невозможно. Однако можно попробовать это сделать в случае, когда логарифмическая сложность не требуется при каждом обращении к ИМ с операциями поиска/вставки/удаления. Достаточно, чтобы это было верно в амортизационном смысле: в серии операций одна или несколько могут быть плохими, зато все остальные настолько быстрые (за счет оптимизации внутренней структуры ИМ), что в целом эта серия будет иметь приемлемые временные характеристики.

Одно из таких решений предложили на рубеже 1990-х годов Андерссон и Гальперин с Ривестом. Эти деревья получили название деревья с вычислением штрафа, или Scapegoat-деревья. Аналогично, как и для WB[α]-деревьев, фиксируется параметр сбалансированности α, $\frac{1}{2} \leq \alpha \leq 1$, и вершина называется α-сбалансированной, если величины $N(T_L)$, $N(T_R) \leq \alpha(N(T_L) + N(T_R) + 1)$ (см. рис. 11.9).

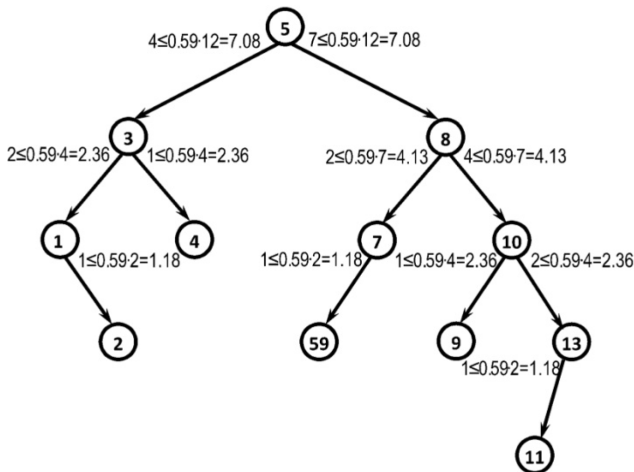


Рис. 11.9. Пример сбалансированного дерева с 12 вершинами и параметром α = 0.59, $h_{0.59}(12) = \left\lfloor \log_{\frac{1}{0.59}} 12 \right\rfloor = \lfloor 4.71 \rfloor = 4$

Крайние параметры α допускают идеально сбалансированные деревья (только их) и любые деревья соответственно. При таких условиях сбалансированности высота дерева не может превосходить $\log_{1/\alpha} N$. Таким образом, процедура вставки выглядит так:

1. Вставляем элемент, как это делается обычно для деревьев бинарного поиска, попутно вычисляя высоту нового элемента v.
2. Если $h_v > \log_{1/\alpha} N$ (т. е. условие сбалансированности нарушено), то приступаем к процедуре балансировки, иначе стоп.
3. Поднимаясь от нового элемента к корню, ищем обязательно существующую вершину-штрафника (scapegoat-вершину), т. е. вершину, для которой нарушено правило α-сбалансированности. Таких вершин на пути к корню может быть несколько и выбрать можно любую (рис. 11.10). Это дает возможность выбирать различные стратегии балансировки: первую попавшуюся, ближайшую к корню и т. д.

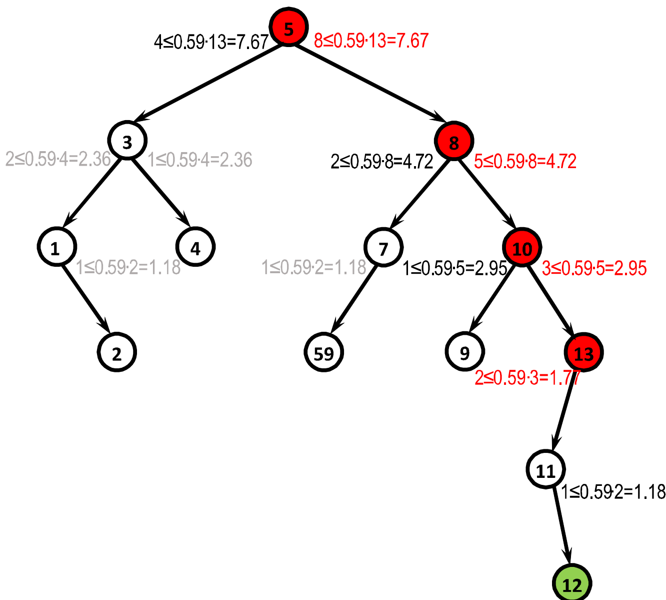


Рис. 11.10. В предыдущее дерево вставили число 12, высота дерева стала $h = 5$. Но с 13 вершинами высота должна быть $\boldsymbol{h_{0.59}(13) = \lfloor \log_{1/0.59} 13 \rfloor = \lfloor 4.86 \rfloor = 4}$, при этом вершины 13, 10, 8, 5 – нарушители

4. Выполняем процедуру идеальной балансировки дерева с корнем в вершине-штрафнике. Опять же, возможны различные подходы к этому. Простейший, но требующий выделения временного дополнительного массива, – это сохранить дерево в массив посредством его инфиксного обхода, а затем воссоздать дерево, моделируя его постфиксный обход посредством выбора медиан в массиве (см. рис. 11.11).

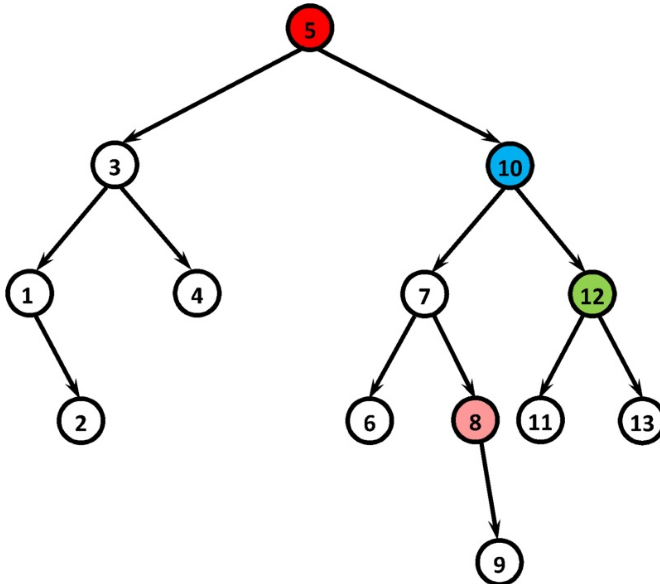


Рис. 11.11. Если выбрать для балансировки вершину 8, то получится дерево, изображенное на рисунке – поддерево с корнем в 10 является идеально сбалансированным. Отметим, что вершина 5 по-прежнему является нарушителем, хотя дерево сбалансированно

Так как вершиной-штрафником может оказаться корень дерева, то его перестройка может потребовать $O(n)$ шагов. Однако несколько последующих операций поиска/вставки/удаления будут иметь сложность $O(\log n)$.

Достоинством деревьев с вычислением штрафа является то, что не требуется записи дополнительной информации в узлы дерева. В отличие от других деревьев, они являются очень гибкими. Например, у них процедура балансировки не зависит от параметра α в отличие от WB[α]-деревьев, где разработчик должен на этапе

дизайна алгоритма выбрать диапазон допустимых значений α и под него разработать процедуру балансировки. Бо́льшее значение α приводит к меньшему количеству операций балансировки, что ускоряет вставку, но замедляет поиск и удаление. Для меньшего α верно обратное. Поэтому в приложениях параметр α может быть выбран динамически в зависимости от частоты выполнения тех или иных действий.

Деревья с вычислением штрафа начинают оптимизировать, если после вставки обнаруживается «очень» длинный путь в дереве. Для расширяющихся деревьев (Splay-деревья), предложенных Слейтером и Тарьяном в 1985 году, сбалансированность вообще не контролируется. При определенных условиях (в серии некоторых «однородных» операций) такое дерево может просто выродиться в линейный список. Однако серия «неоднородных» операций приведет к самобалансировке дерева, которая обеспечит высоту дерева, близкую к логарифмической, а значит амортизационная сложность поиска/вставки/удаления будет $O(\log n)$. Эти деревья обладают также следующим свойством, чрезвычайно полезным в некоторых приложениях: элементы, к которым обращаются чаще всего, скапливаются ближе к корню. Это означает, что доступ к таким элементам будет очень быстрый (их высоты будут небольшими). Существует гипотеза, что расширяющиеся деревья являются асимптотически оптимальными среди динамически самобалансируемых деревьев в смысле средней длины путей.

В этом контексте упомянем оптимальные деревья поиска. Цель – сконструировать дерево, в котором наиболее «интересные» элементы (т. е. те, которые мы ищем чаще всего) расположены как можно ближе корню, а менее «интересные» дальше. Оптимальность обеспечивается статически, в том смысле, что зная вероятности распределения N ключей заранее, алгоритм конструирует дерево (сложность $O(N^2)$, предложен Кнутом в 1971 году), которое дальше не изменяется (для него выполняются только операция поиска). Это пример недеструктивного ИМ (после инициализации используется только операция доступа). Отметим, что если ключи равновероятны, то это соответствует идеально сбалансированному дереву. Примеры использования – словари с известным распределением частот слов.

В-деревья, или сильноветвящиеся деревья

Возможен и иной подход к обеспечению для древесных структур данных эффективности операций вставка/поиск/удаление. В отличие от рассмотренных выше сбалансированных деревьев, в которых значения ИМ хранятся во всех узлах дерева и, соответственно, их высоты различны, в В-деревьях значения ИМ хранятся только в листьях, которые находятся на одинаковой высоте. Внутренние вершины используются только для организации структуры дерева. Кроме того, они могут быть небинарными, хотя остаются по-прежнему деревьями поиска. Первый вариант таких деревьев был предложен Хопкрофтом в 1970 году и получил название 2-3-дерева.

У каждой внутренней вершины 2-3-дерева либо 2, либо 3 потомка (1 быть не может). Ключи находятся в листьях, они упорядочены. Для каждой внутренней вершины известны максимальные ключи листьев, находящихся в первом и втором поддереве. Они используются при поиске для выбора поддерева (рис. 11.12), заведомо содержащего нужный элемент (если он есть в дереве).

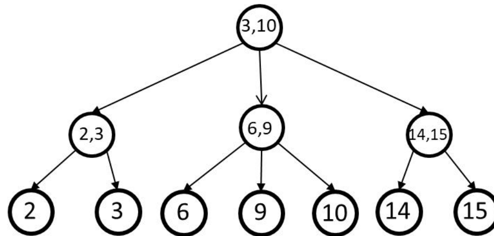


Рис. 11.12. Пример АВЛ-дерева

Если h – высота дерева, то количество хранимых объектов N (количество листьев) оценивается $2^{\mathrm{h}} \leq N \leq 3^{\mathrm{h}}$ (дерево в точности бинарное или тернарное). Таким образом,

$$0.63\log_2 N < \log_3 N \leq h \leq \log_2 N,$$

а значит сложность поиска в 2-3-дереве $O(\log n)$.

Какова же сложность поддержания структуры 2-3-дерева при вставке и удалении элементов? Оказывается, она аналогична сложности поиска. Рассмотрим вставку (рис. 11.13). Двигаясь от корня, мы за h шагов найдем внутреннюю вершину, наследником-листом которой должен стать новый элемент (если его еще нет в дереве).

Если у нее только два наследника, то проблем нет – вставляем его как соответствующее поддерево. Однако если имеется три наследника, то четвертым новый элемент не может быть по условию. В этом случае формируются две пары, каждая из которых состоит из двух листьев с подходящими значениями ключей (чтобы сохранялось упорядочение) и одной внутренней вершиной. Если внутренняя вершина на предыдущем уровне имела в запасе одну неиспользуемую ссылку на поддерево, то всё хорошо – подвешиваем, иначе снова повторяем операцию «рассечения» вершины. И так продолжаем двигаться дальше к корню. Удаление элемента выполняется аналогично.

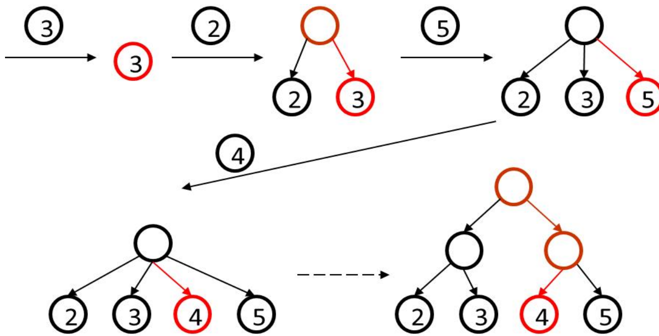


Рис. 11.13. Вставка последовательности 3, 2, 5, 4 в 2-3-дерево

Может сложиться впечатление, что такая конструкция дерева, хотя и гарантирует хорошую производительность, очень нерационально использует память, так как внутренние вершины служат лишь «стрелочниками». Однако легко убедиться (предлагаем это сделать самостоятельно), что хотя расходы на память действительно больше, но они вполне сравнимы с «обычными» деревьями бинарного поиска.

2-3-деревья являются частным случаем так называемых B-деревьев порядка m (данном случа $m = 3$). Они были предложены в 1970 году Байером и МакКрейтом. Они также называются *сильноветвящимися деревьями*, так как количество потомков у внутренних ершин может достигать m (и оно всегда не меньше $m/2$). На практике m может варьироваться от 100 до 4000 и больше. Внутренние вершины хранят массив пар <значение,адрес> длины m, упорядоченный по значениям (смысл такой же, как и в 2-3-деревьях). При движении к листьям для всякой внутренней вершины посредством бинарного поиска за $O(\log m)$ шагов можно найти требуемое поддерево. Высота дерева

степени m с N листьями оценивается $\log_m N \le h \le \log_{m/2} N$. При больших m внутренние вершины заполнены (использовано элементов массива пар) примерно на 69 %. В-деревья и их варианты интенсивно используются для организации быстрого доступа в файловых системах и системах управления базами данных (индексирование данных).

11.5.5. Хэш-таблицы

Зачастую и логарифмическая сложность операций с информационным множеством может оказаться недостаточно эффективной. Например, в процессе обработки текста программы компилятор практически на каждом шаге выделения лексемы проверяет, что это за лексема. Если со служебными словами более-менее все понятно, то для идентификаторов это может быть имя типа, переменной, константы, функции и т. д. Проверка может состоять в наличии их описания, которое не должно конфликтовать с уже существующими в этой области видимости именами, или правильности использования, но которое должно соответствовать правилам языка. Например, если язык требует, чтобы переменные были описаны перед их использованием, то информация об этом должна быть уже где-то сохранена. В данном случае имеет место ситуация, когда вставки (элемент описывается в области видимости) и удаления (описанный элемент покидает область видимости) относительно редки в сравнении с чрезвычайно интенсивными поисками информации об этом элементе. При этом можно отметить, максимальное количество элементов, которые потенциально могут появиться в области видимости, оценивается «разумной» величиной. Именно этот контекст – эффективная работа трансляторов – привела А.П. Ершова и А. Дюмэя в 1956 году к идее *хэш-таблиц* (таблиц расстановок в русскоязычной традиции).

Не уменьшая общности, можно считать, что универсум объектов составляют натуральные числа (возможно, очень большие). Идея состоит в том, чтобы «порубить» и «перемешать» исходное число с помощью так называемой *хэш-функции*, которая выдаст новое число. Это число, которое еще называется *хэш-кодом,* или просто хэшем исходного числа, должно вычисляться детерминированным образом (при повторных вычислениях результат один и тот же) относительно легко и быть относительно небольшим. Первое требование связано с тем, что перед нами стоит задача организовать «быстроработающее» ИМ, а значит вспомогательные вычисления не могут занимать много

времени. Второе требование определяется способом организации ИМ – это будет одномерный массив – таблица – с индексами от 0 до $M - 1$ (везде в этом параграфе M обозначает размер таблицы). Выбор хорошего M – довольно непростая задача в общем случае. Некоторые обстоятельства, которые принимаются во внимание, мы отметим ниже. Первое очевидное – длина массива должна быть адекватной ограничениям на память, накладываемым в приложении.

Читатель может заметить, что массивы уже появлялись в нашем рассмотрении ИМ, но ни неупорядоченные ни упорядоченные не смогли удовлетворить нас в целом. Для достижения лучших (при определенных сценариях использования) показателей хэш-таблицы отказываются от требования упорядоченности (что делает их близкими к неупорядоченным таблицам), но вычисленный хэш дает очень хорошее приближение для того индекса в массиве, по которому элемент должен находиться (что их приближает к упорядоченным массивам, в которых мы использовали знание об упорядочивании для поиска нужного места). Отметим, что отказавшись от упорядоченности, мы получили возможность использовать универсумы, где для сравнения элементов не требуется операции больше/меньше (линейный порядок на элементах). Достаточно уметь их сравнивать на равенство.

Так почему же всего лишь «хорошее приближение», а не точный индекс, где должен находиться/разместиться очередной элемент? Ответ на этот вопрос тесно связан с ответом, который уже должен возникнуть у пытливого читателя. Если попытаться отобразить более чем M различных чисел в интервал также различных чисел от 0 до $M - 1$ (найти биективную функцию) [42], то этого мы сделать не сможем, так как найдется пара (и даже больше в общем случае) элементов, у которых хэши совпадут. Это называется *коллизией* хэшей, и далее мы эту коллизию должны научиться решать каким-то не очень затратным способом. Поэтому вычисленный в первый момент хэш, он еще называется *первичным*, является в общем случае не окончательным местом размещения элемента, это лишь первое приближение. Два метода решения этой проблемы представлены ниже. Из вышесказанного следует, что еще одним важным свойством хэш-функций явля-

[42] Если количество элементов заранее известно, то можно построить так называемую *совершенную* хэш-функцию, которая в общем случае является инъекцией.

