МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

М.А. Бульонков и П.Г. Емельянов

БАЗОВЫЕ ПОНЯТИЯ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Новосибирск
2012

УДК
ББК

Бульонков М.А., **Емельянов П.Г.** Базовые понятия и методы программирования: Учебное пособие / Новосиб. гос. ун-т. Новосибирск, 2012. **Ошибка! Закладка не определена.** с.

ISBN

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, объектно-ориентированное программирование и проектирование, а также проблематику организации взаимодействия человека и машины посредством графического интерфейса и организации больших структурированных хранилищ информации.

Рецензенты:

Учебное пособие разработано в соответствии с требованиями ФГОС ВПО к профессиональному циклу основных образовательных программ по группе 01 направлений подготовки (математика, механика, информатика). Издание подготовлено в рамках реализации *Программы развития государственного образовательного учреждения высшего профессионального образования «Новосибирский государственный университет» на 2009-2018 гг.*

ISBN

1 ПРЕДИСЛОВИЕ

Это пособие основывается на конспектах лекций, которые читались на первых курсах ММФ Новосибирского госуниверситета в течении последних ??. Отправной точкой в развитии этого курса являлось пособие «Современные средства и методы программирования», написанное в 19?? году М.М.Бежановой и И.В.Поттосиным, которые в свою очередь стояли у истоков преподавания программирования в НГУ. Понятно, что многое поменялось за это время: появлялись и сходили на нет языки программирования, кардинально повысилась мощность и, что ещё важнее, доступность вычислительных машин, формировались и приобретали первостепенное значение новые области применения. Как следствие, изменялось и представление о профессиональных знаниях, умениях и навыках, которыми должны обладать программисты. Поэтому 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 объёме школьного курса «Основы информатики» и соответствующими навыками работы с компьютером и способен самостоятельно написать 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 скажем, что в данном курсе это язык C. Однако, ни в коем случае данное пособие

не следует рассматриваеть 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 в совершенстве освоить конкретный язык или стать компьютерным полиглотом, а в том, чтобы научиться ставить вопрос: «Почему так и как можно было бы по-другому?».

2 ВВЕДЕНИЕ

Программирование можно определить как способ заставить кого-то достичь поставленной нами цели. Часто в качестве примера программирования приводят кулинарные рецепты, которые описывают пошаговый процесс приготовления из исходных продуктов некоторого блюда. Программой может служить и математический алгоритм. Например, алгоритм Евклида задаёт последовательность операций, которые достаточно выполнить для нахождения наибольшего общего делителя по двум заданным целым числам. Однако следует иметь в виду, что последовательность и пошаговость не является неотъемлемой чертой программирования. Даже в случае с кулинарным рецептом некоторые операции можно выполнять параллельно. Если же мы в качестве примера программы рассмотрим Правила дорожного движения, целью которых является обеспечение безопасности и избежание заторов на дорогах, то мы заметим, что это 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 данного курса 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.

2.1 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 таким вопросам, как

Надёжность, устойчивость: 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».

Интуитивность, удобство 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, 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.

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.

Гуманитарные аспекты, к которым можно отнести учёт того, что пользователь может плохо различать цвета, иметь физические проблемы с использованием клавиатуры, не знать иностранных языков и т.п.

2.2 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. К системным программам можно отнести следущее:

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 объёмам информации.

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

2.3 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. Этим занимается *технология программирования*. Помимо собственно составления программ (называемого также *кодированием*) технология должна поддерживать весь *жизненный цикл* программы, в частности:

Спецификацию создаваемого программного обеспечения, проверку соответствия требованиям заказчика.

Проектирование, то есть разработку общей архитектуры системы, взаимодействие компонентов и т.п.

Отладку и тестирование – нахождение и своевременное исправление ошибок. Идеальная технология программирования не должна допукать 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.

Документирование в виде как инструкций для пользователя, так и технического описания самой программы для тех программистов, которые будут её далее развивать или сопровождать.

Сопровождение, версионность – реакцию на замечания и рекламации пользователей, необходимость поддерживать несколько версий программы и т.п.

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

2.4 ТЕОРЕТИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Сложность реальных программ для анализа и обработки приводят к необходимости построения формальных моделей и 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;

Алгебра , логика, теория алгоритмов – при формальном описании семантики программ и верификации, то есть проверки и доказательства соответствия программы спецификации;

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

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

3 «ОКРУЖЕНИЕ» ПРОГРАММИРОВАНИЯ

3.1 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 много другое, что подробно обсуждается в курсе по архитектуре ЭВМ. Нам будет достаточно схемы, представленной рис.1:

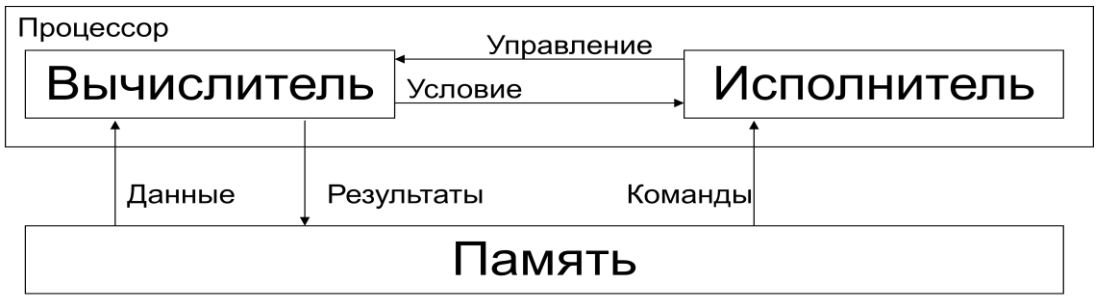


Рис. 1. Структура ЭВМ

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, то есть 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 битов. Из четырёх байтов формируется слово, с которым может оперировать машинная команда. Схематично это отображено на рис.2.

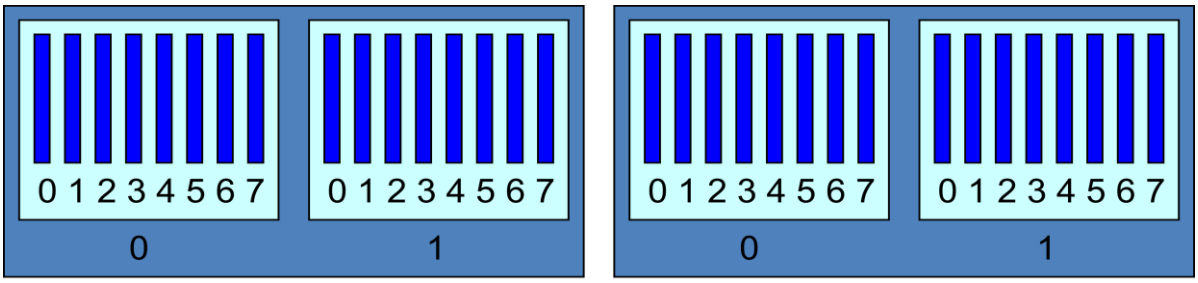


Рис. 2. Дискретная память

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 битов;
- 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 моём 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 придётся 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.

3.2 ОПЕРАЦИОННАЯ СИСТЕМА

Для того, чтобы программа попала в память и начала выполняться, необходимо выполнить определённые действия. Изначально эти действия выполнял *оператор* ЭВМ, заполняя с помощью тумблеров нужную область памяти кодом программы и её данными и нажимая 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, называется *операционной системой* (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, сетевой трафик 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 запрещённую 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, флеш-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.

Надо отметить, 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 использований тех функций, которые предоставляет её «содержимое».

4 ПОКОЛЕНИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

4.1 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.

4.2 МНЕМОКОД

Естественно, что запись программ в двоичном коде воспринимается сложно. Ещё сложнее внести 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 даёт *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.[1]
- Вместо номеров команд и ячеек данных использовать их мнемоничные имена.

[1] Здесь и далее в этой главе нас интересуют в первую очередь содержательные понятия, а не способ их оформления в конкретном языке. Поэтому здесь 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, то есть 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.

4.3 МАКРО-АССЕМБЛЕР

Поскольку машинные команды (обычно) выполняют только самые элементарные действия, то содержательные действия чаще всего реализуются некоторой последовательностью 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, указать изменяемые части, чтобы в дальнейшем не выписывать её снова и снова, а указать там, где требуется, 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. Естественно, что именование предпочтительнее нумерации с точки зрения понимаемости. 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 четырёх команд: вплоть до команды ENDM, означающей конец макроса.

Место, где макрос используется, 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
```

В первом вызове вместо формального параметра C1 подставляется текст DI, вместо C2 – A, вместо CP – S1, вместо MP – S2, а во втором вызове – вместо C1 – S, вместо C2 – 2, вместо CP – DI, вместо MP – SI. Если мы подставим тело макроса в место вызова, произведя замену формальных параметров на фактические, то мы получим 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
```

Налицо существенное сокращение записи. Отметим, что макросредства задают по существу преобразование текста программы: выполнив все подстановки вызовов макросов, мы получим текст программы на мнемокоде, которая уже напрямую транслируется в машинную программу. Более того, работа *макропроцессора* – программы, осуществляющей преобразование программы с макро-определениями – не зависит от того, что именно записано в теле функции – мнемокод или любой другой текст. Для неё существенно только то, как оформляются макроопределения и вызовы макросов.

Отметим ещё 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 использования этой библиотеки. Далее эта возможность способствует совместной разработке большой программной системы коллективом разработчиков.

4.4 АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ ВЫСОКОГО УРОВНЯ

Хотя макро-ассемблер и предоставляет средства расширения языка путём определения новых «команд», он остаётся «привязанным» к конкретному машинному языку. Следующий шаг в повышении уровня языка был сделан с появлением *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 определённого 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, и в этом смысле все одни эквивалентны. Однако, некоторый язык может оказаться удобнее, компактнее, выразительнее, чем другие, для решения конкретного класса задач. Существует 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 традиционно считается ориентированным на научные расчёты. 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 Фортран для научных расчётов, но и имеют определенные преимущества, если рассматривать другие языковые аспекты. Кроме того, сам по себе уровень языка и его выразительные возможности не предохраняют от написания в нём 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 *парадигмы*, то есть базовой системы понятий, на которой они основаны. Ниже мы кратко рассмотрим наиболее

распространённые 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.

4.4.1 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++ и десятки других. Были попытки создания универсального, подходящего для всех прикладных областей, языка программирования - Алгол 68, PL/I, Ada - но успешными их назвать нельзя. Новые языки программирования появляются до сих пор.

4.4.2 Функциональные языки

Свойственное императивным языкам 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 возможность для параллельного вычисления подзадач, являются функциональные языки программирования: Lisp, Scheme, Miranda, ML, Haskell, Scala и т.д.