ется свойство равномерно распределять появляющиеся ключи по ячейкам хэш-таблицы с целью равномерного распределения коллизий. Такая хэш-функция называется *идеальной*. К сожалению, это очень сложная задача при отсутствии знаний об универсуме объектов хранения ИМ. Обычно довольно просто сконструировать набор «плохих» ключей, которые скомпрометируют быстро вычислимую хэш-функцию: все ключи будут отображены в один хэш. Однако при тщательном анализе предметной области такие риски можно свести к минимуму и построить довольно хорошую хэш-функцию.

Приведем примеры хэш-функций [31]:

1. $\mathrm{hash(k)} = k_1 \oplus k_2 \oplus \dots \oplus k_l$, $k_\mathrm{i} \in 0 \dots M-1$. k_i – это, как правило, части машинного слова (значения там хранящиеся), которые удобно выбирать. Обычно это байт или два байта, для которых $M = 2^8$ и $M = 2^{16}$ соответственно. Очень скоростной метод хэширования, при благоприятных обстоятельствах дающий хороший результат.
2. $\mathrm{hash(k)} = \mathrm{k}\, mod\, M$, где M – взаимно просто с основанием системы счисления, используемой в компьютере (т. е. обычно это не степень двойки), желательно простое число.
3. Биты ключа рассматриваются как коэффициенты полинома степени $\lceil \log \mathrm{k} \rceil$. Хэш – коэффициенты полинома-остатка от деления на специально подобранный полином степени m, $M = 2^\mathrm{m}$.
4. $\mathrm{hash(k)} = \lfloor M\{\mathrm{k}A\}\rfloor$, где A – иррациональная константа, например, золотое сечение $A = \frac{\sqrt{5}-1}{2}$, а внутренние и внешние скобки являются функциями взятия дробной и целой части вещественного числа соответственно. На практике иррациональную константу заменяют на подходящее рациональное Приближение A.
5. $\mathrm{hash(k)} = \sum_{\mathrm{i}=1}^{l} k_\mathrm{i} p^\mathrm{i} \bmod M$, где p – простое, обычно $p = 3, 5$, а k_i – это величины, выбор которых описан в пункте 1. Этот метод иногда еще называют методом Горнера, так как реализуется он аналогично вычислению значения полинома в точке методом Горнера с учетом того, что операции должны выполняться в модулярной арифметике по модулю M, которое взаимно просто с p.

Отметим, что хэш-функции имеют многочисленные применения и вне контекста организации ИМ (например, контроль ошибок при передаче информации или аутентификация текстов), и там к ним могут предъявлять некоторые другие требования.

Метод цепочек (прямого связывания)

В данном методе для размещения элементов используется дополнительная динамическая память, а ячейки хэш-таблицы служат своего рода входом в ИМ (существенно) меньшего размера, которые хранят элементы с одинаковым первичным хэшем. Реализации этих меньших ИМ могут быть разными. В классическом варианте при возникновении коллизии ключи с одинаковыми хэшами помещаются в один неупорядоченный список. В некоторых современных реализациях используются сбалансированные деревья (в Java 8 в HashMap это красно-черные деревья). Этот метод обладает такими хорошими свойствами, как эффективность вставки и удаления элемента.

Если хэш-таблица уже содержит N элементов, то сложность поиска в худшем (все элементы имеют одно хэш-значение) в ней при реализации со списками составит $O(N)$, а с деревьями $O(\log N)$. Оценим сложность в среднем для реализации с помощью списков.

Пусть хэш-таблица уже содержит N элементов и $q_i = \mathbb{P}(\mathrm{hash}(\mathrm{k}) = i)$ – вероятность того, что хэш-функция выдает хэш i для элементов из универсума. Напомним, для идеальной хэш-функции все эти вероятности равны $1/M$.

$$\mathbb{P}(\text{длина списка по } i\text{-му адресу равна } l = p_{\mathrm{i}}(l, N)) = \binom{\mathrm{N}}{l} q_{\mathrm{i}}^{l}(1 - q_{\mathrm{i}})^{\mathrm{N}-l}$$

$$\mathbb{P}(\text{хотя бы один список имеет длину } l) = P(l, N) = \sum_{\mathrm{i}=1}^{\mathrm{M}} q_{\mathrm{i}} p_{\mathrm{i}}(l, N) = \sum_{\mathrm{i}=1}^{\mathrm{M}} \binom{\mathrm{N}}{l} q_{\mathrm{i}}^{l+1}(1 - q_{\mathrm{i}})^{\mathrm{N}-l}$$

$$T_{\mathrm{ave}} = \mathbb{M}(\text{длин списков}) = \sum_{\mathrm{i}=1}^{N} lP(l, N) = \sum_{l=1}^{\mathrm{N}} l \sum_{\mathrm{i}=1}^{\mathrm{M}} \binom{N}{l} q_{\mathrm{i}}^{l+1}(1 - q_{\mathrm{i}})^{\mathrm{N}-l} = \sum_{\mathrm{i}=1}^{\mathrm{M}} \sum_{l=1}^{\mathrm{N}} l \binom{\mathrm{N}}{l} q_{\mathrm{i}}^{l+1}(1 - q_{\mathrm{i}})^{\mathrm{N}-l}$$

Принимая во внимание, что $l\binom{\mathrm{N}}{l} = N\binom{\mathrm{N}-1}{l-1}$ и вынося Nq_{i}^2, имеем

$$= N \sum_{\mathrm{i}=1}^{\mathrm{M}} q_{\mathrm{i}}^2 \sum_{l=1}^{\mathrm{N}} \binom{\mathrm{N}-1}{l-1} q_{\mathrm{i}}^{l-1}(1 - q_{\mathrm{i}})^{\mathrm{N}-l} =$$

$$= N \sum_{\mathrm{i}=1}^{\mathrm{M}} q_{\mathrm{i}}^2 \sum_{\mathrm{j}=0}^{\mathrm{N}-1} \binom{N-1}{j} q_{\mathrm{i}}^{\mathrm{j}}(1 - q_{\mathrm{i}})^{\mathrm{N}-\mathrm{j}-1} =$$

$$= N \sum_{\mathrm{i}=1}^{\mathrm{M}} q_{\mathrm{i}}^2 (q_{\mathrm{i}} + 1 - q_{\mathrm{i}})^{\mathrm{N}-1} = N \sum_{\mathrm{i}=1}^{\mathrm{M}} q_{\mathrm{i}}^2.$$

Таким образом, для идеальной хэш-функции имеем среднюю сложность:

$$T_{\text{ave}} = NM\frac{1}{M^2} = \frac{N}{M}.$$

Метод перехэширования (открытой адресации)

В данном методе дополнительная динамическая память не используется, элементы размещаются в одном массиве. В случае коллизии вычисляется новое хэш-значение (*вторичный* хэш) и в том же массиве по новому индексу проверяется ячейка. Если она занята, то процедура выполняется снова. В худшем случае нам надо перебрать все имеющиеся элементы (все они попали в одну цепочку перехэширования). В конце концов мы либо найдем ключ, либо свободное место, либо придем в исходную точку. Последнее возможно, если таблица заполнена (это можно выяснить и другим методом) или хэш-функция, выполняющая перехэширование, не очень удачная (она не перебирает все M индексов массива, начиная с некоторого). Если массив заполнен, то либо сообщается об ошибке, либо массив увеличивается и перестраивается (все накопленные значения перехешируются. Следует отметить, что удаление элемента – дорогостоящая процедура, так как если этот элемент удалить, то поиск последующих элементов может работать неправильно ввиду того, что индекс удаленного элемента входил в какую-либо цепочку перехэширования. Поэтому когда это оправдано по каким-либо причинам, элементы не удаляют «физически», это выполняется «логически» – они помечаются как удаленные. Это создает проблему, если таких «мертвых» элементов становится много, а работа с хэш-таблицей не предполагает ее перестройку при переполнении. Если предполагается перестройка таблицы, то в ее процессе «мертвые» элементы удаляются.

Перестраиваемые таблицы наиболее часто встречаются на практике, причем обычно перестроение случается (этот момент можно контролировать) задолго до полного заполнения таблицы. Вводится параметр *коэффициент загрузки* λ, который равен отношению вставленных элементов к общему размеру таблицы. При достижении заданного коэффициента загрузки таблица принудительно перестраивается.

Наиболее часто используемые функции перехэширования:

1. $\text{hash}_i(k) = (\text{hash}_0(k)+i) \bmod M$
2. $\text{hash}_i(k) = (\text{hash}_0(k) + i\Delta(k)) \bmod M$
3. $\text{hash}_{i+1}(k) = (a_1\text{hash}_i(k) + a_0) \bmod M$
4. $\text{hash}_{i+1}(k) = (a_2\text{hash}_i(k)^2 + a_1\text{hash}_i(k) + a_0) \bmod M$

Первый метод называется методом линейных проб, который хоть и является самым простым, обладает существенным недостатком – он приводит к вторичному скучиванию, когда после некоторой позиции в хэш-таблице начинают собираться другие элементы с тем же или близким хэшем. Наиболее качественным при правильном подборе коэффициентов a_n относительно M из этих простых методов перехэширования является четвертый.

Как уже было сказано, поиск в худшем при коэффициенте загрузки λ будет пропорционален λM. А какой будет сложность в среднем? Пусть размер таблицы M достаточно большой [43]. Последовательно вычисленные для ключа k индексы $\alpha_0=\text{hash}_0(k),\alpha_1,\alpha_2,\ldots$, с равной вероятностью распределены на интервале $0\ldots M-1$ независимо друг от друга. Рассмотрим вероятность, что длина цепочки проб равна i:

$\mathbb{P}$(нужно i проб для обнаружения присутствия/отсутствия ключа k = $\mathbb{P}$(ячейки по адресам $\alpha_0,\ldots,\alpha_{i-2}$ заняты, в α_{i-1} ключ или пусто) = $= \lambda^{i-1}(1-\lambda)$.

Таким образом, математическое ожидание количества проб при загрузке λ:

$$T_{\text{ave}} = \mathbb{M}(\text{длина цепочки}) = \sum_{i=1}^{\infty} i\lambda^{i-1}(1-\lambda) = \frac{1}{1-\lambda}.$$

Как будет себя вести метод перехэширования при постепенном заполнении хэш-таблицы, т. е. сколько в среднем будет вестись поиск на серии вставок элементов в таблицу (коэффициент загрузки

---

[43]Это предположение сделано для упрощения рассуждений, чтобы можно было перейти к «удобным» бесконечным величинам, которые не зависят от M. Но эту оценку можно провести и не делая этого предположения и используя M.

будет расти от 0 до некоторого λ)? Ответ дает теорема о среднем значении функции:

$$T_{\mathrm{am}}(\lambda) = \frac{1}{\lambda}\int_0^{\lambda} T_{\mathrm{ave}}(x)dx = \frac{1}{\lambda}\int_0^{\lambda} \frac{dx}{1-x} = \frac{1}{\lambda}\ln\frac{1}{1-\lambda}.$$

Сравнение временных сложностей поиска с перехэшированием для разного заполнения хэш-таблицы:

λ	**0.25**	**0.50**	**0.72**	**0.75**	**0.90**	**0.95**	**0.99**
$\mathbf{T_{ave}}$	1.33	2.00	3.57	4.00	10.00	20.00	100.00
$\mathbf{T_{am}}$	1.15	1.39	1.77	1.85	2.56	3.15	4.65

Отметим значение 0.72. Именно такой коэффициент был определен как оптимальный в исследованиях Microsoft и использован в их библиотечной реализации хэш-таблиц с перехэшированием. При достижении этой границы загрузки происходит автоматическое увеличение размера хэш-таблицы. Исследователи отметили, что при большем значении коэффициента загрузки происходит существенная деградация производительности.

Сравнение методов хэширования приведено в следующей таблице:

Метод цепочек	**Перехэширование**
Не бывает переполнения	Возможно переполнение
Коллизии при вставке разрешаются быстро	Разрешение коллизий может требовать времени
Накладные расходы, связанные с динамической памятью	Не требует дополнительной памяти
Эффективно, при равномерном перемешивании, эффективной работе динамической памяти и непредсказуемом количестве объектов хранения	Эффективно, когда есть возможность выделить массив размером в 1.5–2 раза больше, чем количество предполагаемых объектов хранения

В заключение отметим, что в программистском фольклоре хэш-таблицы часто представляются панацеей от всех проблем в части организации информационных множеств. В частности, утверждается, что время операций для хэш-таблиц $O(1)$. Это верно (в практическом смысле) при предположениях:

- хэши появляющихся ключей равномерно распределены и независимы (хэш-функция «хороша» для ожидаемых ключей);
- в случае метода цепочек размер таблицы «сравним» с количеством ключей (хотя и может быть меньше), а в случае перехэширования размер хэш-таблицы «много» больше, чем количество появившихся ключей, т. е. коэффициент заполнения не очень велик.

В общем же случае, как читатель мог убедиться, это не так.

12. СПЕЦИАЛЬНЫЕ МЕТОДЫ ПРОГРАММИРОВАНИЯ

К сожалению, ввиду ограниченности в изложении материала, в данном пособии не рассматривается важный метод программирования – программы, управляемые событиями (Event-driven programming).

12.1. Программы, управляемые данными (Data-driven programming)

Рассмотрим следующую простую задачу: написать программу, которая выводит «да», если входная цепочка состоит из четного числа нулей и четного числа единиц, «нет» – в противном случае. Задача имеет «очевидное» решение (в предположении, что ничего другого, кроме нулей и единиц, на входе не встречается):

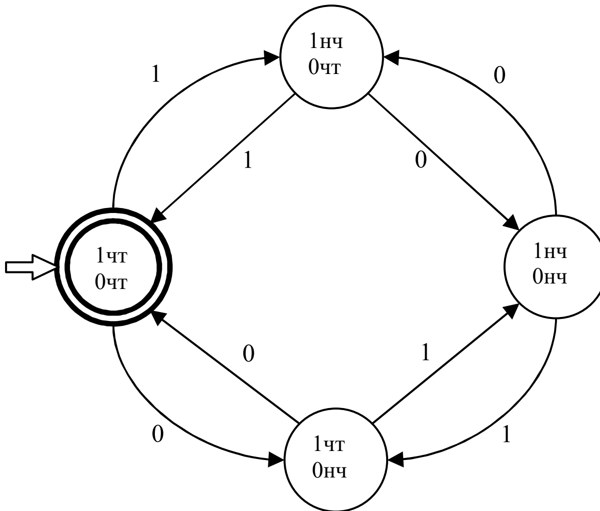
```
int  cnt0=0, cnt1=0;
char ch;
while ((ch=getc(stdin)) != EOF)
   if (ch==’0’)
      ++cnt0;
   else
      ++cnt1;
   if (cnt0%2 || cnt1%2)
      printf(“нет”);
else
      printf(“да”);
```

у которого, однако, легко найти недостаток – возможное переполнение счетчиков. Обычно на это возражают, что это маловероятно. Попытаемся решить ее другим способом, который, с одной стороны, лишен этого недостатка, с другой стороны, применим к широкому классу задач, относящихся к распознаванию цепочек, а с третьей, позволит обсудить некоторую общую интересную проблему.

Заметим, что какая бы часть входной цепочки ни была уже прочитана, наше «знание» о ней, достаточное для решения задачи, может находиться лишь в одном из четырех состояний:

- встретилось четное число нулей и четное число единиц;
- встретилось нечетное число нулей и четное число единиц;
- встретилось четное число нулей и нечетное число единиц;
- встретилось нечетное число нулей и нечетное число единиц.

При этом переход из одного состояния в другое определяется только очередным входным символом. Именно такое поведение характерно для известной нам вычислительной модели – конечного автомата [1, 16, 26, 40, 53]. Действительно, этот автомат легко построить из предыдущего описания:



Приведем для справки соответствующее регулярное выражение:

$$(00+11)^*((01+10)(00+11)^*(01+10)(00+11)^*)^*.$$

Программная реализация данного автомата могла бы иметь вид:

```
int state=0;
char ch;
while ((ch=getc(stdin)) != EOF)
   switch (state) {
   /*1чт 0чт*/ case 0: state=(ch=='0') ? 1 : 2; break;
   /*1чт 0нч*/ case 1: state=(ch=='0') ? 0 : 3; break;
   /*1нч 0чт*/ case 2: state=(ch=='0') ? 3 : 0; break;
   /*1нч 0нч*/ case 3: state=(ch=='0') ? 2 : 1;
   }
if (state == 0)
   printf("да");
else
   printf("нет");
```

Для такого простого автомата решение вполне допустимое. Однако если сложность описания конечного автомата возрастает, то, очевидно, будет расти и сложность оператора **switch**. Причем «разбухание» может происходить в двух направлениях: размер алфавита языка (может достигать тысяч символов) и количество состояний автомата (может достигать сотен тысяч). То, что это будет совершенно нечитаемый код – в этом сомнения нет, но и не всякий компилятор способен «переварить» операторы такого размера.

Для простоты дальнейших рассмотрений будем полагать, что наш автомат является детерминированным, полностью определенным, дочитывающим входную строку до конца и имеет одно заключительное состояние FINAL. Теория нам подсказывает, что всякий конечный автомат можно привести к такому виду. В частности, рассмотренный выше автомат уже является таковым. Авторы полагают, что реализация не полностью определенного, не дочитывающего до конца и имеющего несколько заключительных состояний автомата (а также имеющего алфавит отличный от {0,1}) не окажется для читателя слишком сложной задачей. Итак,

```
#define FINAL 0
/*таблица переходов автомата*/
int dfa[][]={{1,2},{0,3},{3,0},{2,1}};
int  state=0;
char ch;
/*автомат всегда дочитывает вход*/
while ((ch=getc(stdin)) != EOF)
   /*ch-'0' - декодирование алфавита*/
   state=dfa[state][ch-'0'];
/*вход допустим, если остановились в финальном состоянии*/
if (state == FINAL)
   printf("да");
else
   printf("нет");
```

Сравним два последних варианта решения. В первом варианте нам пришлось тщательно выписывать операторы и передачу управления между ними. В данном случае не составляет труда установить связь между размером полученной программы и параметрами автомата: размер пропорционален произведению количества состояний на мощность алфавита. Во втором варианте управление было «записано» с помощью данных – массива **dfa**, который представляет собой таб-

лицу переходов нашего конечного автомата. Операторы самой программы организованы простейшим образом и не зависят от управляющих данных. Вполне может оказаться и так, что эта программа, где сложные управляющие цепочки заменены на выборку данных из массива, будет работать быстрее. Легко видеть, что для любого другого массива, представляющего таблицу переходов, а значит, другой автомат, сколь сложным он ни был, логику самой программы менять не придется.

Рассмотренные примеры иллюстрируют концепцию программ, управляемых данными, т. е. парадигму data-driven programming.

Отметим следующее свойство последней программы, реализующей конечный автомат. Массив **dfa** не является «неотъемлемой» частью данного программного текста. Действительно, можно, например, вынести определение этого массива в независимый файл и подключать его с помощью директивы препроцессора для последующей совместной компиляции. Эта техника используется, например, в генераторах лексических анализаторов lex/flex/JLex/GPLex, когда по некоторому языку спецификаций регулярных выражений строится таблица переходов конечного автомата. Очевидно, что определение автомата можно считывать и из файла динамически после запуска программы, что увеличивает гибкость использования данной программы.

Также отметим, что аналогичным способом может быть решена задача синтаксического анализа: есть спецификация КС-грамматики, которую генератор синтаксического анализатора переводит в программный текст, который можно включить в программу. Примерами таких генераторов являются yacc/bison/ANTLR и т. д. Интересным и практически значимым подходом такого рода являются многочисленные SAX-парсеры, принадлежащие множеству XML-технологий [63, 64].

12.2. Предметно-ориентированное программирование (Domain-specific languages)

В предыдущем разделе можно заметить одну важную особенность. У нас снова возникла необходимость разработать некоторые языки (языки спецификаций регулярных выражений и КС-грамматик), для которых понадобились свои трансляторы (генераторы лексических и синтаксических анализаторов), которые в свою очередь порожда-

ли программный код на каких-то языках программирования, реализующий, по сути, интерпретатор конечного или магазинного автомата соответственно [44]. А уже этот программный код мы, вместе с другим нашим кодом, использовали для решения нашей задачи в целом.

Эта плодотворная идея – вместо построения единого языка, способного и синтаксически и семантически удовлетворить все потребности потенциальных пользователей, дать возможность создавать:

- понятные по синтаксису специалистам в предметной области методы спецификации данных и действий,
- которые адекватно описывали бы семантику (смысл) происходящего,
- и были при этом достаточно безопасными –

давно бродила в умах программистов и в конце концов была сформулирована в форме предметно-ориентированного программирования.

Прогресс объектно-ориентированного программирования и функциональных языков можно рассмотреть именно под этим углом. С течением времени разработчики программных систем осознали важность концептуального уровня используемых языков программирования. Для этого есть несколько причин. Возросшая мощность компьютеров и прогресс в теории и практике трансляции избавили от требования к языкам программирования «высокого уровня» быть «легко» транслируемыми в низкоуровневый машинный код, а значит, в языках могли появиться не только простейшие структуры данных и управляющие конструкции. Ужесточились сроки разработки программных систем, которых требовалось все больше и они становились все объемнее. Возможности «разжевывать» предметную область до уровня примитивных конструкций языков программирования – переменная, массив, вызов функции с заданными параметрами и так далее – на это уже просто не хватало времени. Появилось желание оперировать в программах конструкциями более

---

[44] Конечно же, такие языки спецификаций позволяют решать не только задачи распознавания лексем/синтаксических конструкций. В них также возможно специфицировать «реакцию» на появившийся элемент, т.е. действия, которые следует выполнить, если на входе прочитана та или иная лексема или идентифицирована та или иная синтаксическая конструкция.

высокого уровня, адекватно представляющими предметную область. Отметим, что это потенциально повышает надежность программ, так как процесс «разжевывания» поручается транслятору. Кроме того, программистский цех перестал быть закрытым элитарным сообществом [45]. Все больше людей вовлекалось в программирование, при этом они были специалистами в своей предметной области, но не слишком жаждали погружаться в биты, косвенную адресацию и точки возврата.

Автор С++ Б. Страуструп указывал [51, 52], что одним из побудительных мотивов для создания нового языка послужила его неудовлетворенность имеющимся программным инструментарием при работе над системой моделирования аэропорта. Конечно же, он не был первопроходцем. Можно сказать, что первым попытался выразить мысль о том, что язык программирования должен быть адекватным предметной области, был такой мастодонт компьютерной эры, как Кобол (и, в каком-то смысле, Фортран). Примерами могут служить операторы

```
MOVE ZERO TO CHECK-SUM
```

или

```
IF EMPLOYEE-HOURS IS GREATER THAN MAXIMUM
```

К сожалению, эта попытка ограничилась лишь лексикой языка программирования. Его организация управления и семантика исполнения напрямую соответствовали простейшей фон-неймановской вычислительной модели. А потребности реальных приложений окончательно превратили этот язык в бизнес-ассемблер.

Итак, одна из задач, которые решал Б. Страуструп при разработке С++, состояла в том, чтобы в языке появились удобные и эффективные средства для представления объектов и процессов предметной области, при этом сама предметная область не фиксировалась. В итоге был получен универсальный язык программирования, совместимый с предшественником. С помощью его довольно выразительных средств можно решать любые задачи, однако платой за

[45] Чтобы вспомнить, как строились взаимоотношения программистов и научных сотрудников, можно перечитать «Понедельник начинается в субботу» А. и Б. Стругацких.

это стала необходимость облекать объекты и процессы частной предметной области, рассматриваемой в данный момент, в формы частного объектно-ориентированного языка. Это снижает и эффективность отображения решаемой задачи в программу, и надежность этого отображения.

Полезно рассмотреть аналогию с развитием языков, предназначенных для адекватного отображения вычислительной модели. Вычисления, которые могут быть описаны и в λ-исчислении, и посредством нормальных алгорифмов, и логическим выводом в исчислении высказываний, могут быть реализованы программами, написанными на императивных языках программирования и исполняемыми на вычислителе фон-неймановской архитектуры. Однако появились языки программирования, которые напрямую оперируют концепциями соответствующей вычислительной модели. И это выражается на уровне самого языка (Лисп, Рефал, Пролог). Более того, появились «железные» реализации таких вычислительных моделей (например, Лисп-машины).

Можно сказать, что самым известным и широко распространенным примером предметно-ориентированного языка является SQL [12, 16, 17], хотя он и не всегда упоминается в этом контексте. Действительно, на SQL можно смотреть как на пример чрезвычайно удачного языка, специализирующегося в довольно узкой области – работа с таблично-организованными хранилищами данных (реляционными базами данных) [46], основы подавляющего большинства СУБД. Важно подчеркнуть, что это удачный пример как с точки зрения глубоко проработанной математической теории, лежащей в основании SQL, так и с точки зрения выдающихся технологических успехов, достигнутых при его практической реализации и внедрении.

Следует отметить важный урок, который преподносит SQL всем разработчикам новых предметно-ориентированных языков. «Классический» SQL (таковым иногда называют стандарт 1992 года

[46] То, что исходную предметную область (например, данные о студентах университета или описание банковских транзакций) еще надо сформулировать в терминах реляционной модели [12, 17], не отменяет того факта, что SQL – это предметно-ориентированный язык, где предметом являются отношения, оформленные в виде таблиц.

ANSI SQL-92) не является полным по Тьюрингу языком. Дальнейшие разработки позволили ликвидировать этот «недостаток». Например, были разработаны версии, расширяющие SQL процедурными возможностями, такие как Oracle PL/SQL, Microsoft SQL Server T-SQL или SQL with CTE and Windowing (SQL with Common Table Expressions and Windowing Functions). Стандарт SQL 2008 года, позволяющий писать рекурсивные запросы, также определяет полный язык. Конечно же, желание расширить SQL появилось не на пустом месте. Во-первых, при реализации некоторых приложений возникали ситуации, когда с помощью «классического» SQL-запроса не удавалось выразить полностью все условия для выборки данных, и поэтому результаты запроса приходилось обрабатывать дополнительно. Во-вторых, иногда сам запрос, приложив определенные усилия, сконструировать было можно, однако он получался настолько сложным, что дальнейшая работа с ним была затруднена и, как следствие, окончательное решение было аналогично предыдущему: простой SQL-запрос и дальнейшая постобработка. Можно сказать, что в этих случаях решение проблемы «невыразительности» SQL перекладывается на программиста, который должен решить, какую часть работы поручить SQL, а какую обработке на другом языке программирования, соотнеся их сложность и другие соображения.

Однако зададимся вопросом, всегда ли неполнота по Тьюрингу является недостатком? Если в языке можно выразить любую вычислимую функцию, то это означает, что мы потенциально получим в подарок весь букет неразрешимых/сложных алгоритмических проблем, среди которых укажем проблему остановки и проблему оптимизации [6, 21, 26, 27, 39, 40, 53]. Именно неполнота языка дает надежду (но не гарантирует), что эти сложные алгоритмические проблемы могут быть эффективно решены при его реализации. Многолетняя мировая практика использования «классического» SQL позволяет говорить о том, что в нем соответствующие проблемы получили удовлетворительное решение.

Предметно-ориентированному языку совсем необязательно существовать как отдельной языковой единице. Решения бывают различные.

Очень часто программирование на предметно-ориентированных языках, реализованных «в рамках» универсальных, – это типичное

программирование, управляемое данными: декларативное описание свойств предметной области с использованием лексики и синтаксиса языка подается на вход заранее написанных библиотек, реализующих семантику предметной области (например, существующие во многих языках библиотеки типа RegEx, реализующие работу с регулярными выражениями). Такой подход широко практикуется в языке Python [37, 38, 45], популярность которого во многом связано с богатством доступных библиотек для различных предметных областей.

Однако более системно эта идея нашла свое выражение в языках Kotlin [47] и в какой-то мере языке C# [61], где для этого используются средства самого языка, восходящие к объектно-ориентированным и функциональным корням этих языков. Это важно, так как позволяет вовлечь в процесс весь синтаксическо/семантический арсенал языка, что повышает надежность предметно-ориентированного языка.

Если (универсальный) язык программирования обладает развитыми лексическими и синтаксическими средствами, которые позволяют естественным образом описывать предметную область, для этой предметной области разработаны программные библиотеки, реализующие «динамику» предметной области, то целевую программу будет производить, возможно, с препроцессированием, транслятор с универсального языка. Можно отметить: довольно давно известный подход, основанный на макропроцессорах (например, описание типовых операций для визуальных средств Windows в Visual Studio C++ или параметрическая макрогенерация кода для алгоритмов типа преобразования Фурье), сильно различается на семантическом уровне с подходом Kotlin и C#. Даже более продвинутые, чем стандартный C/C++ препроцессор, макропроцессоры имеют слабое представление о семантике, что заставляет программиста работать с высочайшей аккуратностью.

13. ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

13.1. Последовательные, параллельные и совместные вычисления

Выполнение программы на одном из почти всех общеупотребительных языков всегда рассматривается как последовательный процесс, который выполняет вычислительное оборудование в соответствии с семантикой конструкций, определяемой эталонным описанием языка. Вместе с тем даже в рамках такого рассмотрения понятно, что отображение программы на реальный вычислитель не является однозначным. Так, последовательность операторов

$$a = 7; b = 10; \tag{1}$$

может выполняться в любом порядке или даже одновременно, если оборудование имеет такую возможность: доступны два независимых процессора. При прочих равных условиях все эти последовательности приведут к одному и тому же результату. Обобщая это наблюдение, заметим, что нарушение семантических предписаний, например, таких как регламент выполнения последовательности операторов в том порядке, в котором они написаны в программе, иногда допустимо, а потому с целью ускорения вычислений или по каким-либо иным причинам можно подменять вычисления исходной программы другими, более предпочтительными.

Главное условие отхода от заданного языком эталонного выполнения программы – сохранение вырабатываемых результатов при одних и тех же вводимых данных. Для определения допустимости отступления от языковых регламентов вычислений требуется специальный анализ, показывающий, что с точностью до времени выполнения и занимаемой при этом памяти при любых перерабатываемых данных будет получаться тот же самый результат, что и при выполнении исходной программы. Такой анализ покажет, что в следующем фрагменте:

$$a = 7; b = a + 10; \tag{2}$$

зависимость одного оператора от другого не разрешает изменять заданный исходной программой порядок вычислений.

В качестве альтернативы как строгому заданию порядка вычислений, так и порядку, определяемому компилятором, мог бы стать фиксируемый пользователем порядок. Средства такого рода иногда предлагаются языками. Например, заключение фрагмента (1) в специальные скобки

$$\mathbf{parbegin}\ a = 7; b = 10; \mathbf{parend} \quad (3)$$

требует, чтобы компилятор организовал параллельное выполнение двух операторов, если это возможно, или обеспечивал тот же эффект, когда ресурс двух процессоров отсутствует.

Понятно, что подобное преобразование фрагмента (2)

$$\mathbf{parbegin}\ a = 7; b = a + 10; \mathbf{parend} \quad (4)$$

сделает результат выполнения неопределенным. В этом случае компилятор должен выдать соответствующее предупреждение о возможной ошибке.

Интересный подход в обсуждаемом примере демонстрирует Алгол-68, в котором декларируется следующие варианты выполнения последовательности операторов (предложений – в терминологии языка):

- *Последовательное вычисление*. Последовательность предложений, разделяемых символами «;» выполняется *в заданном* в тексте программы *порядке*, если она не заключена в скобки **parbegin** и **parend**. Этот порядок может быть нарушен компилятором только в том случае, когда можно гарантировать, что результаты вычислений в заданном порядке и в новом порядке будут совпадать;
- *Параллельное вычисление*. Последовательность предложений, разделяемых символами «;», обрамленная скобками **parbegin** и **parend**, выполняется *параллельно*, если для этого имеется процессорный ресурс, а в противном случае порядок выполнения определяется компилятором;
- *Совместное вычисление*. Последовательность предложений, разделяемых символами «,» может выполняться *в произвольном порядке*, в том числе и параллельно, если это возможно.

Соглашение о совместности вычислений распространяется на вычисление аргументов коммутативных операций, а для некоммутативных операций предписывается последовательное вычисление.

Понятие совместности согласуется с тем, как трактуется конструкция списка фактических параметров процедуры: если предложения, представляющие параметры, разделяются запятыми, то они вычисляются совместно, т. е. могут готовиться к подстановке в любом порядке, включая параллельность; использование в качестве разделителя параметров символа «;» предписывает строго последовательную подготовку.

Возможность совместности вычислений некоторых фрагментов программы «развязывает руки» разработчикам-компиляторам, позволяя им выбирать отображение таких фрагментов на реальный вычислитель без специального анализа. Все издержки такого решения оказываются на программисте – ему придется следить за корректностью вычислений самостоятельно. Наряду с передачей параметров проблемы могут возникать в выражениях, поскольку вычисление операндов операций языка обычно определяется как совместное, иногда даются варианты. Например, в C/C++ определены две логические операции для конъюнкции: & и &&. Для первой из них операнды вычисляются совместно, а для второй явно говорится, что правый операнд не вычисляется при истинности значения левого операнда, что, в частности, запрещает параллельное вычисление операндов.

13.2. Разработка параллельных программ

До сих пор мы говорили о параллелизме без привязки этой возможности к методам разработки программ. Для этого аспекта возможны два принципиально разных подхода:

- *Распараллеливание* – наиболее распространенный подход, характеризующийся построением последовательной программы, которая для проведения расчетов с использованием нескольких или даже многих доступных процессоров преобразуется в параллельно выполняемую. Здесь речь идет не об автоматическом компиляторном распараллеливании, поскольку разработчик независимо от такой оптимизации строит последовательную программу. Использование встраиваемых в программу процедур, которые реализуют параллельно выполняемые алгоритмы, не всегда стоит считать методом распараллеливания: если при конструировании программы процедура подменяет последователь-

ный фрагмент разрабатываемой программы, то это естественный прием распараллеливания, но когда она сводится к свертке требуемого алгоритма без анализа метода его реализации, то о распараллеливании говорить не приходится. Пара фрагментов (1) и (3) демонстрирует распараллеливание, а (2) и (4) указывает на то, что этот подход нуждается в соответствующем анализе;

- *Параллельное программирование* – подход, при котором программа строится как реализация параллельно выполняемого алгоритма. Построение таких программ предполагает использование средств задания параллельных вычислений, например, блока параллельных вычислений, который оформляется с помощью скобок **parbegin** и **parend**, предписывающих одновременное выполнение элементов заданной в этих скобках последовательности действий. После завершения всех этих элементов выполнение становится последовательным, совместным или снова параллельным, что определяется языковой конструкцией, использующей данный параллельный фрагмент. Если ресурс для параллельного выполнения программы на реальном вычислителе отсутствует, то на системном уровне должен быть обеспечен эффект параллелизма, в частности, недетерминизм вычислений (см. фрагмент (4)). В этом случае параллелизм имитируется, например, с помощью режима *квантования времени*: параллельно выполняемым процессам определяются кванты времени использования процессора, исчерпание кванта приводит к приостановке процесса и прерыванию с передачей процессора для выполнения другого процесса, который возобновляется в приостановленном состоянии.

Понятно, что средства задания параллелизма не ограничиваются скобками **parbegin** и **parend**, но уже эта возможность позволяет дать содержательный пример. Пусть программа должна работать на оборудовании, которое не всегда функционирует корректно. В этом случае для проверки правильности результата можно воспользоваться дублированием счета, которое естественным образом оформляется следующим образом:

parbegin *R*1= <Действие>; *R*2==<Действие>; **parend**; (5)

if (*R*1< >*R*2) **then** {*R*3 = <Действие>; <Выбор совпадающего результата>}

Этот прием корректен, если вычисление Действий не приводит к побочным эффектам. Обратим внимание на то, что здесь не существенно, будут ли операторы, вычисляющие $R1$ и $R2$, выполняться параллельно или параллелизм имитируется. В то же время, важно заметить, что при решении использовать фрагмент (5) программист думает о нем, как о параллельном.