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 определённые. Характерными для функциональных программ является отсутствие (или очень ограниченное использование) изменения состояния памяти, а также *рекурсия*, то есть 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.

5 РЕАЛИЗАЦИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Алгоритмические языки высокого уровня в отличие, скажем, от языка макроассемблера, в значительной степени потеряли непосредственную связь с машинным языком. Это дистанцирование, то есть стремление сделать язык машинно-независимым, приблизить его к предметной области и алгоритмам, потребовало создания инструментов, позволяющих выполнить 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.

5.1 ИНТЕРПРЕТАТОРЫ

Интерпретатор можно сравнить с переводчиком-синхронистом, который воспринимает очередную, обычно достаточно небольшую фразу и сразу её 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 потребовать 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 итерпретатор 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 её 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.

5.2 ТРАНСЛЯТОРЫ

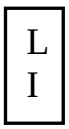
В отличие от интерпретатора транслятор можно сравнить с литературным переводчиком. Он получает текст либо сразу целиком, либо значительную его часть, имеет возможность прочитать и проанализировать текст многократно, сделать подстрочный перевод и лишь в конце составить окончательный текст. Именно то, 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 её на другой язык.

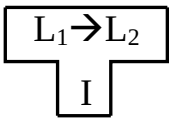
5.3 Т-ДИАГРАММЫ

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



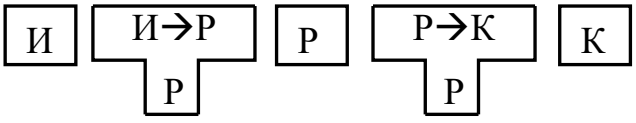
Этот кирпичик можно рассматривать как элементарное устройство, на которое сверху можно положить любую программу на языке L, и, если это устройство удастся заставить работать, то заработает и положенная программа. Заставить же работать сам интерпретатор можно, например, положив этот кирпичик (как программу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

Для графического изображения транслятора используется Т-блок, в основаниии которого указывается язык реализации, слева – входой язык транслятора, справа – выходной:

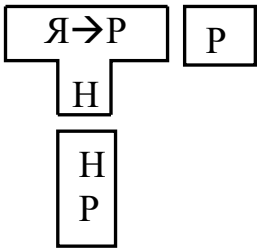


Входные данные, т.е. программы на языке L_1 подаются слева, а результат указывается справа.

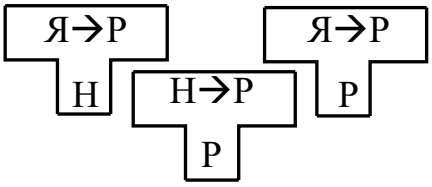
Из таких кирпичиков можно строить разнообразные схемы, соблюдая при этом правила стыковки. *Многофазная трансляция* может быть полезна для разбиения сложного процесса на последовательность более простых или доступных переводов. Она означает, что перевод с исходного на конечный язык осуществляется не 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 быть выглядеть как



где блоки, помеченные буквами И, Р и К означают тексты на итальянском, руском 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 языке, скажем на немецом (Н), то для того, чтобы её выполнить нам потребовалась бы помощь интерпретатора,



либо следовало предварительно перевести инструкцию с немецкого на русский,

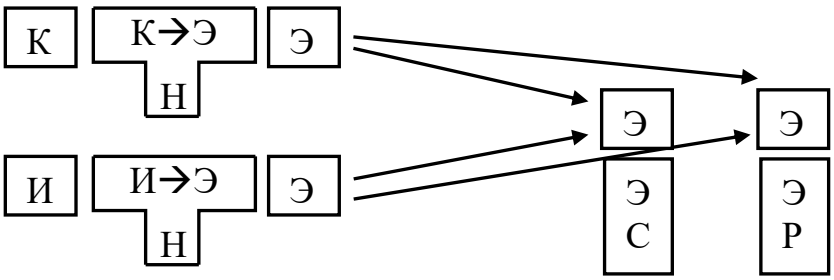


и затем использовать её по старой схеме.

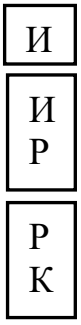
Типичным примером многофазной трансляции является трансляция с языка высокого уровня в язык ассемблера, который затем траслируется в машинные команды «штатным» транслятором. Пошаговая трансляция может состоять из более чем двух шагов. Так, например, одна из возможных реализаций языка C++ состоит в том, чтобы сначала перевести его в язык С, затем с языка С – 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 путём трансляции в машинные команды, либо интерпертацией. 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. Причём совершенно неважно, на каком языке осуществляется перевод, например, на намецком (Н). Одним из примеров такого общего языка в практике программирования выступает *байт-код*, реализуемый на разнообразных встроенных или мобильных устройствах интерпретаторами (*виртуальными машинами*).