Более интересен следующий пример, который рассчитан на то, что параллельное вычисление будет реально выполняться. Пусть имеются два алгоритма, первый из которых выполняется быстрее на одних данных, а второй – на других. Если к тому же проверка особенностей перерабатываемых данных чрезмерно сложна, то вместо нее можно попытаться запустить два алгоритма Действие1 и Действие2 параллельно. Один из них закончит работу раньше другого, и требуемый результат будет получен:

$$\mathbf{parbegin} < \text{Действие1} >; < \text{Действие2} >; \mathbf{parend}; \qquad (6)$$

Здесь, как и прежде, параллельно выполняемые действия не должны влиять друг на друга, и, в частности, должно быть обеспечено экранирование изменений исходных данных. В противном случае одно из действий будет работать с искаженной информацией. Это можно сделать разными способами, которые предлагаются для самостоятельной разработки.

К сожалению, приведенное решение некорректно по двум причинам. Во-первых, фрагмент (6) будет выполняться по худшему варианту, так как семантика параллельного блока требует завершения всех параллельно выполняемых процессов, а во-вторых, не предусмотрен способ передачи результата вычисления для дальнейшего использования.

Первая ошибка может быть исправлена уничтожением процесса, реализующего одно из действий, которое перестает быть нужным. Здесь мы воспользуемся специальным оператором **kill**, который завершает выполнение процесса, передаваемого ему в качестве параметра, пока не вдаваясь в обсуждение того, как решается задача уничтожения. Для исправления второй ошибки нужно обеспечить сохранение результатов обоих действий в контексте фрагмента и выбора для дальнейшего использования полученного результата.

С учетом этих замечаний решение поставленной задачи принимает следующий вид:

```
TR R;                   // Описание переменных для хранения результатов   (7)
                        // Действий 1 и 2. Значение R определяется
                        // при выполнении < Действие1 > или < Действие2 >.
                        // TR – тип требуемого результата,
                        // который должен быть описан заранее.
int p12 = 0;            // Этой переменной выставляется значение 1, если
                        // результативно < Действие1 >, 2, если
                        // результативно < Действие2 >, и 0 в противном случае.
parbegin
   M1: {                   // < Действие1 > оформляется как процесс с именем M1.
      TRr;                 // r – локальная в M1переменная, используемая для
                           // временного хранения результата.
      <Действие1>;  // Определен требуемый результат, сохраняемый в r.
      kill (M2); p12 = 1; R = r;                                        (7.1)
      };
   M2: {                   // < Действие2 > оформляется как процесс с именем M2.
      TRr;                 // r – локальная в M1переменная, используемая для
                           // временного хранения результата.
      <Действие2>;  // Определен результат, сохраняемый в r.
      kill (M1); p12 = 2; R = r;                                        (7.2)
      };
parend;
```

С точностью до уничтожения процессов этот фрагмент наглядно решает обсуждаемую задачу. В хороших традициях повышения надежности программ он предусматривает формирование кода завершения: значения переменной p12. Если оно равно 1 или 2, то результат получен < Действием1> или <Действием2> соответственно, а значение 0 указывает, что результат не получен. В то же время корректность решения зависит от того, какая семантика связывается с оператором **kill**. Ошибка может возникнуть, если после того, как начал выполняться и еще не окончился один из них, активизируется другой. В этом случае оба процесса окажутся уничтожены и, как следствие, значения переменных p12 и R окажутся неопределенными. Эта недопустимая ситуация должна быть исключена.

По поводу реализации **kill** можно предложить различные варианты. Первый, не самый лучший, но достаточно понятный – воспользоваться проверкой переменной p12 в критических циклах, реализующих алгоритмы < Действие1 > и < Действие2 >. Как только она оказывается равной 2, завершается первое, а при p12 = 1 – второе действие. Равенство p12 нулю после выполнения фрагмента означает ошибку: по каким-то причинам одно из действий до своего естественного окончания принудительно завершило выполнение параллельного блока. Недостаток этого варианта в том, что он требует модификации программ альтернативных алгоритмов. Уменьшить необходимые изменения программ алгоритмов можно, если воспользоваться методом оперирования исключениями, в рамках которого завершение одного из действий после окончания его выполнения выставляет исключение для завершения другого. Это требует дополнения принудительно завершаемого действия реакцией на исключение, которая в данном случае сводится к прекращению программы альтернативных алгоритмов.

Самый простой путь устранения ошибки без модификации программ альтернативных алгоритмов возможен при централизованном управлении параллелизмом. В этом случае **kill** соотносится с управляющим компонентом, где этот оператор реализует следующее:

- приостановка процесса, активизировавшего **kill**. Другой процесс продолжает выполнение своей программы, но выполнить **kill** он не сможет, так как этот оператор занят (как это достигается, обсудим чуть позже);
- завершение процесса, заданного параметром **kill**, из контекста управляющего компонента;
- активизация приостановленного процесса (выполнение оператора, следующего за **kill**).

Если централизованное управление не предусмотрено, а это для многопроцессорных вычислительных систем скорее правило, чем исключение, нужны специальные средства блокировки вычисления второго оператора **kill**, как только первый из них начинает выполняться. Проблема в том, что **kill** перестает быть единственным и общим для системы оператором. Для отражения этого различия операторов в нотации будем обозначать операторы **kill**, относящиеся к процессам M1 и M2, соответственно как M1.**kill** и M2.**kill**. При

желании можно считать их методами классов этих объектов. Задание взаимоисключающего выполнения этих операторов можно сделать, например, с помощью семафора Дейкстры – специального типа данных, для которого определены две неделимые операции (их выполнение невозможно прервать):

- **down** (S) (или P (S)) – проверяет и изменяет значение семафора S: (8)
  если S > 0, **то** S = S – 1
  иначе {процесс блокируется, операция считается незавершенной}
- **up** (S) (или V (S)) – увеличивает значение семафора S на 1:
  {
  S = S + 1;
  если есть заблокированные процессы,
  то{разблокируется один из них и завершается его операция
  down (S): S = S – 1}
  }

Отметим, что операции **down** и **up** являются системными вызовами, и именно это позволяет говорить об их неделимости. Задача синхронизации процессов может быть решена без использования таких операций, т. е. с помощью специальной организации вычислений в программе. Классическую реализацию взаимоисключающего выполнения процессов демонстрирует алгоритм, предложенный Деккером. Хотя это решение следует рассматривать скорее в качестве теоретического доказательства осуществимости синхронизации, нежели как практический метод, мы настоятельно рекомендуем самостоятельно ознакомиться с алгоритмом Деккера.

В нашем примере в контексте, в котором представлен параллельный блок, нужно описать и инициализировать максимально возможное значение семафора:

sem S=1; // если нуждаются в синхронизации N процессов, то **sem** S=N–1;

и модифицировать строки (7.1) и (7.2):

down (S); M2.**kill**; p12 = 1; *R* = *r*; **up** (S); (7.1')

...

down (S); M1.**kill**; p12 = 2; *R* = *r*; **up** (S); (7.2')

Децентрализованное взаимоисключающее выполнение операторов достигается следующим образом. Первый из процессов, дос-

тигших при вычислении **down** (S), получает общую для двух процессов переменную со значением 1, и это приводит к тому, что S становится равной 0. Далее он вызывает **kill**, относящийся ко второму процессу и выполняемому на его процессоре. Если второй процесс успевает привести исполнение программы к **down** (S), он блокируется (см. (8)) и не сможет выполнить искажающие присваивания значений переменным p12 и *R*, а поскольку **kill** уничтожит второй процесс, выполнение параллельного блока будет корректным. В обсуждаемом примере можно обойтись без оператора **up** (S); поскольку применение семафора связано с оператором **kill**, который ликвидирует параллельное выполнение процессов. Но в таком случае нельзя было бы использовать семафор S в дальнейшем.

Часть программы, защищаемая семафором от одновременного использования разными параллельно выполняемыми процессами, называется *критической областью*. В нашем примере это операторы **kill**, вызываемые в двух процессах, и присваивания значений переменным p12 и *R*. Заметим, что поскольку в результате одного из вызовов остается только один выполняемый процесс, включение присваивания значений переменным p12 и *R* избыточно – всегда будет выполняться только одна из этих пар операторов. Это наблюдение приводит к следующему варианту строк (7.1) и (7.2):

down(S); M2.**kill**; **up** (S); p12 = 1; *R* = *r*; (7.1")

...

down (S); M1.**kill**; **up** (S); p12 = 2; *R* = *r*; (7.2")

В обсуждаемом примере можно вообще обойтись без оператора **up** (S); поскольку применение семафора связано с оператором **kill**, который ликвидирует параллельное выполнение процессов. Но в таком случае нельзя было бы использовать семафор S в дальнейшем.

Понятно, что в общем случае нужно стремиться к точному заданию критических областей. Так, если бы в примере защищалось не уничтожение параллельной ветви, а какое-либо иное действие, завершающееся этими присваиваниями, то нужно было бы при задании критической области использовать строки (7.1') и (7.2') – в противном случае из-за параллельного продолжения выполнения двух процессов результирующие значения переменных p12 и *R* оказались бы не определенными.

Приведенный пример иллюстрирует использование так называемого *считающего семафора*, который в состоянии синхронизировать два и более процессов. Для данной задачи можно было воспользоваться так называемым *мьютексом*, или *бинарным семафором*, который организует взаимное исключение двух процессов. По своей сути мьютекс – это обычный семафор Дейкстры, который может принимать только два значения, но его реализация может быть более эффективной.

Резюмируя обсуждаемый пример, отметим, что как при централизованном, так и при децентрализованном управлении синхронизация процессов связана с использованием общих данных. В первом случае это делается неявно, через контекст: критическая область, которая запрещена для одновременного выполнения двумя процессами, скрыта внутри программы kill, где это условие, вполне вероятно, обеспечивается с помощью тех же семафора и операторов down и up. Во втором – мы явно поместили синхронизирующий семафор в общий контекст, и предложенное решение будет работать на вычислительных системах с общей памятью. Для распределенных вычислений на автономных процессорах, т. е. в случае, когда каждый процессор имеет прямой доступ только к своей памяти, нужен другой механизм синхронизации, который связан с передачей сообщений между процессорами (он будет рассмотрен вскоре).

13.3. Порождение параллельных процессов и передача сообщений между процессами

Заметим, что использование нашей нотации неявно предполагает наличие памяти с общим доступом: параллельный блок встроен в единый для всех процессов контекст, а его завершение, т. е. окончание всех параллельно выполняемых операторов (несущественно, каким образом это происходит) влечет за собой переход в режим последовательных вычислений. При задании параллелизма для распределенных вычислительных систем более предпочтительны другие средства и нотации, следовательно, и другой семантики, которые в состоянии обеспечить возможность работы автономно выполняемых процессов на независимых процессорах.

Основой всех таких подходов являются специальные операторы, порождающие новые процессы, которые могут выполняться автономно, т. е. независимо от их родительского процесса, а также

механизм передачи сообщений между процессами. Последовательность задач, которые должны решаться при порождении процесса, сводятся к следующему:

- *Подготовка нового процесса* к возможности запуска на свободном процессоре:
  - формирование информации о новом процессе, необходимой для обеспечения доступа к нему в дальнейшем (идентификатор процесса и паспорт процесса – заполняется, когда процесс назначен для выполнения на выделенном для него процессоре),
  - формирование структур данных нового процесса (строится по программе, описывающей процесс);
- *Назначение ресурсов* для выполнения нового процесса или постановка процесса в очередь на активизацию, когда процессор требуемого типа освободится, если он занят, или когда новый процесс заблокирован по иным причинам;
- *Активизация нового процесса* и получение информации о выделенных ему ресурсах для заполнения паспорта процесса.

В результате порождения нового процесса у родительского процесса остается информация, необходимая для влияния на порожденный процесс и для получения данных от него. Эту информацию можно предавать другим процессам, и, тем самым, могут строиться произвольные *вычислительные сети*, связи которых реализуются путем *передачи сообщений* между процессами. Как главная программа, дающая начало всем последующим порождениям процессов-потоков, так и другие родительские процессы могут завершаться до того, как завешаются порождаемые ими потоки. При завершении выполнения программы процесса он сохраняется в неактивном состоянии, т. е. как структуры данных, доступных для других процессов. При этом возможность возобновить программу процесса остается. Когда этого не требуется, процесс уничтожается, а занятый им процессор освобождается и предоставляется одному из процессов, ожидающих активизации в очереди. Все эти задачи решаются с помощью специальных средств, которые обеспечивают формирование сети, адекватной специфике требуемых вычислений.

Обычно процессы вычислительной сети, порождаемые, активизируемые, выполняемые, завершаемые и уничтожаемые в соответствии с представленными соглашениями, называются *потоками*, или *нитями* (threads). Потоки взаимодействуют между собой с помо-

щью *передачи и приема сообщений*, и это единственный механизм взаимодействия. В частности, у потоков нет общей памяти, и их процессоры могут работать автономно. Если для потоков требуется централизованное управление, то оно организуется с помощью выделения управляющего потока (часто его роль выполняет главная программа main), который определяет нужную логику взаимодействий. Для управления отображением потоков на процессоры потокам могут назначаться приоритеты, которые регулируют продвижение потоков в очередях на активизацию.

13.4. Механизм send-receive

Передача сообщений между потоками по сравнению с семафорами является более общим механизмом – она позволяет синхронизировать потоки без предположения наличия доступа к общей памяти. Основная функциональность этого механизма реализуется двумя примитивами, реализующими, соответственно, *посылку* и *прием* сообщения:

- **send** (destination, message) – передать сообщение (message) потоку-получателю, заданному параметром destination;
- **receive** (source, message) – получить сообщение (message) от потока-источника, заданного параметром source.

Как и семафоры, эти примитивы являются системными вызовами, а не конструкциями языка. Их использование должно быть подчинено определенной дисциплине, которая делает обмен сообщениями между потоками корректным, т. е. исключает ситуации, когда процессу предписано получить сообщение, которое еще не послано, или наоборот, когда посылается сообщение для получателя, который не готов к приему. Поэтому обе **send** и **receive** могут быть *блокирующими* или *неблокирующими*:

- блокирующий **send** обеспечивает блокировку процесса-отправителя до тех пор, пока процесс-получатель не готов к получению сообщения (т. е. пока он не вызовет **receive**);
- неблокирующий **send** отправляет сообщение без проверки готовности процесса-получателя к приему (в этом случае возможно, что сообщение никогда не будет получено);
- блокирующий **receive** обеспечивает блокировку процесса-получателя до тех пор, пока процесс-отправитель не пошлет сообщение (т. е. пока он не вызовет **send**);

- неблокирующий **receive** не препятствует продолжению выполнения процесса-получателя в тех случаях, когда сообщение не поступает.

Из этих определений видно, что если **send** является неблокирующим, то чтобы процессу-отправителю узнать, получено ли его сообщение, когда это необходимо, требуются дополнительные действия, а именно: процесс-получатель должен прислать, а процесс-отправитель получить сообщение, подтверждающее получение первоначального сообщения. Возможны и в другие проблемные случаи, связанные с блокировкой. Например, когда используется блокирующий **receive**, а процесс-отправитель сообщение не посылает, получатель может оказаться заблокированным навечно. Чтобы такие ситуации не возникали, вместо блокирующего **receive** часто используется специальный примитив, позволяющий получателю проверить, есть ли для него сообщение, и вызывать **receive** только тогда, когда это условие выполнено.

В зависимости от целей использования передачи и приема сообщений могут быть полезны различные комбинации этих условий:

- блокирующий **send** и блокирующий **receive** – это так называемая «схема рандеву». Она не требует буферизации сообщений и часто используется для синхронизации процессов;
- неблокирующий **send** и блокирующий **receive** – это обычный прием в системах клиент/сервер: серверный процесс блокируется в ожидании очередного запроса для обработки, в то время как клиент, пославший запрос серверу, может продолжать выполняться, не ожидая окончания обработки своего запроса;
- неблокирующий **send** и неблокирующий **receive** – эта схема предполагает, что при необходимости проверка корректности передачи и приема сообщений организуется специально. В любом случае оба процесса могут продолжать выполнение, не дожидаясь окончания коммуникации.

Приводимая ниже программа демонстрирует решение задачи об альтернативных алгоритмах с помощью механизма **send–receive**. Прежде всего отметим, что планирование выполнения Действия 1 и 2 в разных потоках и на разных процессорах приводит к решению о том, что оба потока создаются как дочерние от главного потока main. Последний будем считать потребителем результата одного из Действий. Этот поток будет выполняться на третьем процессоре.

Чтобы не перегружать решение деталями, мы опускаем часть программы, которая занимается созданием потоков и назначением для них процессоров. Это делается в потоке main, который создает два потока M1 и M2, предназначенных для выполнения альтернативных алгоритмов Действий 1 и 2:

```
// Контекст главного потока:
TR R; // Описание переменных для хранения требуемых результатов Действий 1 и 2
int p12 = 0;// и индикатора результативности, назначение которого то же, что и прежде.
…
// Главный поток
void main ()
{
    T_Passport Pr1, Pr2;
    // Переменные для идентификации процессоров, выделенных потокам.
    …   // Определение процессоров, для исполнения потоков и значений Pr1 и Pr2.
    create (M1, Pr1, …);
    // Создание потока для <Действие1> и <Действие2>. Отсутствующие параметры
    create (M2, Pr2…);
    // задают информацию, связанную с предоставлением процессоров потокам.
    activate (M1);
    activate (M2);
    while p12 == 0 {
      // Цикл ожидания сообщения (только одного!) от дочерних потоков
      receive (M1, &p12);
      //Оба оператора receive не препятствуют продолжению выполнения потока main.
      receive (M2, &p12);
      // Цикл прекращается, когда сообщение будет получено одним из операторов.
    }
    if (p12 == 1) {              // Распознавание отправителя сообщения
            receive_block (M1, &R);            // Ожидание получения результата.
    }                                                                  (9)
    else {
            receive_block (M2, &R); // Ожидание получения результата.
    }                                                                  (10)
    … // Дальнейшие действия с результатом счета, полученным в одном из потоков.
}
```

Предложенный подход не предусматривает ликвидацию потоков в среде его выполнения: нет никакой информации о том, что Действие 1

или 2 должно прекратиться, когда результат уже получен. Эту ошибку мы ликвидируем после рассмотрения программ потоков M1 и M2:

```
// Рабочий поток M1
function Thread M1 {
                    TR r;
//Локальные переменные r, используемые для хранения результатов.
                    < Действие1 >;
//Определен требуемый результат. Он должен быть сохранен в r.
                    send_block (main, 1);
// Потоку main посылается сигнал о том, что результат получен.
                    send_block (main, r);
//Блокировка используется для ожидания готовности main к приему сообщения.
}

// Рабочий поток M2
function Thread M2 {
                    TR r;
//Локальные переменные r, используемые для хранения результатов.
                    < Действие2 >;
//Определен требуемый результат. Он должен быть сохранен в r.
                    send_block (main, 1);
// Потоку main посылается сигнал о том, что результат получен.
                    send_block (main, r);
//Блокировка используется для ожидания готовности main к приему сообщения.
}
```

Вернемся к отмеченной выше ошибке. Исправить ее в рамках потоков M1 и M2 нельзя, так как они в принципе ничего не знают о существовании друг друга. Если не имеет значения, что один из использованных процессоров будет выполнять бесполезную работу, то можно оставить всё как есть. В противном случае требуется принудительная ликвидация M1 или M2. Как и ранее, нежелательно при этом затрагивать алгоритмы Действий 1 и 2.

Если система поддержки распределенных вычислений позволяет родительскому потоку ликвидировать дочерние потоки, выполняемые на автономных процессорах, то она берет на себя все необходимые действия. И единственное, что нужно предусмотреть, – вызов операторов **Destruct** для каждого рабочего потока, который

перестал быть нужным. В нашем случае достаточно в поток main добавить в конце каждой ветви условного оператора (перед строками 9 и 10) следующее:

Destruct (M2);

и

Destruct (M1);

Возможно, что ликвидация дочерних потоков поддерживается лишь для тех из них, которые выполняются тем же процессором, что и родительский. В этом случае нужно подготовить контекст для ликвидации потоков еще при их создании, а принудительное завершение дочернего потока организовать путем посылки сообщения о ликвидации. Вполне разумно с этой целью организовать на рабочих процессорах специальные управляющие потоки, который ожидают сигнала о ликвидации. При получении сигнала управляющий поток выполняет вызов оператора **Destruct** для ликвидируемого потока и, если это необходимо, освобождает процессор.

Мы разрабатывали программу, рассчитывая, что параллельное выполнение альтернативных действий, вырабатывающих один и тот же результат, будет эффективнее за счет избавления от предварительного анализа данных. С точностью до того, как реализуются эти действия, для такого метода повышения скорости получения результата достаточно двух процессоров. Третий процессор играет роль распределения вычислений и данных управления. Другие процессоры не будут загружены. А что будет, если используется только один процессор? Прежде чем ответить на вопрос, заметим, что предложенная программа устроена так, что результат будет получен, пусть даже не так быстро, и в этом случае. Однако будет или нет достигнут эффект ускорения по сравнению с решением, требующим предварительного анализа данных, зависит от стратегии организации вычислений. Рассмотрим некоторые из вариантов такой организации:

- Второй процесс объявляется *более приоритетным* по отношению к первому. Это означает, что во всех случаях конкуренции двух процессов, когда программа явно не предписывает, какой из них нужно активизировать, процессор будет выделяться второму из них. Применительно к нашей программе это означает следующее. После того как второй процесс начнет выполнять

Действие 2, управление не перейдет к первому процессу. Как следствие, результат будет получен по второму алгоритму;

- Первый процесс объявляется *более приоритетным* по отношению ко второму. С точностью до порядка инициации процессов всё симметрично предыдущему варианту, и можно ожидать, что результат будет получен за счет Действия 1. Однако если в этом случае главная программа не будет более приоритетной по отношению к порождаемым ею процессам, то окажется ошибочным уничтожение второго процесса (вызов его деструктора), так как до его порождения дело не дойдет;
- Оба процесса объявляются с одинаковым приоритетом и каждому из них задается квант времени для исполнения, по истечении которого процесс блокируется и управление передается другому процессу. Такая имитация параллельного выполнения потоков может дать эффект повышения эффективности получения результата, если скорости вычислений Действий 1 и 2 разнятся более, чем в два раза плюс накладные расходы на переключение процессов.

Вариантов системной поддержки потоковых вычислений довольно много. Это не удивительно, поскольку разработчики обычно стремятся к сочетанию наглядности и удобства предлагаемых средств с обеспечением максимально полного учета возможностей оборудования. Также понятна тенденция к стандартизации потоковых вычислений, которая способствовала бы технологичности их реализации, к переносимости потоковых систем на различные архитектурные решения оборудования. Сегодня наиболее известным стандартом является POSIX Threads (англ.– переносимый интерфейс операционных систем Unix), который является одним из стандартов, разработанных в операционной системе Unix. (POSIX – аббревиатура **p**ortable **o**perating **s**ystem **i**nterface for Uni**x**). Стандарт POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995) определяет API для управления потоками, их синхронизации и планирования. Его реализация имеется для большинства UNIX-подобных систем (GNU/Linux, Solaris, FreeBSD, OpenBSD, NetBSD, OS X), а также для Microsoft Windows и других ОС. Друглй пример, начиная с выхода в 2011 году нового стандарта языка C++ реализация потоков была закреплена на уровне стандартной библиотеки.

13.5. Событийный механизм и параллелизм

Механизм событий – специальный самостоятельный стиль программирования, используемый, когда алгоритм в программе естественно описывать как конечный автомат. Характерной особенностью стиля является представление программы в виде комплекта фрагментов, каждый из которых активизируется, когда в ходе вычислений возникает событие, для обработки которого этот фрагмент предназначается. Понятие *события*, восходящее к терминологии операционных систем, где оно ассоциируется с аппаратными прерываниями, сегодня трактуется максимально широко: это произвольное действие, в результате которого становится истинным некоторое условие, разрешающее или даже требующее выполнить определенные фрагменты программы, связанные с этим событием. Такие условия называют *спусковыми функциями события*, а фрагменты, связанные с ним, – *обработчиками события*. В результате возникновения события разрешенным для выполнения обработчикам обычно передается информация – *сообщение*.

Событийная программа обычно строится как набор обработчиков со своими спусковыми функциями. Может быть определена программа-генератор событий, которую естественно рассматривать как распределитель вычислений по обработчикам, завершающим содержательные действия, возвратом управления программе-генератору. Эта программа может сама ожидать события как от обработчиков, так из иных источников. Таким образом достигается содержательная структура событий, которая задает регламент обработки данных.

Проиллюстрируем событийный механизм на примере цепочки действий, которая складывается в результате нажатия клавиши пользователем, работающим с текстовым редактором:

- *Событие прерывания* в связи с нажатием клавиши. Обработчик события – программа, которая зафиксирует сигнал от клавиатуры. Продолжение выполнения прерываемой программы откладывается до того момента, когда сигнал о том, какая клавиша нажата, запомнена. Прерываемая программа – это совсем необязательно текстовый редактор, который, скорее всего, находится в состоянии ожидания символа или команды.

Процессор, которому поручена обработка события нажатия клавиши, может заниматься другой работой, требующей возобновления после выполнения реакции на прерывание. Обработка события прерывания закачивается генерацией следующего события;

- *Событие для драйвера клавиатуры*. Обработчик формирует коды нажатых клавиш (ранее нажатых и новой) в качестве сообщения для еще одного события. Чаще всего именно он сигнализирует об окончании выполнения реакции на прерывание;
- *Событие о готовности символа* для использования. Обработчики события «знают» семантику служебных и функциональных клавиш, возможно, историю использования ранее подготовленных символов и другую информацию, которая нужна для обработчика. В условиях работы пользователя с текстовым редактором такими обработчиками являются его специальные подпрограммы, которые могут выполнить команды редактора (перемещение курсора, выделение, удаление и пр.) и/или изображения символа на экране, вставки его в файл и др. Возможны обработчики, непосредственно не связанные с редактированием, например, для подсчета всех символов текста. При соблюдении корректности параллельных вычислений некоторым обработчикам могут быть предоставлены свои автономные процессоры. Подпрограммы-обработчики символа обеспечивают возможность генерации семантической информации для выполнения сопутствующих действий, в частности, другими обработчиками;
- *Специальные события о готовности информации* для использования. Обработчики таких событий выполняют функции, весьма отдаленно связанные с нажатием клавиши, которое опосредованно привело к формированию нужных данных. В качестве примера такого события уместно указать формирование слова, одной из реакций на которое будет активизация проверки орфографии.

Использование событийного управления по своей сути является регламентированным применением механизма send-receive, в котором отправитель не заботится о том, какие обработчики активизи-

руются при возникновении события, как они распорядятся сообщением. При необходимости для отравителя можно предусмотреть извещения от обработчиков о получении сообщения. Все назначается отправителем, а определяется готовностью получить сообщение получателем-обработчиком. Когда оказываются истинными спусковые функции нескольких обработчиков, порядок их активизации обычно определяется как совместное выполнение. При использовании многопроцессорных систем и распределении обработчиков по разным процессорам это позволяет выполнять реакцию на событие реально параллельно. Разумеется, что в любом случае параллелизма необходимо обеспечить синхронизацию выполнения обработчиков. Стратегия, которой удобно придерживаться в синхронизации событийного механизма, подобна синхронизации потоковых вычислений. Программа-генератор событий следит за ходом выполнения обработчиков, которые посылают ей сигналы о состоянии процессов. На основе этой информации генерируют события, оповещающие соответствующие обработчики, которые в состоянии влиять на выполняющиеся процессы. Применительно к нашему примеру о конкурирующих алгоритмах это означает необходимость отслеживания сигналов о завершении Действий 1 и 2 и генерации события-оповещения специального обработчика о том, что он должен обеспечить уничтожение потока с альтернативным Действием. С точностью до генерации событий все очень похоже на решение с использованием механизма send-receive.

13.6. Синхронизация параллельных вычислений

Предыдущий анализ потокового решения задачи о конкурирующих альтернативных алгоритмах показывает, что даже при использовании однопроцессорного вычислителя возможно повышение производительности за счет параллелизма. Заметим, что это давно учитывается при разработке вычислителей и мотивирует модификацию канонической модели вычислений фон Неймана, направленную на использование параллелизма на уровне оборудования. Так, процессор работает параллельно с доставкой аргументов команд, эффективность вычислений повышается за счет быстрых регистров. Список подобных модернизаций, в разных аспектах связанных с внутренним параллелизмом вычислений, можно продолжить. Важно отметить, что, хотя такой параллелизм и другие

технические способы повышения производительности вычислений (например, кэширование) реально используются, на языки и методы программирования они влияют незначительно. Эффект от их внедрения как в однопроцессорные, так и в многопроцессорные вычислительные системы достигается преимущественно за счет компиляторной оптимизации. И это одна из причин ограниченного влияния возможностей параллелизма на языки программирования, и, как следствие, возможности развития параллельного образа мышления программиста ограничиваются.

Наш пример конкурирующих алгоритмов по своей сути демонстрирует параллельный образ мышления: предложенное решение явно рассчитано на два активно используемых процессора. В этом смысле задачи, в которых параллельность в таких локальных задачах, как, например, сложение двух векторов, является лишь подспорьем, сделать это можно компактно и быстрее, чем в стандартном цикле

```
for (int i = 0; i <N; i++) { A[i] = B[i] + C[i] }    (10)
```

Такого рода задачи не связаны с необходимостью думать о параллельных алгоритмах. Такие решения, особенно в случаях, когда программисту предлагаются встроенные в язык функции (intrinsic), позволяют рассматривать алгоритм в виде локальной для использующей программы свертки. Вместе с тем реализация подобных сверток уже требует знания и умения разрабатывать параллельные программы. Так, для параллельной реализации цикла (10) нужно иметь в виду следующее:

- *Баланс загрузки процессоров*. Если в динамике выполнения программы число процессоров, которое может предоставить вычислительная среда, равно `N`, то для каждого `i` оператору `A[i]=B[i]+C[i]` разумно предоставить свой процессор. Баланс достигается: все `N` процессоров будут одновременно работать над своими задачами и закончат эту работу примерно одновременно. Когда число свободных процессоров меньше `N`, баланс их загрузки может быть достигнут, если это число кратно `N`. В этом случае процессоры будут назначены для выполнения равных групп операторов. Сбалансированность загрузки может сохраниться, если есть возможность незадействованные процессоры

передать другим фрагментам программы для параллельного вычисления.

- *Синхронизация завершения*. Даже одинаковые группы действий, назначенные одинаковым процессорам, совсем не обязательно будут выполняться одно и то же время, а если продолжение вычислений требует готовности всех данных, которые продуцируются параллельно выполняемыми процессами, необходимо приостановить те процессы, которые готовы к продолжению вычислений, до тех пор, пока отстающие процессы не придут к такому же состоянию. Применительно к нашему примеру это означает ожидание окончания вычисления всех `A[i]` и только после этого продолжить каждый из задействованных, но не завершенных процессов.
- *Учет масштабируемости процессорных ресурсов*. Программа, реализующая сложение двух векторов, должна быть рассчитана на все случаи применения от использования только одного процессора до отсутствия дефицита процессорного ресурса, включая все промежуточные возможности. Дополнительным требованием к алгоритму здесь является балансировка загрузки процессоров для всех указанных случаев в сочетании с минимизацией накладных расходов на распределение процессорного ресурса.

Подобные задачи всегда возникают при разработке параллельных программ. Для типовых и наиболее востребованных ситуаций существует ряд стандартных решений, с которыми имеет смысл познакомиться заранее.

13.7. Взаимодействие поставщиков и потребителей данных

Рассмотрим проблему так называемой *гонки данных* (data race), которая возникает, когда один процесс – *поставщик данных* – готовит данные для другого – *потребителя данных*, причем оба процесса выполняются параллельно. Проблемная ситуация заключается в следующем. Если не позаботиться о том, чтобы потребитель данных получал их только после того, как поставщик их произведет, вычисления программы могут оказаться некорректными.

В Википедии в статье «Состояние гонки» проблема и возможные ее решения с использованием средств языка Java демонстрируются на следующем примере. Имеется целочисленная переменная `x`,

доступная для двух потоков. Поток 1 реализует цикл, каждая итерация которого увеличивает значение x на 1:

```
// Поток 1:
while (!stop)
{
  x++;
   ...
}
```

Поток 2 потребляет значения x, т. е. в цикле печатает x, с фильтрацией, т. е. пропуская нечетные значения:

```
// Поток 2:
while (!stop)
{
   if (x%2 ==0)
      System.out.println("x="+ x);
   ...
}
```

Если потоки выполняются параллельно, а именно это позволяет говорить об осмысленности примера, то фильтрация срабатывает необязательно. Пусть для определенности первоначально x=0. Тогда при выполнении последовательности действий:

1. Оператор **if** в Потоке 2. Он проверяет x на четность.
2. Оператор «x++» в Потоке 1. Он увеличивает x на единицу.
3. Оператор вывода в Потоке 2. Это приведет к выводу «x=1» хотя, казалось бы, переменная проверена на четность.

Далее обсуждаются варианты преодоления проблемы.

13.7.1. Локальная копия

Для исправления ошибки можно попытаться воспользоваться копированием x в локальную переменную Потока 2 (программа Потока 1 не меняется):

```
// Поток 2:
while (!stop)
{
  Int cached_x = x;
  if (cached_x%2 == 0)
      System.out.println("x="+cached_x);
  ...
}
```

Для корректности решения необходимо, чтобы переменная `x` была описана как волатильная (изменяющаяся):

```
volatile int x;
```

в результате чего гарантируется, что связь между потоками происходит, когда один поток присваивает значение волатильной переменной, другой поток осведомляется об этом. Таким образом, Поток 2 имеет возможность контролировать изменение переменной `x`.

Этот способ работает только тогда, когда для обработки передается только одна переменная и копирование производится за одну машинную команду (непрерываемую по определению).

13.7.2. Синхронизация

Для более надежного способа решения можно воспользоваться конструкцией **synchronized**, позволяющей задавать фрагменты разных потоков, которые не только знают друг друга, но и выполняются в определенной последовательности:

```
// Поток 1:
while (!stop)
{
synchronized(SomeObject) // Начало блока синхронизации.
  {                      // Он выполняется,если другой блок
                         // с именем SomeObject не начал
    x++;                 // выполняться раньше. Выполнение
                         // данного блока запрещает выполняться
  }                      // другому блоку с именем SomeObject.
                         // Завершение текущего блока
...                      // снимает запрет на выполнение
                         // другого блока, тем самым он
}                        // начинает выполняться.

// Поток 2:
while (!stop)
{
synchronized(SomeObject)// Все выполняется точно так же,
  {                      // как для блока в Потоке 1
   if (x%2 ==0)
   System.out.println("x="+ x);
  }
...
}
```

В результате выполнения этого варианта решения блоки `SomeObject` в Потоках 1 и 2 выполняются по очереди, что и требуется для данной задачи.

13.7.3. Комбинированный способ

К сожалению, в случае когда в Потоке 2 вместо `System.out.println` требуется более сложная обработка, а Поток 1 передает для обработки более одной переменной, блок синхронизации оказывается слишком большим, что приведет к дисбалансу использования процессорных ресурсов. Способ локальной копии не подходит, поскольку имеется требование непрерывности копирования (реализация его за одну машинную команду для двух переменных).