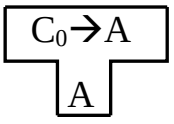


Промежеточные языки могут использоваться не только для трансляции, но и для *многоуровневой интерпретации*. Вернёмся к примеру перевода с итальянского на китайский. Для того чтобы обеспечить синхронный перевод, мы можем привлечь двух переводчиков: первый будет слушать итальянскую речь и тут же повторять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

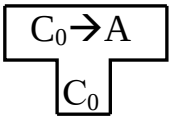


Ясно, что качество такого перевода вызывает сомнения, поскольку, первому переводчику для точного перевода краткой и ёмкой фразы на итальянском может потребоваться пространный текст на русском, второй же переводчик, не зная, что сказал итальянец, будет ещё более пространно излагать русскую речь на китайском.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

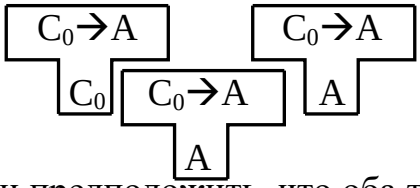
Каким же образом транслятор с некоторого языка появляется на машине? Представим себе ситуацию, когда нам принесли новую машину, на которой ничего нет, кроме операционной системы и (чтобы немного смягчить ситуацию) языка ассемблера, а нам требуется создать работающий на этой машине транслятор с языка С. Конечно, можно просто засучить рукава и начать писать требуемый транслятор на ассемблере, но ввиду того, что язык С далеко не самый простой, а язык ассемблера далеко не самый удобный и надёжный, 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 А



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

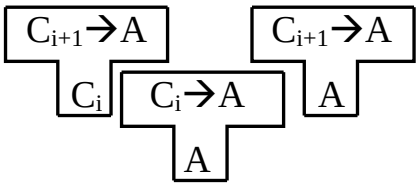


Поскольку второй транслятор является программой на языке C_0, то мы можем применить к нему первый транслятор



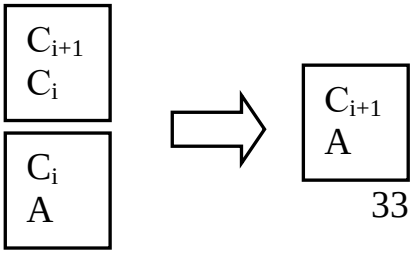
Заметим, что если предположить, что оба транслятора эквивалентны, но написаны на разных языках, то в результате такого применения получится ещё один эквивалентный транслятор, также написанный на языке ассемблера.

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 учётом 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:



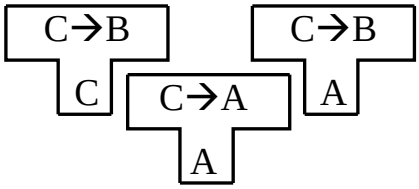
В конце этого процесса, когда будут включены уже все конструкции языка C, осталось выкинуть всё лишнее, что мы вставили в C_0.

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 интерпертаторов в один:

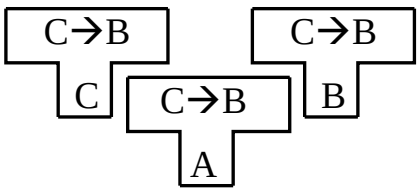


Один из возможных подходов к реализации этого преобразования дают *смешанные вычисления*, которые выходят за рамки данного курса.

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 геренерирующий 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) её 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.

6 СИСТЕМЫ ПРОГРАММИРОВАНИЯ

Исполнение программы, о котором мы говорили в предыдущем разделе, то есть 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 её создании нескольким разработчикам и, во-вторых, использовать стандартные составные части.

Комплексация программ может проходить на нескольких уровнях. Самый простой способ - собирать программу на уровне исходного текста, подобно тому, как используются библиотеки макросов в макроассемблере. Для этого язык должен поддерживать макрообработку и предоставлять возможность «вставить» в указанное место текст из другого файла. В языке С это реализуется директивой

```
#include "имя файла"
```

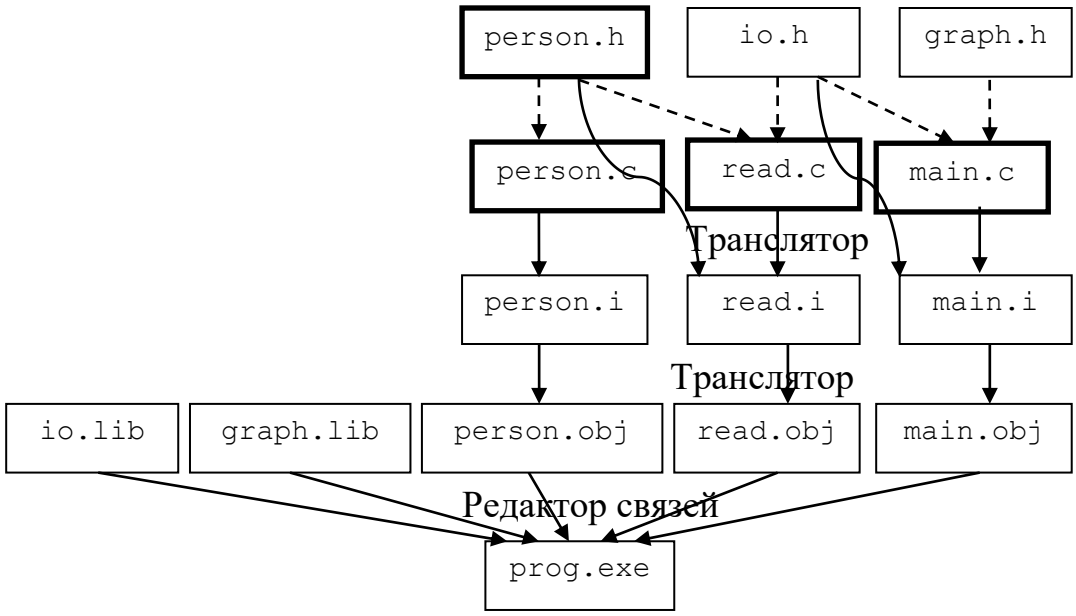
а *включаемые файлы* обычно имеют расширение `.h`. Включаемые файлы чаще всего содержат описание используемых структур данных и функций. Формирование подлежащих компиляции файлов из текстовых заготовок выполняет *препроцессор*. Получаемые на выходе файлы уже не содержат директив препроцессора и имеют 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), определённые 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. Таким образом, на этом этапе инеобходимо 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 её 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, а пунктирные стрелки указывают на наличие директив препроцессора:



Количество фаз трансляции больше двух, например, когда файлы на языке С 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 её схема, то есть 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 даёт 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, путём просмотра исходного текста на языке C и обнаружение 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 приведённом 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, низкоуровевый 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 инструменами), 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 определённом входном языке, то естественно ожидать от него автоматического форматирования, раскраски текста и т.п. Кроме того, пользователь заглядывает в текст программы не только в момент её 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, причём пронизывающая весь путь преобразования программы. Таким образом, система программирования чаще всего включает в себя собственный *языково-ориентированный текстовый редактор*, а мы можем говорить о *интегрированной среде разработки*, для которой редактор является основым интерфейсом, через который

осуществляется не только изменение тескта программы, но и все другие действия.

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 предопределённых конструкциях и объектах, но и об определённых 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:

- *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.

Включённость в систему программирования позволяет редактору осуществлять и более сложные реадактирующие действия. Например,

помимо простого текстового поиска редактор может использовать информацию, полученную от транслятора и загрузчика, для отображения *перекрёстных ссылок*: перехода от использующего вхождения переменной или функции к соответствующему определению и наоборот – поиска всех использований объекта, причём 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 её вывод и, если повезёт, понять причину ошибки. *Интерактивная отладка* дополнительно позволяет задавать точки останова, достигая которые, 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-ый 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.

Ещё одна задача, 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 привело к новой. Альтернативой отладки в этом смысле является *тестирование*, цель которого состоит нахождении такого набора входных данных, правильная работа программы на которых, если и не доказывала, то убеждала бы в её правильности. Система программирования может предоставлять средства для автоматического построения набора тестов как для отдельных компонент программы (например, функций), так и программы в целом. Постороение 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) даёт 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.

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

- любой переменной присваивается некоторое значение до момента её 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 алгоритмическими разрешимыми и даже частные решения, т.е. находящие не все потенциальные ошибки, могут быть чрезвычайно трудоёмкими. Тем не менее, ущерб, который наносит ошибка при использовании программы, может многократно превосходить затраты на её исключение при разработке. Поэтому доказательное программирование, реализующее принцип *раннего обнаружения ошибок*, является перспективным направлением создания надёжного программного обеспечения.

Самым трудоёмким 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 может 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 своём развитии от собственно исполнения программы, и решаемые ими задачи намного сложнее и комплекснее.

7 ЯЗЫКИ ПРОГРАММИРОВАНИЯ

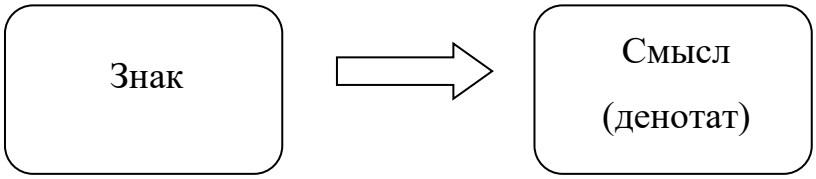
Мы не будем детально обсуждать вопрос о том, как реализуются трансляторы и интерпретаторы - это тема отдельного курса. Но нам необходимо понимать, какой смысл заключён в программе как в тексте на языке программирования хотя бы для того, чтобы наше представление соответствовало тому, что реализует система програмирования. 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, однозначнее, а их правила могут быть в значительной степени описаны формально[2]. Ниже мы рассмотрим некоторые аспекты языков программирования и способы их описания.

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



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

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 мы дали такой ответ, то всё 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 определённой 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, всё 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, хотя в русской нотации для этого следовало бы использовать запятую.

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

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

Аналогичные, но гораздо более сложные процессы происходят и при сопоставлении смысла программам, исходным представлением которых является последовательность символов[3], а конечным, например, функция, реализующая отображение входных данных 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 её класс - идентификатор, служебное слово, число, служебный символ и т.п. Отметим, что лексемы, хотя и могут иметь привязку к исходому тексту, являются абстрактными объектами, и для последующих стадий уже не важно, каким образом было получена, скажем, лексема "действительно 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, подчинённым оборотом и т.п. Из предложений могут собираться параграфы, сноски, разделы, эпиграфы, главы, послесловия т.д. Таким

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

образом, результатом анализа является иерархия объектов, относящихся к определённым 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 определённые 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 фаз осмысления текста есть и другие важные свойства, такие как лаконичность, стиль и т.п.

7.1 ЛЕКСИКА

Лексика языка программирования описывает, из каких слов строятся фразы, то есть 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[4] лексем, каждый элемент которого содержит

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

[4] 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.

`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 подчерка в идентификаторе, как в `student_count`, а если допускается, то в каком месте? Может ли идентификатор начинаться с подчерка или заканчиваться им, как в `__FILE__` и `_1`? А состоять из одних подчерков, 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 они входят в имя идентификатора или игнорируются?