Предлагаемое ниже решение связано с комбинированием двух рассмотренных способов: «опасные» переменные копируются в блоках синхронизации. С одной стороны, это снимет ограничение на использование одной машинной команды, с другой – позволяет избавиться от слишком больших блоков синхронизации.

```
volatile int x1, x2;
...
// Поток 1:
while (!stop)
{
  synchronized(SomeObject)
  {
    x1++;
    x2++;
  }
  ...
}

// Поток 2:
while (!stop)
{
  long cached_x1, cached_x2;
  synchronized(SomeObject)
  {
    cached_x1 = x1;
    cached_x2 = x2;
  }
  if ((cached_x1 + cached_x2)%100 ==0)
    DoSomethingComplicated(cached_x1, cached_x2);
  ...
}
```

Как видно из обсуждения решений, исключение гонки данных является довольно трудной задачей. Очевидных способов выявления и исправления состояний гонки не существует. Лучший способ избавиться от гонок – проектирование взаимодействия поставщиков и потребителей данных без прямого незащищенного использования общей памяти. В то же время состояния гонки могут оказаться полезными. Так, потребитель может знать, какие данные могут вырабатывать поставщики, каждый из которых наряду со своей работой формирует значение, отражающее, как закончилась эта работа. Тот факт, что используемая потребителем переменная получила значение, отличное от первоначального, указывает, что работа одного или нескольких процессов выполнена. Если поставщики выполняли одну и ту же работу, это означает что результат работы получен, и в случае, когда безразлично, какой из поставщиков победил в соревновании, можно принудительно завершить все потоки. Легко видеть, что этот случай является решением задачи конкурирующих альтернативных алгоритмов, которую мы обсуждали в разделе 0.

13.8. Мониторы

В языках программирования предлагаются различные средства повышения уровня поддержки взаимодействия и синхронизации процессов, обеспечивающих доступ к неразделяемым ресурсам. Одним из таких средств являются *мониторы*, которые обеспечивают взаимоисключение обращения процессов к общим ресурсам. В отличие от семафоров, которые требуют расстановки синхронизирующих примитивов down и up по разным фрагментам программы, мониторы позволяют решать задачу взаимоисключения. Идея мониторов была предложена Хоаром, а первая реализация выполнена Хансеном. Хоар разработал теоретическую основу и показал эквивалентность мониторов механизму семафоров. В языке программирования впервые механизм мониторов представлен в Concurrent Pascal. Для структурирования межпроцессного взаимодействия он был использован в операционной системе Solo.

Монитор – это программный модуль, состоящий из *набора конкурирующих процедур*, которые имеют возможность обращаться к общим ресурсам *информационных структур*, которые представляют этот ресурс для активного использования только одной из процедур набора, *мьютекса*, обеспечивающего взаимоисключение

выполнения процедур набора. Условие, выполнение которого позволяет избежать состояния гонки данных, называемое *инвариантом* монитора, часто рассматривается в качестве составной части монитора. Обычно инвариант явно не выражается в коде, но, к примеру, в языках Eiffel и D проверка инварианта и блокировка выполнения, организуемая компилятором, предусматриваются. Это делает мониторы более удобными, так как не требует от программиста ручного добавления операций блокировки и разблокировки. Но за такую повышенную безопасность приходится платить избыточностью кода, когда истинность инварианта проверять не требуется.

Выполняемая процедура из набора монитора захватывает мьютекс перед началом работы и держит его до своего завершения. Если мьютекс оказался уже захваченным, то она ожидает момент, когда инвариантное условие становится истинным.

Следующий пример иллюстрирует использование монитора для поддержки корректности выполнения транзакций банковского счета. Здесь инвариантом является условие, что баланс (переменная `balance`) должен отразить все прошедшие операции до того, как начнется новая операция. Иными словами, операции `withdraw` (изъятие средств) и `deposit` (пополнение счета), изменяющие `balance`, упорядочиваются так, чтобы выполнение очередной из них не начиналось до завершения предыдущей. Это достигается захватом мьютекса в начале операции, ожиданием в очереди новых операций до тех пор, пока выполняемая операция не завершится и не освободит мьютекс.

В данном примере явное представление инварианта не требуется.

```
monitor account {
  int balance := 0
  function withdraw(int amount) {
   if amount < 0 then error "Счет не может быть отрицательным"
   else if balance < amount then error "Недостаток средств"
   else balance := balance - amount
  }
  function deposit(int amount) {
   if amount < 0 then error "Счет не может быть отрицательным"
   else balance := balance + amount
  }
}
```

Чтобы избегать состояния активного ожидания, процессы должны сигнализировать друг другу об ожидаемых событиях. Мониторы обеспечивают эту возможность с помощью так называемых *условных переменных*, которые отражают выполнение/невыполнение условия, разрешающего/запрещающего процедуре монитора для выполнения дальнейшей работы. Во время ожидания процедура временно отпускает мьютекс и выбывает из списка исполняющихся процессов. Любой процесс, который в дальнейшем приводит к выполнению этого условия, использует условную переменную для оповещения ждущего ее процесса. Оповещенный процесс захватывает мьютекс обратно и может продолжать.

Следующий пример использует условные переменные для реализации канала между процессами, который может хранить одномоментно только одно целочисленное значение.

```
monitor channel {
int contents
boolean full := false
condition snd              //условная переменная,
condition rcv

function send(int message) {
 while full do wait(rcv)   //Ожидание освобождения канала
 contents := message
 full := true
 notify(snd)               //Оповещение, что сообщение передано
 }

function receive() {
 var int received
 while not full do wait(snd) //Ожидание освобождения канала
 received := contents
 full := false
 notify(rcv)               // Оповещение, что сообщение получено
 return received
 }
}
```

Независимо от того как используются операции send и receive в конкретных процессах, представленный монитор обеспечивает корректный обмен сообщениями через канал, связывающий эти процессы.

13.9. Специализированные параллельные вычислительные системы

Анализ различных алгоритмов, допускающих параллельную реализацию, показывает, что очень часто они включают типовые шаблоны, которые естественно реализовывать на аппаратном уровне и с использованием параллелизма. Это привело к постановке задачи создания особых моделей вычислений и, соответственно, специализированных параллельных вычислительных систем. Ниже сначала даются примеры такого рода, а затем обсуждаются архитектуры вычислительных систем, которые в состоянии обеспечивать параллелизм.

Исторически первый пример специализированных вычислений, для которых наработаны шаблоны оперирования, – обработка информации, представленной векторами и матрицами. Потребность максимально эффективно реализовывать оперирование с этими типами данных, в частности, при решении задач моделирования физических процессов, осознана довольно давно. Осознание ее привело к разработке специализированных *векторных* и *матричных вычислительных машин*. Их характерной особенностью является наличие команд, которые позволяют складывать, вычитать, умножать на скаляр и др. одновременно для всех компонент массивов. В реализации это выражается как наличие большого числа процессоров, способных выполнять одну и ту же команду с разными, стандартизованно представленными в памяти данными.

Шаблоны более высокого уровня, первоначально предназначавшиеся для оперирования графическими объектами с целью получения реалистичных картин на экране, привели к разработке так называемых *графических процессоров* (GPU – graphics processing unit), на аппаратном уровне реализующих полный комплект типичных команд для динамического вывода данных в виде экранных изображений. Оказалось, что эти шаблоны можно эффективно применять и для решения вычислительных задач. При желании это можно рассматривать как развитие матричных вычислений. Однако такое рассмотрение не совсем верно – помимо этого для вычислений используются возможности параллельного вывода на экран фрагментов картин, развитое оперирование палитрой цветов и другие средства оперирования GPU. Это позволяет привлекать к вычислениям как давно известные методы, например, в начертательной геомет-

рии, так и новые подходы. В результате появляется новая модель вычислений с очень высокой степенью поддержки параллелизма.

Еще одно направление специализации вычислений с учетом особенностей решаемых задач, в частности, для создания условий для организации параллельных вычислений, связано с так называемыми *программируемыми логическими интегральными схемами* (ПЛИС, англоязычный вариант: FPGA – Field Programmable Gate Array). Это комплексы поддержки разработки микросхем, которые позволяют конфигурировать логические блоки, реализующие требуемую функцию, программировать электронные связи между блоками, организовывать связь внешнего ввода/вывода микросхемы с внутренней логикой ее функционирования (программирование блоков ввода/вывода). Данное направление реализует весьма широкий круг методов обработки информации, которые отвечают различным моделям вычислений, связанных со спецификой приложений.

Эти подходы сочетаются с инженерными решениями, которые способны, по мнению разработчиков, увеличить эффективность вычислений. Часто такие решения предлагаются лишь как технические усовершенствования базовой модели, но впоследствии осознается возможность программистов управлять вычислениями с использованием нового средства. В качестве примера такого рода может служить изобретение кэша как компонента оборудования, сокращающего время доступа к общей памяти за счет дублирования на быстрые регистры содержимого некоторых ячеек. Первоначально кэши команд и данных не допускали управления программистом – можно было использовать только знание стратегии кэширования, подбирая программный код в соответствии с ней. Но очень скоро появились специальные команды, влияющие на кэширование. И хотя известно, что кэширование противоречит многопроцессорности, умножая проблемы синхронизации параллельных процессов, появляются архитектуры, которые этот факт игнорируют.

13.10. Классификация архитектур вычислительных систем

Множество архитектур систем, предлагаемых потребителям вычислительных услуг, достаточно разнообразно. Вместе с тем для каждой из них можно указать на базовые решения, которые определяют реализуемую модель вычислений в общих чертах и указывают,

для каких типов вычислений целесообразно использовать то или иное оборудование. Исходя из возможных базовых решений М. Флинт еще в 1966 году предложил классификацию архитектур, которая в 1972 году была дополнена и в настоящее время рассматривается как общепризнанная. Основой классификации служат понятия *потоков* (stream), или *пулов* (pool) *команд* и *данных*, говоря о которых имеют в виду, соответственно, источники двух последовательностей: выполняемых процессором команд и элементов перерабатываемых данных. Флит выделяет четыре класса архитектур SISD, MISD, SIMD и MIMD, которые характеризуются вариантами сочетания одиночного и множественного источников команд и данных.

13.10.1. SISD (single instruction stream over a single data stream)

В русскоязычной литературе используется также обозначение ОКОД – одиночный поток команд и одиночный поток данных.

Это вычислительные системы, в которых как для команд, так и для данных имеются только единственные источники (см. рис. 13.1). В таких системах все команды исполняются последовательно, каждая команда выбирается из потока Instruction Pool. Она выполняет одну операцию с данными, заданными ее аргументами из потока Data Pool.

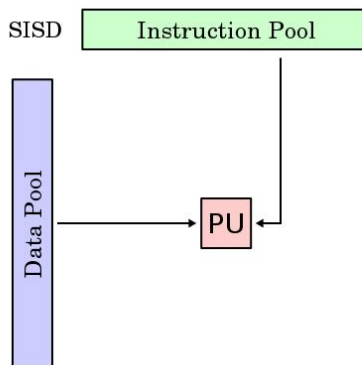


Рис. 13.1. SISD-архитектура

В данном классе не используется параллелизм ни данных, ни инструкций, т. е. SISD-машина не является параллельной. Это машины с классической базовой моделью вычислений фон-неймановского

типа. Известная модернизация их (ввод/вывод, кэширование и др.) допускается, но ограничивается автономными механизмами реализации.

К этому классу помимо фон-неймановских машин обычно относят системы, использующие процессоры с внутренним параллелизмом:

- *Конвейерные процессоры*, которые при выполнении команды предусматривают разделение ее обработки на последовательность независимых стадий с сохранением результатов в конце каждой стадии. Это позволяет выполнять одновременно несколько последовательных команд, начиная следующую команду сразу после передачи результата стадии текущей команды ее следующей стадии;
- *Суперскалярные* процессоры, которые имеют возможность загружать для исполнения сразу несколько аппаратных блоков выполнения. Решение о запуске команды или ее части на исполнение принимает сам вычислительный модуль на основе анализа зависимостей по данным;
- *VLIW-процессоры* (**v**ery **l**ong **i**nstruction **w**ord – очень длинная машинная команда). Это архитектура процессоров с несколькими вычислительными устройствами, характеризующаяся тем, что одна команда процессора содержит несколько операций, которые должны выполняться параллельно. Фактически это «видимое программисту» микропрограммное управление, когда машинный код представляет собой лишь немного свернутый микрокод для непосредственного управления аппаратурой. Длина команды может достигать 128 или даже 256 бит;

13.10.2. SIMD (single instruction stream over a multiple data stream)

Эта архитектура (русскоязычная аббревиатура **ОКМД** – **о**диночный поток **к**оманд и **м**ножественный поток **д**анных) предусматривает для одной команды одновременное выполнение операции с данными, поступающими из нескольких источников (см. рис. 13.2). В частности, векторные и матричные вычислители выполняют арифметическую операцию над аргументами, в качестве которых используются пары элементов, извлекаемых из массивов. В SIMD-машинах один процессор загружает одну инструкцию, набор дан-

ных к ним и выполняет операцию, описанную в этой инструкции, над всем набором данных одновременно. Способ выполнения векторных и матричных операций не фиксируется. Это может быть процессорная матрица или конвейерная обработка (см. п. 6.1).

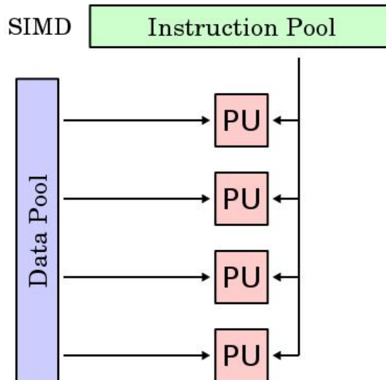


Рис. 13.2. SIMD-архитектура

Класс SIMD подразделяется на два подкласса, различающихся по отношению к доступу к памяти:

- **SM-SIMD** (**s**hared **m**emory **SIMD**). Это так называемые *векторные процессоры*, выполняющие одну команду сразу со всеми элементами массива (или подходящими частями массивов) независимо друг от друга. С середины 70-х годов векторные процессоры используются в качестве базы суперкомпьютеров. Однако в реальных задачах требуется не только высокий уровень параллелизма на массовых операциях, но и скалярное оперирование, которое не вписывается в структуру базовой модели вычислений. По этой причине в SM-SIMD-машинах предусматриваются различные модификации модели, позволяющие преодолевать проблемы несбалансированности загрузки оборудования;
- **DM-SIMD** (**d**istributed **m**emory **SIMD**). К этому подклассу относятся матричные процессоры, которые представляют собой массив процессоров, контролируемых одним управляющим процессором, выполняя по его команде одну операцию над своей собственной порцией данных, хранящихся в локальной памяти.

Так как обмена данными между процессорами нет, не требуется никакой синхронизации, что позволяет достигать огромных скоростей вычислений и с легкостью расширять систему, просто увеличивая количество процессоров. Так как матричные процессоры можно использовать только на ограниченном круге задач, до появления потребности реализации реалистичных изображений DM-SIMD-машины не получали широкого распространения. Положение изменилось в связи ростом потребности разработки развитых средств машинной графики. Сегодня эта модель вычислений является базовой в графических процессорах.

13.10.3. MISD (multiple instruction stream over a single data stream)

Эта архитектура (русскоязычная аббревиатура **МКОД** – **м**ножественный поток **к**оманд и **о**диночный поток **д**анных), предполагающая обработку одного потока данных с помощью нескольких процессоров (см. рис. 13.3), не нашла свое воплощение в реальных вычислительных машинах. Иногда ее рассматривают как вариант конвейерного вычислителя. Это не совсем корректно, поскольку конвейерные процессоры имеют дело с последовательными стадиями одной команды (см. выше), а не работу автономных процессоров со своими программами обработки данных.

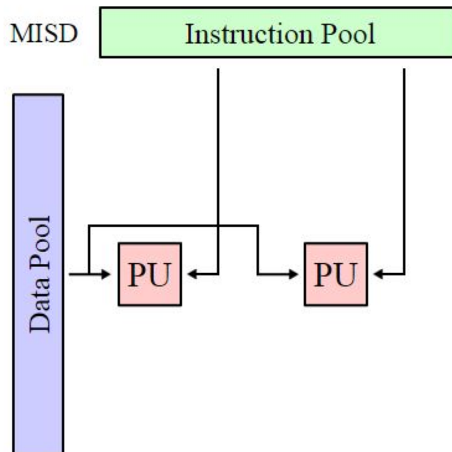


Рис. 13.3. MISD-архитектура

13.10.4. MIMD (multiple instruction stream over a multiple data stream)

Класс **MIMD** (русскоязычная аббревиатура **МКМД** – **м**ножественный поток **к**оманд и **м**ножественный поток данных) можно охарактеризовать как вычислительные системы с несколькими устройствами обработки команд, объединенных в единый комплекс и работающих каждое из них со своим потоком команд и данных (см. рис. 13.4). Он включает многопроцессорные системы, где процессоры обрабатывают множественные потоки данных. Сюда принято относить традиционные мультипроцессорные машины, многоядерные и многопоточные процессоры, а также компьютерные кластеры. Вариантов организации таких комплексов довольно много, и это один из недостатков классификации Флинта.