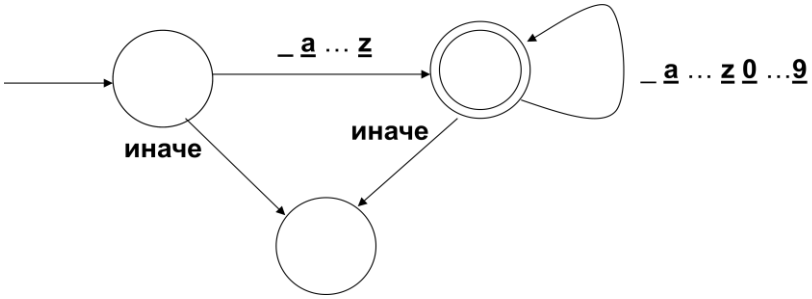
Таким образом, опредение лексики требует более строгого, формального описания. Наиболее часто для этой цели используются регулярные выражения. Так, например, описание идентификатора может быть записано следующим образом:

(_|[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 определённый смысл. Приведённое 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 допускаемым 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. 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 ведёт 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.

- 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 путём выделения максимальной возможной лексемы.

После того как мы выделили из входного потока текст очередной лексемы, необходимо вычислить её 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, то есть приведение их к некоему нормальному виду, например:

- Числа `1.23` и `123e-2` приводится 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 приводится к форме 0.

Некоторые идентификаторы преобразуется в специальные лексемы, называемые *ключевыми словами*, которые играют особую роль в определении синтакиса. 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 нём 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 могут уже использоваться 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 определённом 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, некототорые 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 три версии процедуры на языке Алгол-60 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
```

```
procedure НОД(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;
   for u:=ОСТ(x,y) while u ≠ 0
   do
    begin
     y := x; x := u
    end;
  end;
  z := x
end
```

```
проц НОД(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 объёме словаря ключевых слов языка программирования, то ему подойдёт и вторая версия. Однако, он должен отдавать себе отчёт 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.

7.2 СИНТАКСИС

Имея на входе поток лексем, синтаксический анализ разбирает структуру предложения языка программирования. В этом разделе мы будем рассматривать контекстно-свободный синтаксис, то есть такой, при котором структура фразы определяется вне зависимости от того, в каком месте эта фраза написана. Для задания контекстно-свободного синтаксиса мы будем использовать РБНФ и синтаксические диаграммы.

7.2.1 Форма Бэкуса-Наура (БНФ)

Далее, для описания синтаксиса мы будем использовать *контекстно-свободные грамматиками*, 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 даёт «толкование» некоторому определяемому понятию,

называмому также *нетерминалом*. Для одного и того же нетерминала может быть несколько правил, и в этом случае они рассматриваются как альтернативы. *Терминальными символами* (или сокращённо *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** подчёркнутым. Тогда каждое правило грамматики имеет вид

нетерминал ::= последовательность терминалов и нетерминалов

Значок «::=» читается как «это». В таких обозначениях рассмотренное выше определение идентификатора может быть представлено следующей БНФ-грамматикой:

буква ::= _
буква ::= a
...[5]
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 ::=

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

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*]

Повторение некоторой конструкции ноль или более раз задаётся 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 к следующему виду:

буква ::= **_**|**a**|…|**z**

цифра ::= **0**|…|**9**

идент ::= *буква* (*буква* | *цифра*)*

Переход от БНФ к РБНФ не привносит никакой содержательной информации и служит только для сокращения записи – любую РБНФ можно преобразовать обратно в БНФ.

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 следущий вывод:

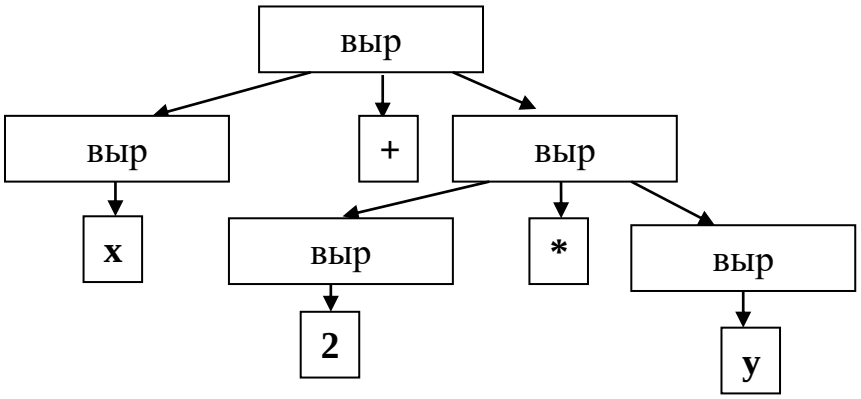
Выр
выр **+** *выр*
x **+** *выр*
x **+** *выр* ***** *выр*
x **+** **2** ***** *выр*
x **+** **2** ***** **y**

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 то, что мы используем контекстно-свободную грамматику, делает этот выбор 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 её можно вывести из главного нетерминала.

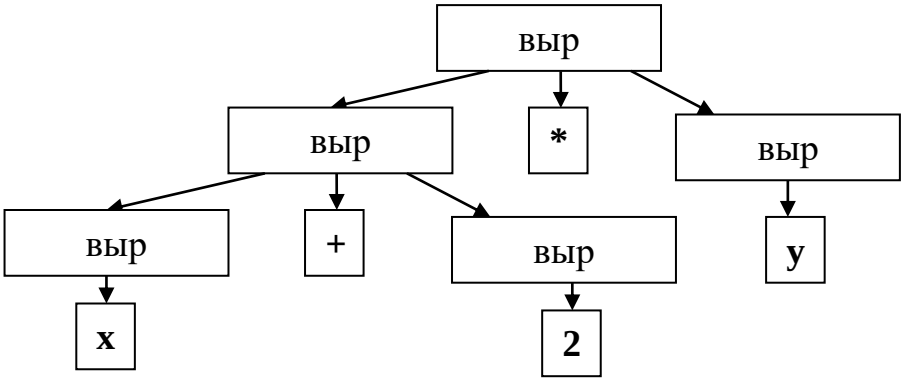
Для того, чтобы сохранить и отобразить процесс вывода, используют *деревья вывода*, называемые также *деревьями разбора*. Корнем этого дерева является начальный нетерминал, листьями – терминалы, а любая вершина, являющаяся нетерминалом, имеет ровно столько потомков и ровно в той последовательности, сколько 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 [Ахо,Ульман]. Отметим только, что эта задача с теоретической точки зрения в общем случае эффективно разрешима.

Поскольку дерево разбора задаёт 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. В нашем случае 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:

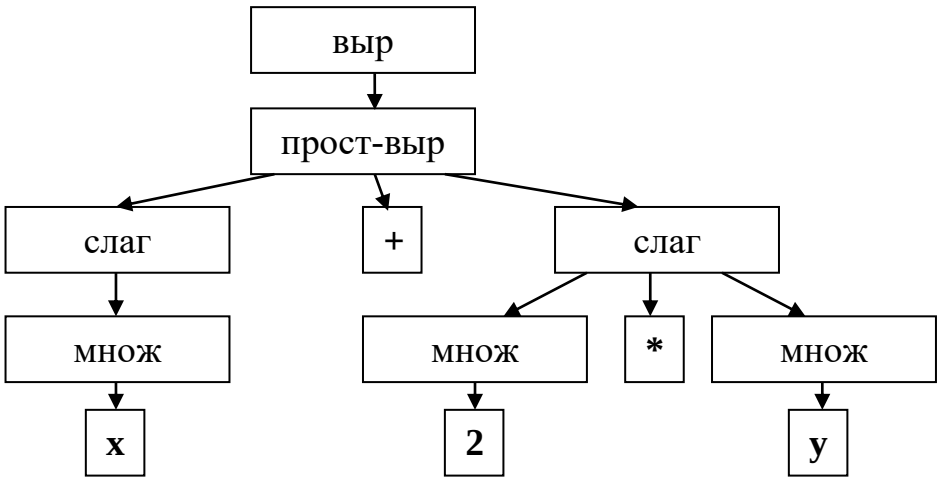
выр ::= *прост-выр* [(= | < | <= | <>) *прост-выр*]

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 произведённые 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 придётся показать, что при изменении грамматики мы не повлияли на распознаваемый язык. Это поднимает вопросы об доказательстве эквивалентности и системах эквивалентных преобразований контекстно-свободных грамматик, но они выходят за рамки данного курса.

В некоторых случаях избавиться от неоднозначности путём преобразования грамматики не удаётся. 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 её 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
```

7.2.2 Синтаксические диаграммы

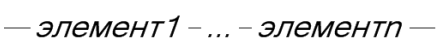
Более наглядный способ задания контекстно-свободных грамматик представляют синтаксические диаграммы: определение нетерминального символа задаётся в виде структурированного ориентированного графа с одним входом и одним выходом, вершинами которого являются нетерминалы и терминалы. Вход и выход обозначаются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

Вход 

Выход 

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

Обязательные элементы

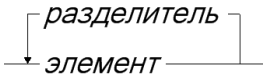


Необязательный элемент

Игнорируемый элемент

Повторение элемента

Повторение через разделитель



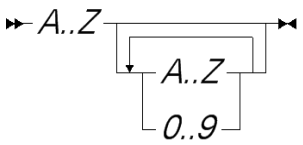
Здесь под *элементами* подразумеваются либо нетерминалы, либо терминалы, либо такого же вида подграфы. Отличие игнорируемого элемента от необязательного состоит в том, что его отсутствие не влияет на смысл программы. Таким образом, игнорируемые элементы используются только для улучшения читаемости программы 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

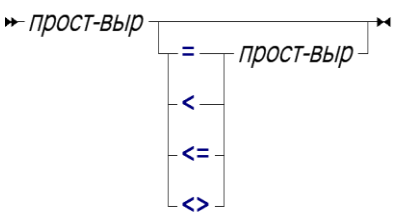
идент ::= *A..Z* [(*A..Z* | *0..9*)*]

имеет вид

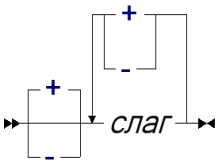


Пример: описанная выше грамматика для арифметических выражений задаётся совокупностью диаграмм:

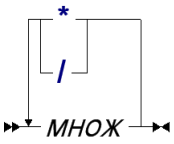
выр



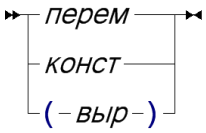
прост-выр



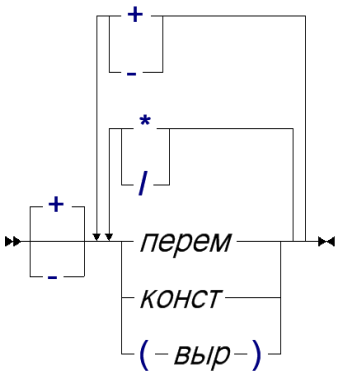
слаг



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*:



7.2.3 Устойчивость синтаксиса

Формальное описание синтаксиса и однозначность разбора несомненно важны для автоматической обработки текста программы. Однако, не менее важно, чтобы текст программы воспринимался человеком[6]. 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 С:

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

Очевидно, что здесь имелось ввиду, 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 и, соответственно, цикл выполнит N-1 «холостую» итерацию, после чего пересылка будет выполнена лишь

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

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
```