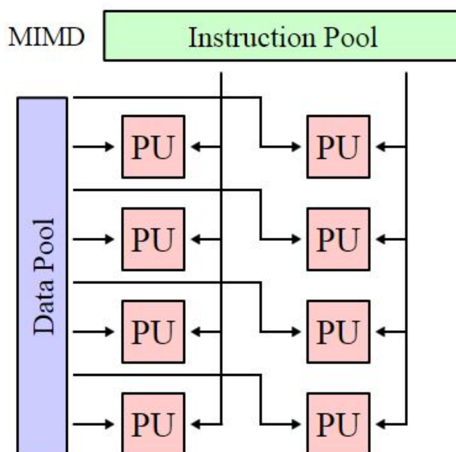


Рис. 13.4. MIMD-архитектура

Как и SIMD, MIMD-класс подразделяется на два подкласса, различающихся по отношению к доступу к памяти:

- **SM-MIMD** (**s**hared **m**emory **MIMD**). В эту группу попадают многопроцессорные машины с общей памятью, многоядерные процессоры с общей памятью. Классический и самый распространенный пример – симметричное мультипроцессирование (**SMP** – **S**ymmetric **M**ulti**p**rocessing), при котором два или более одинаковых процессора сравнимой производительности подключаются

к общей памяти и периферийным устройствам единообразно и выполняют одни и те же функции (именно это и называется симметричностью). В таких машинах память каждому процессору видна как общее адресное пространство, процессоры обмениваются данными по общей адресной шине через общие переменные (shared variables).

Для каждого процессора доступ к любому участку памяти является одинаковым, что обычно достигается реализацией SMP посредством так называемой **UMA**-архитектуры (**UMA – u**niform **m**emory **a**ccess, однородный доступ к памяти). Все процессоры в UMA-архитектуре обращаются к общей памяти одновременно, и время запроса к данным не зависит ни от того, какой именно процессор обращается к памяти, ни от того, какой именно чип памяти содержит нужные данные. Вместе с тем каждый процессор может использовать свой собственный кэш.

Достоинством SM-MIMD-систем является относительная простота программирования (поддержка SMP существует уже давно во всех ведущих операционных системах).

Недостаток таких систем – невысокая *масштабируемость*: чем больше процессоров в системе, тем выше становится нагрузка на общую шину.

Память, предоставляемая в распоряжение программисту, как одно общее адресное пространство, может быть физически распределена по узлам системы. В этом случае говорят об особом подклассе систем: **DSM-MIMD** (**d**istributed **s**hared **m**emory **MIMD**). В этом подклассе у каждого процессора имеется своя локальная память, а к другим участкам памяти процессор обращается через высокоскоростное соединение. Так как доступ к разным участкам общей памяти является неодинаковым (к своей памяти быстрее, к другой – медленнее), то такие системы носят название **NUMA** (**n**on-**u**niform **m**emory **a**ccess). Физическая распределенность памяти приводит тому, что каждый процессор не видит в памяти изменения, сделанные другими процессорами. Для решения этой проблемы предлагаются различные способы: через согласование кэша – ccNUMA, без согласования кэша – nccNUMA.

По сравнению с UMA-архитектурами NUMA-системы имеют более высокую масштабируемость, позволяя создавать массово-

параллельные вычислительные системы, где число процессоров достигает нескольких тысяч. Модель программирования в таких системах остается прежней – потоки исполнения обмениваются друг с другом данными через общие переменные.

- **DM-MIMD** (**d**istributed **m**emory **MIMD**). В этот подкласс попадают многопроцессорные MIMD-машины с распределенной памятью. У каждого процессора имеется своя собственная локальная память, которая не видна другим процессорам. Каждый процессор в такой системе выполняет свою задачу со своим набором данных в своей локальной памяти. Если процессору нужны данные из памяти другого процессора, данный процессор обменивается с другим процессором сообщениями (см. раздел 3).

  Главное преимущество DM-MIMD-машин – их высокая масштабируемость, которая позволяет создавать массово-параллельные системы из несколько сотен тысяч процессоров.

13.10.5. Поддержка распределенной и разделяемой памяти

Программирование параллельных программ всегда должно соответствовать модели вычислений, определяемой архитектурой оборудования, которое используется в расчетах. Предложенная Флинтом и усовершенствованная в дальнейшем классификация позволяет выбирать метод программирования лишь отчасти – более правильно исходить из особенностей решаемой задачи и уже после определять, как принимаемый метод отображать на конкретную среду вычислений. В этом отношении важно с самого начала понять, может ли задача ограничиваться при взаимодействии процессов эпизодическими обменами информацией, и тогда для ее решения достаточно использования вычислительной системы с процессорами с автономным адресным пространством, т. е. архитектуры с *распределенной памятью* (distributed memory). Это DM-SIMD и DM-MIMD вычислительные системы. Если же для задачи нужен совместный доступ параллельно работающих процессов к общей *разделяемой* между процессорами памяти (shared memory), то требуются вычислительные системы классов SM-SIMD и SM-MIMD.

Использование SM-SIMD-архитектур и, соответственно, общей разделяемой памяти процессоров в первую очередь имеет цель достижение высокого уровня параллелизма на массовых операциях.

Эта ориентация архитектуры накладывает ограничение возможности обработки различных данных, связанных с жесткой последовательностью выполняемых процессорами команд. Для таких случаев более подходящими являются SM-MIMD-архитектуры в сочетании со специальным методом программирования, получившим название **SPMD** (**s**ingle **p**rogram over **m**ultiple **d**ata – единая программа для обработки множества данных), который обеспечивает высокоуровневый параллелизм. В SPMD множество автономных процессоров одновременно выполняют одну и ту же программу, но не обязательно одну и ту же ее операцию. SPMD-метод характеризуется тем, что процессы задачи разделены и работают одновременно на нескольких процессорах с различными входными данными с целью более быстрого получения результатов. Метод может рассматриваться как специализированный стиль программирования, благодаря которому процессы могут выполняться на процессорах общего назначения. Вместе с тем для управления потоками данных для SIMD вполне разумно использование векторных процессоров. Понятно, что эти методы не являются взаимоисключающими.

Стиль SPMD-программирования не обязательно связывать лишь с обработкой, основанной на разделяемой памяти. Вполне адекватно в этом стиле реализуется и оперирование, организующее вычисления с распределенной памятью, т. е. в тех архитектурах, в которых используется набор независимых компьютеров, называемых «узлами». Каждый «узел» запускает свою собственную программу, которая является его частью разрабатываемой программной системы, и взаимодействует с другими узлами посредством отправки и получения сообщений. Барьерная синхронизация, т. е. ожидание окончания каких-либо процессов в узлах также может быть реализована путем сообщений. В настоящее время программист избавлен от знания деталей передачи сообщений (которые мы обсуждали в разделе 4) за счет использования стандартных интерфейсов, таких как **PVM** (**p**arallel **v**irtual **m**achine – параллельная виртуальная машина) и **MPI** (**m**essage **p**assing **i**nterface – интерфейс передачи сообщений).

На машинах с общей памятью с несколькими процессорами, которые обращаются к общему адресному пространству, механизм сообщений может быть организован путем передачи на хранение их содержания в общую область памяти. Такой метод часто является

наиболее эффективным для машин общей памяти с большим числом процессоров, особенно для машин с неравномерным доступом к памяти (NUMA архитектура), где память является локальной для процессора и доступ к памяти другого процессора занимает больше времени. Поддержка SPMD в случае с разделяемой памятью, как правило, реализуется стандартными процессами ОС.

Многопроцессорная система с симметричной разделяемой памятью (SMP) дает возможность распараллеливания исполнения за счет независимого пути исполнения приложения на каждом процессоре, с использованием общей памяти как механизма обмена данных. Программа начинает выполняться на одном процессоре и выполнение делится на так называемые *параллельные регионы*, которые определяются в коде с помощью параллельных директив (объявление начала параллельного региона в коде программы). В параллельном регионе процессоры выполняют одну программу (фрагмент главной программы) с разными данными. Типичным примером является параллельный цикл DO, где различные процессоры работают на отдельных частях массива, обрабатываемых в цикле. В конце цикла выполняется синхронизация, при этом только один процессор продолжает работу, а остальные ждут. В настоящее время стандартным интерфейсом для многопроцессорной системы с разделяемой памятью является **OpenMP** (**o**pen **m**ulti-**p**rocessing). Он обычно реализуется с помощью потоковых методов.

13.10.6. Квазипараллельные системы

Параллельное программирование всегда приводит к недетерминированному выполнению программ. Это связано с тем, что в общем случае нельзя точно знать время, которое потребуется для выполнения процессов, и порядок, в котором они будут выполняться. Поэтому приходится вводить в программу специальные действия, обеспечивающие синхронизацию, непосредственно с логикой программы не связанные. Недетерминизм не обязательно означает, что разные запуски на выполнение параллельной программы с одними и теми же данными приводят к различным результатам. Если для программы такое возможно, то, как правило, она считается некорректной. К сожалению, распознать некорректность параллельной программы намного сложнее, чем для последовательной программы, и именно потому, что из-за недетерминизма ход

параллельных вычислений обычно предсказать невозможно. И эта одна из проблем, связанных с параллелизмом, в частности, с отладкой. Другая проблема – понимание трудностей параллельного программирования, особенно теми, кто сталкивается с недетерминизмом впервые.

В обучении очень важно, чтобы у обучающегося была возможность повторять ситуации, возникающие при выполнении программы. Иными словами, требуется не параллельное выполнение, а его имитация с детерминированным поведением. При хорошо организованной имитации ею можно управлять: приостанавливать вычисления, выбирать тот или иной вариант выполнения программы, повторять расчеты и др. Тогда обучающийся оказывается в состоянии увидеть эффекты, которые могут проявиться при реальном параллельном счете, и соответственно, корректировать принимаемые решения. Эти же средства управляемой имитации полезны и при отладке параллельной программы, в частности, для выявления критических ситуаций с вероятным нарушением синхронизации, гонки данных и других случаев некорректного поведения программы.

Во многих случаях при разработке параллельной программы ее потребность в параллелизме превосходит доступные процессорные ресурсы. Эта ситуация характерна, в частности, для задач моделирования развивающихся во времени взаимодействующих процессов, которые влияют на автономное поведение друг друга. При таком моделировании важно выявлять влияние отдельных факторов на поведение системы в целом, а потому предпочтительно проведение повторяемых расчетов и, следовательно, управляемый детерминизм.

Представленные иллюстрации указывают на потребность разработки систем, которые имитируют параллелизм, полностью или частично подменяя его последовательными вычислениями. Такие системы называют *квазипараллельными* (от латинского слова «quasi» – «якобы», «почти»), что подчеркивает генетическую связь логически параллельной программы с ее последовательным вариантом выполнения. Подходы, используемые при разработке квазипараллельных систем, различаются в зависимости от их целей. Ниже мы рассмотрим два в некотором смысле противоположных подхода к разработке квазипараллельных систем. Первый из них

основан на методах разделения процессорного времени между процессами, а второй связан с построением так называемых *систем с дискретными событиями*.

13.10.7. Разделение времени

Механизм *разделения времени* имитирует параллельное выполнение процессов путем выделения для каждого процесса кванта времени для вычислений, исчерпание которого приводит к приостановке процесса и активизации другого на время, определенное его квантом. При этом образуются очереди процессов на выполнение. Варианты организации таких очередей и правил постановки процесса в очереди связываются с принимаемой стратегией распределения процессорных ресурсов по задачам и с введением приоритетов процессов.

Построение очередей процессов при разделении времени – это один из вариантов применения к задаче упорядочивания параллельных вычислений общего метода, с помощью которого множество объектов, не связанных отношением порядка, выстраивается в последовательность, т. е. становится линейно упорядоченным. Этот метод называется *линеаризацией множества*. При разделении времени линейный порядок процессов динамически изменяется: первый процесс очереди, исчерпав свой квант, либо переносится на другое место в очереди, либо, если он выполнил свою программу, исключается из нее. В других применениях линеаризации, например, в методе волны, с помощью которого упорядочивается поиск чего-либо в иерархической структуре, порядок объектов остается неизменным.

Метод линеаризации – одно из самых распространенных средств преодоления сложности взаимодействия процессов. Можно сказать, что все детерминированные переборные алгоритмы являются не чем иным, как выстраиванием процессов обработки в линейном порядке. Однако линеаризация всегда базируется на свойствах конкретного отношения линейного порядка, а потому есть опасность игнорирования других особенностей перерабатываемых данных. Применительно к разработке параллельных программ это предостережение указывает на то, что линеаризация множества процессов не должна нарушать упорядоченность процессов, связанную с зависимостями их по данным.

Имитация параллелизма с помощью линеаризации процессов может быть *полной* – и тогда выполнение программы фактически, т. е. с точки зрения потребления процессорного ресурса, перестает быть параллельным, либо *частичной*, при которой одна часть параллельных процессов выполняется последовательно, а другая – параллельно. В обоих случаях с логической точки зрения программа остается параллельной, хотя и выполняется последовательно.

Как полная, так и частичная линеаризация вполне достаточны для реализации систем поддержки обучения, для отладки. В случае с оптимизацией использования ограниченного процессорного ресурса имитация параллелизма, основанная на линеаризации, часто рассматривается как вынужденная. Поэтому она является обычным механизмом операционных систем, которые в принципе не могут и не должны сверх необходимого знать особенности обслуживаемых процессов. В этой области есть еще одна причина использования разделения времени: требование не допустить потери внешних сигналов. Если интервал, за который сигнал требуется обработать, превышает квант времени функционирующих процессов, то обработчик, которому система устанавливает высший приоритет, успевает зафиксировать необходимую информацию о сигнале. Альтернативное решение задачи, не связанное с выделением квантов, – прерывание текущей задачи и замораживание очереди задач – также употребительно в операционных системах, но его реализация не связана с линеаризацией, которая рассматривается как метод уменьшения недетерминизма квазипараллельных вычислений.

13.10.8. Системы с дискретными событиями

При моделировании требование детерминированности расчетов может оказаться решающим, и в таких случаях вместо частичной и полной линеаризации, основанной на разделении времени, подход, связанный с управляемой дискретизацией вычислений для управления взаимодействиями, оказывается более предпочтительным. В этом плане заслуживает внимания подход, предложенный в языках Simula и Simula-67. Он провозглашает явную управляемую имитацию параллелизма и, как следствие, оставляет для программиста возможность думать о выполнении программы в терминах взаимного влияния процессов друг на друга, составлять программы, которые, ограничиваясь строго последовательными вычислениями,

способны выстраивать поведение автономных и работающих совместно агентов. Это общий метод преодоления противоречия между принципиальным параллелизмом и реально последовательной его реализацией, который оказывается очень естественным способом выражения многих алгоритмов. Речь идет о *системах с дискретными событиями*.

Основой системы с дискретными событиями является так называемый *управляющий список* – строго упорядоченная динамическая очередь процессов, назначаемых для выполнения, которые в терминологии Simula и Simula-67 называются *деятельностями* (activity).

В управляющем списке первая деятельность-процесс – это та, программа которой выполняется. Ее текущее состояние называется *активным*. Состояние каждой из последующих деятельностей управляющего списка называется *приостановленным* – исполнение ее программы прервано, но запомнена *точка возобновления* программы, т. е. сохранена информация, достаточная для организации продолжения деятельности, когда она станет первой в очереди. Помимо процессов-деятельностей, которые в текущий момент находятся в управляющем списке, часть деятельностей системы не представлена в этом списке. Состояние таких деятельностей называется *пассивным*, если для них запомнена точка возобновления вычислений, или в *завершенным* состоянием, если для них точка возобновления не запомнена.

Событие системы с дискретными событиями – это любое изменение управляющего списка, которое появляется из-за действий активной деятельности. В результате этих действий деятельность, которая до этого была пассивной, может появиться в управляющем списке, в частности, стать активной, если до этого пассивная деятельность назначается первой в списке. Пока какое-либо событие не произошло, состояние деятельностей и их положение в управляющем списки не могут измениться. Действия активной деятельности могут привести приостановленную деятельность в пассивное или даже в завершенное состояние. При выполнении действий, изменяющих управляющий список, активная деятельность может оставаться таковой, если ее программой не предусматривается самоприостановка, т. е. перенесение ее перед или после другой деятельности из списка. Активная деятельность может стать завешенной, и тогда следующая деятельность управляющего списка станет активной. Все события системы разбивают вычисления на последо-

вательность *шагов*. Постулируется, что при вычислениях системы с дискретными событиями нарушение целостности данных к моменту начала выполнения каждого шага не допускается.

В выполнении программы, реализующей систему с дискретными событиями, можно выделить три фазы:

- *Инициализация вычислений системы*, в ходе которой происходит начальное заполнение управляющего списка (как минимум, в нем появляется первая деятельность);
- *Цикл вычислений системы*, состоящий из серии дискретных *шагов*, каждый из которых начинается, когда активизируется первая деятельность управляющего списка, и заканчивается, когда выполняемая деятельность-процесс перестает быть активной. Цикл вычислений системы прекращается, когда управляющий список оказывается пустым;
- *Завершение вычислений системы* – обработка результатов работы системы. Поскольку все деятельности на этой фазе находятся в пассивном или завершенном состоянии, они могут рассматриваться как массивы данных для обработки. Вместе с тем есть возможность возобновления цикла вычислений, в частности, с использованием отработавших пассивных деятельностей.

Средства динамического формирования управляющего списка сводятся к тому, что деятельность может быть *вставлена* в него *до* или *после* некоторой деятельности, уже представленной в нем, или удалена из списка. Кроме того, деятельность может быть *назначена на определенное время*, если в системе предусматривается моделирование времени. Это означает, что она вставляется перед той деятельностью в списке, время выполнения которой – минимальное время, превосходящее назначаемое. Таким образом, множество деятельностей динамически упорядочено отношением «раньше», которое должно быть согласовано с порядком деятельностей в управляющем списке: попытка вставить в него деятельность с нарушением временного порядка квалифицируется как ошибка. Вставка деятельности, назначаемой на время, совпадающее с временем деятельности, уже представленной в списке, может корректироваться указанием «до» или «после» такой деятельности.

Таким образом, конструкция управляющего списка имитирует время с помощью средств его динамического формирования. Это именно имитационная модель, создающая структуру, практически неотличимую для внешнего наблюдателя от реального параллелизма процессов, число которых потенциально не ограничено. Отметим, что в системе с дискретными событиями нет нужды явно выдерживать временные задержки. Они являются лишь средством перестройки управляющего списка.

В связи с обсуждением квазипараллельных систем, детерминированных в рамках систем с дискретными событиями, необходимо отметить ряд важных моментов:

- реального параллелизма в системе с дискретными событиями нет, но есть эффект параллелизма, который лишен многих самых неприятных моментов, требующих заботы о синхронизации;
- в каждый модельный момент времени набор одновременно существующих деятельностей упорядочен частично. Он становится полностью упорядоченным за счет текущей расстановки деятельностей в управляющем списке;
- управляющий список – общая структура данных среды выполнения деятельностей, но ее нельзя считать глобальной, так как явный доступ к этой структуре отсутствует;
- если попытаться реализовать систему с дискретными событиями с использованием реального распараллеливания, то придется столкнуться со всеми неприятными моментами параллельных вычислений, требующих специальной заботы о синхронизации процессов.

Идея систем с дискретными событиями впервые была воплощена в реализации в виде библиотечных надстроек над популярными в свое время языками. Очень скоро была осознана необходимость языкового оформления средств оперирования такими системами. Это можно заметить на примере эволюции языка Simula до Simula-67. В соответствии с мотивировкой необходимости поддержки детерминированных вычислений при моделировании системы с дискретными событиями сегодня по-прежнему встраиваются в специализированные языки, ориентированные на моделирование.

ЛИТЕРАТУРА

1. Ахо А., Лам М. С., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. М.: Вильямс. 2021. 1184 с.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы. М.: Вильямс. 2000. 384 с.
3. Бежанова М. М., Поттосин И. В. Современные понятия и методы программирования, учебный курс. М.: Научный мир, 2000. 191 с.
4. Бек К. Экстремальное программирование: разработка через тестирование. Библиотека программиста. СПб.: Питер, 2003.
5. Брукс Ф.П. Мифический человеко-месяц, или Как создаются программные системы. М.: Символ-Плюс. 2010. 304 с.
6. Бульонков М. А. Смешанные вычисления. Учебное пособие. Новосибирск: Изд-во НГУ, 1995.
7. Верещагин Н. К., Шень А. Лекции по математической логике и теории алгоритмов. Часть 3. Вычислимые функции. 4-е изд., исправленное. М.: МЦНМО, 2012. 160 с.
8. Вирт Н., Иенсен К. Паскаль: руководство для пользователей и описание языка. М.: Финансы и статистика, 1982.
9. Вирт Н. Программирование на языке Модула-2. М.: Мир, 1987. 222 с.
10. Вирт Н. Алгоритмы и структуры данных. М.: Мир, 1989.
11. Вирт Н. Построение компиляторов. М.: ДМК Пресс, 2010.
12. Гарсиа-Молина Г., Ульман Дж. Д., Уидом Д. Системы баз данных. Полный курс. М.: Вильямс. 2004. 1088 с.
13. Геллер Д. П., Фридман Д. П. Структурное программирование на АПЛ. М.: Машиностроение, 1982.
14. ГОСТ 19.201-78. Единая система программной документации. Техническое задание. Требования к содержанию и оформлению. http://vsegost.com/Catalog/31/31884.shtml
15. Грис Д. Наука программирования. М.: Мир, 1984.
16. Грофф Дж. Р., Вайнберг П. Н., Оппель Э. Дж. SQL: полное руководство. 3 из. – М.: Вильямс, 2015. 960 с.
17. Дейт К. Введение в системы баз данных. 8-е изд. М.: Вильямс, 2008. 1328 с.
18. Демарко Т., Листер Т. Человеческий фактор. Успешные проекты и команды. М.: Символ-Плюс. 2011. 256 с

19. Део Н., Нивергельт Ю., Рейнгольд Э. Комбинаторные алгоритмы: теория и практика. М.: Мир, 1980.
20. Джехани Ф. Язык Ада. М.: Мир, 1989.
21. Довек Ж., Леви Ж.-Ж. Введение в теорию языков программирования. М.: ДМК Пресс, 2015. 134 с.
22. Ершов А. П. О человеческом и эстетическом факторах в программировании / Ершов А.П. Избранные труды. Новосибирск, 1994.
23. Интернет-университет информационных технологий ИНТУИТ http://www.intuit.ru.
24. Калинин А. Г., Мацкевич И. В. Универсальные языки программирования. Семантический подход. М.: Радио и связь, 1991. 400 с.
25. Кармайкл Э., Хейвуд Д. Быстрая и качественная разработка программного обеспечения. М.: Вильямс, 2003. 400 с.
26. Карпов Ю. С. Теория автоматов. Учебник для вузов. СПб.: Питер, 2002.
27. Карри Х. Основания математической логики. М.: Мир, 1969. 567 с.
28. Касьянов В.Н. Курс программирования на Паскале в заданиях и упражнениях. Новосибирск: НГУ, 2001.
29. Керниган Б., Ритчи Д., Фбюэр А. Язык программирования Си. Задачи по языку Си. М.: Финансы и статистика, 1985.
30. Клейнберг Дж., Тардос Е. Алгоритмы: разработка и применение / Пер. с англ. Е. Матвеева. СПб.: Питер, 2016. 800 с.
31. Кнут Д. Искусство программирования на ЭВМ. М.: Вильямс, т. I–III, 2000.
32. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. Серия: Классические учебники: Computer Science. М.: МЦНМО, 2002.
33. Кук Д., Бейз Г. Компьютерная математика. М.: Наука, 1990.
34. Лавров С. С. Программирование. Математические основы, средства, теория. СПб: БХВ-Петербург, 2001.
35. Липаев В. В. Программная инженерия. Методологические основы. Учебник. М.: ТЕИС, 2006.
36. Липаев В. В. Человеческие факторы в программной инженерии. Рекомендации и требования к профессиональной квалификации специалистов. М.: Синтег. 2009. 328 с.
37. Лутц М. Программирование на Python, том I, 4-е изд. / Пер. с англ. СПб.: Символ-Плюс, 2011. 992 с.

38. Лутц М. Программирование на Python, том II, 4-е изд. / Пер. с англ. СПб.: Символ-Плюс, 2011. 992 с.
39. Мальцев А. И. Алгоритмы и рекурсивные функции. М.: Наука, 1965. 392 с.
40. Мотвани Р., Хопкрофт Дж., Ульман Дж. Введение в теорию автоматов, языков и вычислений, 2-е изд. М.: Вильямс, 2002.
41. Пейган Ф. Практическое руководство по Алголу-68. М.: Мир, 1979.
42. Прата С. Язык программирования С++. Лекции и упражнения. ДиаСофт, 2000.
43. Пярнпуу А. А. Программирование на Алголе и Фортране. М.: Наука, 1978.
44. Савитч У. Язык Java. Курс программирования. 2-е изд. М.: Вильямс. 2002.
45. Саммерфилд М. Python на практике / Пер. с англ. Слинкин А. А. М.: ДМК Пресс, 2014. 338 с.
46. Себеста Р. В. Основные концепции языков программирования, 5-е изд. М.: Вильямс, 2001. 672 с.
47. Скин Д., Гринхол Д. Kotlin. Программирование для профессионалов. – СПб: Питер, 2019. 464 с.
48. Скопин И. Н. Основы менеджмента программных проектов. Курс лекций. Учебное пособие. М.: ИНТУИТ.РУ. 2004. 363 с.
49. Скопин И. Н. Понятия и модели жизненного цикла программного обеспечения: Учебное пособие. Новосибирск: НГУ, 2003.
50. Соммервилл И. Инженерия программного обеспечения. М.: Вильямс. 2002. 624 с.
51. Страуструп Б. Дизайн и эволюция С++ / Пер. с англ. М: ДМК Пресс; СПб: Питер, 2006. 448 с.
52. Страуструп Б. Язык программирования С++. 3-е изд. / Пер. с англ. М: М: ДМК Пресс; СПб: Питер, 1999. 991 с.
53. Стюарт Т. Теория вычислений для программистов. М.: ДМК Пресс, 2013. 384 с.
54. Тамре Л. Введение в тестирование программного обеспечения. М.: Вильямс, 2002. 368 с.
55. Терехов А. Н. Технология программирования. М.: ИНТУИТ.РУ 2006. 152 с.
56. Технология [Электронный ресурс]. – url: https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%

D0%BE%D0%B3%D0%B8%D1%8F (дата обращения: 1 июня 2022).
57. Фролов Г. Д., Олюнин В. Ю. Практический курс программирования на языке PL-1. М.: Наука, 1983. 384 с.
58. Хантер Р. Основные концепции компиляторов. М.: Вильямс, 2002. 256 с.
59. Хьюз Дж., Мичтом Дж. Структурный подход к программированию. М.: Мир, 1985.
60. Шень А. Программирование: теоремы и задачи. М.: МЦНМО, 1995.
61. Шилдт Г. C# 4.0: полное руководство. М.: Вильямс, 2021. 1056 с.
62. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. М.: Вильямс, 2003.
63. W3 Consortium: XMLTechnolgy. Электронный ресурс. URL: https://www.w3.org/standards/xml/ (дата доступа: 1 июня 2022).
64. XML.ORG. Электронный ресурс. URL: https://www.xml.org/ (дата доступа: 1 июня 2022).

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

анализ типов 117
динамический 119
статический 70, 119, 120

анонимные объекты 115

архитектура фон Неймана 16, 308, 331

ассемблер ....См. редактор связей

блок 111

блочная структура 111

Бэкуса-Наура форма 54
регуляризованная 54, 55, 57

ввод/вывод программы 249
буферизованный 251
низкоуровневый 250
стандартный 252
файл 249
логический 249
физический 249
форматный 253

виды управления 172
выражение 172, 173
последовательное выражение 176
обработка исключения 172, 230
нелокальные переходы 235
оператор 172, 177
блок 177
ветвление
переключатель 181
условный 181
выражение 177
перехода 178
пустой 177
цикл 187
процедура / функция 172, 196

виртуальная машина 32

включаемые файлы 36

вложенные процедуры 211

вычисления
квазипараллельные 350
линеаризация 352
разделение времени 352
с дискретными событиями 354
параллельные ....См.вычисления параллельные
последовательные 313
распределенные 321
совместные 313

вычисления параллельные
гонка данных 333
квантование времени 315
критическая область 320
монитор 337
мьютекс .См. вычисления параллельные, семафор бинарный
обмен сообщениями 323
семафор бинарный 321
семафор Дейкстры 319
синхронизация 331

завершение оператора 178

загрузчик 37

задача
алгоритмическая 258
возведение в степень 265
вычисление полинома в точке 265
подсчет количества ненулевых бит 259

импорт библиотеки 114

интерпретатор 28, 29, 34

информационное множество 269
(не)деструктивное 270
битовые шкалы 272–274
деревья поиска 278

бинарные 279
высота 280
корневые....280
ориентированные 280
сложность операций ....См. деревья поиска, высота
деревья поиска сбалансированные
2-3-деревья....293
B-деревья 294
WB[α]-деревья 285
АВЛ....282
идеально 280, 285, 291, 292
красно-черные 287
оптимально....292
по весу .... См. WB[α]-деревья
с вычислением штрафа....289
сильноветвящиеся ....См. B-деревья
ключ объекта 269
массивы....274
неупорядоченные....275
упорядоченные....276
операция
добавление/вставка элемента....271
инициализации 270
проверка принадлежности элемента ..271
удаление элемента 271
списки.См. информационное множество, массивы
универсум объектов....269
квалифицированное использование имени 112, 114
классификация архитектур компьютеров....341
MIMD....346
MISD....345
SIMD....343
SISD,....342
ключевое слово....51
компилятор См. транслятор
конечный автомат 49, 304
контекстно-свободная грамматика 54
вывод 57
однозначность....60
выражений....57, 59, 62
дерево вывода....58
однозначность....58
дерево разбора .... См. дерево вывода
нетерминал 54
главный....58
терминал 54
контроль типов....См. анализ типов
конъюнкция и дизъюнкция по МакКарти 175
кросс-компилятор.....См.транслятор, кросс-компиляция
лексема....47, 54
значение 47
класс....47
максимально возможная 50
нормализация....51
привязка к тексту 48
линейный порядок 270, 276, 279, 296
Машина Тьюринга....74, 258
метод открытой адресации.... См. метод перехэширования
метод перехэширования 299
вторичный хэш.... 299
коэффициент загрузки.... 299
перестроение таблицы.... 299
сложность 300–301
удаление элемента.... 299
функция перехэширования 300
метод прямого связывания.... См. метод цепочек
метод цепочек 298
сложность 298
метод раскрутки....33
объектный модуль....36
однозначность определения имени.... 111

операционная система 12, 18
память программы
  автоматические объекты 239
  глобальные объекты 239
  динамические объекты 240
    автоматическая сборка мусора 244
    утечка памяти 242
перегрузка операций 117
пессимизация вычислений 266
полиморфизм 118
правило видимости 111
прагматика языка 89
преемственность языка 90
  обратная совместимость 91
препроцессор 36, 92–110
  генерация лексем
    операции # и ## 108
  директива 92
  директива #define 94
  директива #include 36, 102
  директива #undef 102
  директивы #ifdef, #ifndef, #else, #elif, #endif 103
  макрос 92, 94
  псевдо-макрос 96
    __DATE__ 96
    __FILE__ 96
    __LINE__ 96
    __TIME__ 96
    defined 96
присоединяющий оператор 113
программа
  верификация и анализ 42
  документация 40
    пользовательская 40
    системная 40
    техническая 40
  комплексация .... См. сборка
  отладка 41
    инструментирование 41
    интерактивная 41
  поддержка версий 43
  распараллеливание 314
  рефакторинг 40
  сборка 36
  тестирование 42
    критерии 42
    регрессионное 42
  управляемая данными 306
  управляемая событиями 329
программирование
  параллельное 315
    скобки parbegin/parend 315
  предметно-ориентированное 307
  прикладное 9
    безопасность 10
    гуманитарные требования 10
    интуитивная понятность 10
    надежность 10
    эффективность 10
  системное 12
  теоретическое 15
  технология .... См. технология программирования
процедура .... См. функция
раннее обнаружение ошибок 43
регулярное выражение 48, 311
  итерация Клини 56
  ненулевая итерация 56
  операция повторения .... См. итерация Клини
редактор связей 37
семантика 71
  аксиоматическая 79
    постусловие 80
    правила вывода 80–81
    предусловие 80
    тройка Хоара 80

денотационная 71
семантическая функция ...См. семантическая функция
операционная 74
языковая машина 75–79
формальное описание 71
семантическая функция 45, 71–74
синоним 219
синтаксис
абстрактный 66
дерево 67
контекстно-зависимый анализ 69, 70
контекстно-свободный 54
устойчивость к ошибкам 63
синтаксическая диаграмма 54, 61
система программирования 12
редактор связей ....См. редактор связей
система построения 38
среда разработки 38
система типов данных 115
строгая 120
система управления базами данных 12, 309
сложность алгоритма 258
амортизационная 272
в среднем 259
в худшем 258
временная 258
емкостная 259
оценка 258
консервативная 258
точная 258
смешанные вычисления 35
среда программирования
рефакторинг 66
среда разработки
анализатор семантических свойств 42
документирование 39
интегрированная 39
многоязыковая 38
отладчик 41
поддержка версий 43
профилировщик 41
система тестирования 42
справочная система 39
этап анализа и проектирования 39
языково-ориентированный редактор 39
стиль программирования 83
комментарии 88
мнемоничность идентификаторов 87
неиспользование умолчаний 86
стиль программирования: 83
СУБД ....См. система управления базами данных
схема Горнера 267, 297
таблица расстановок .... См. хэш-таблица
Т-диаграммы 29–35
теория сложности
машинно-зависимая 257
машинно-независимая 256
технология программирования 13
тип данных 115
абстрактный 116
арифметический 121
в языке С 145
вещественный 126
потеря точности 128
с плавающей точкой 127
с фиксированной точкой 126
интегральный ....См. перечислимый
литеральные значения 116
логический 122
массив 137
динмический 139
контроль индексов 138
многомерный 138
подвижный 140
статический 137

множество 130
моделируемая категория 116
набор операций 116
объединения 134
определяемый 121
перечисление 131
перечислимый 121
подтип 118
предопределённый 121
приведение 118, 152
простой 121
реализация 116
символьный 122
синтаксис 116
строковый 144
структура 133
структурированный 121
указатели 136
типизированные и нетипизированные 136
упорядоченный 121
целый 124

транслятор 28, 29, 30, 32, 36
кросс-компиляция 35

трансляция
байт-код 32
лексический анализ 46, 47, 50
многоуровневая 32
многофазная 30, 31, 37, 46
раздельная 38
семантический анализ 46
синтаксический анализ 46, 54

трансформационный подход 202

управление в программе 172

функция
thunk 222
вызов 196
граф вызовов 199
дерево вызовов 200
поколения локальных переменных 201
шаги исполнения 198
главная 198
обратного вызова 215
описание 196
параметры
именованные 228
необязательные 228
подстановка параметров 217
по значению 217
по значению-результату 220
по имени 223
по необходимости 221
по ссылке 218
рекурсивная 199

хэш 295–300
вторичный ...См.метод перехэширования
код 295
коллизия 296
первичный 296
таблица 295
функция 295, 297
идеальная 297

эквивалентность типов данных 116, 139
именная 116
структурная 117

язык
национальная версия 52
программирования
алгоритмический высокого уровня 21
императивный 25
макро-ассемблер 21
машинный 20
мнемокод 20
функциональный 26

Учебное издание

Бульонков Михаил Алексеевич, **Емельянов** Павел Геннадьевич,
Скопин Игорь Николаевич

БАЗОВЫЕ ПОНЯТИЯ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Редактор О. Э. Вульф
Верстка О. А. Тенекеджи
Обложка Е. В. Неклюдовой

Подписано в печать 04.09.2023 г.
Формат 60x84 1/16. Уч.-изд. л. 22,9. Усл. печ. л. 21,3.
Тираж 60 экз. Заказ № 192
Издательско-полиграфический центр НГУ
630090, Новосибирск, ул. Пирогова, 2.