Ещё один пример на языке С - синтаксически корректный фрагмент:

```
float x;
for (float x=0.0; x<=1,2; f=+0.1)
    s = + f(x);
```

Здесь первая проблема заключается в «случайной» запятой в условии `x<=1,2`, хотя, очевидно, подразумевалось `x<=1.2`. Синтаксис последовательного выражения языка С делает это условие эквивалентным `x<=2`, что приведёт к тому, что цикл будет выполнен лишних 8 раз. Вторая ошибка связана с устаревшей формой присваивания =+, означающего «увеличить на». «Случайный» пробел после равенства делает равенство обычной пересылой, а + - унарной операцией. В результате оператор стал эквивалентным `s=f(x)`. Новая форма совмещённого присваивания += не имеет этого недостатка.

Рассмотрим пример на языке С, 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 остаётся 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;
```

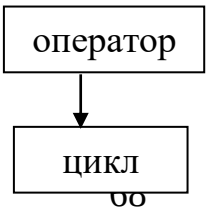
Таким образом, устойчивость синтаксиса языка также служит для раннего обнаружения ошибок: гораздо дешевле и безопаснее выявить ошибку на этапе синтаксического анализа, чем откладывать это на неопределённый срок с непредсказуемыми последствиями.

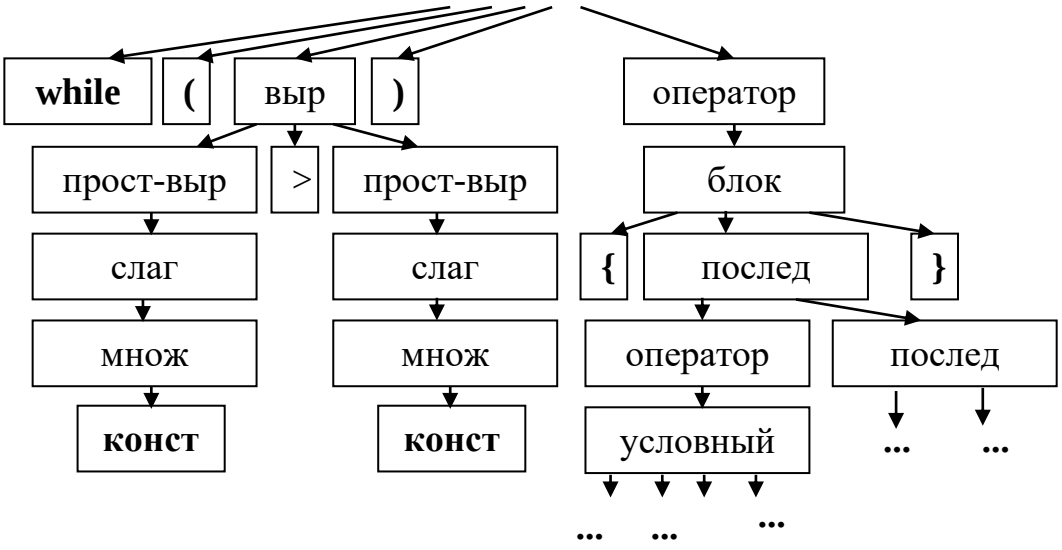
7.3 АБСТРАКТНЫЙ СИНТАКСИС

Дерево разбора программы может оказаться избыточно детальным для последущей 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, 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 трёх 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 означает синтаксическую категорию, для каждой альтернативы понятия указан её 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*)

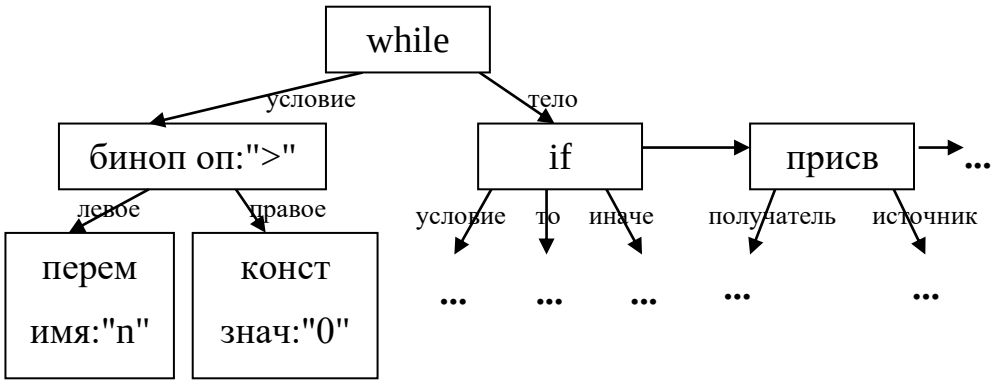
| (присв получатель:перем источник:выр)

выр ::= (перем имя:строка)

| (конст знач:число)

| (биноп 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:(биноп оп:">" левое:(перем имя:"n") правое:(конст знач:"n"))

тело:{(if

условие:(биноп оп:">"

```
        левое:(перем имя:"n")
        правое:(конст знач:"0")))
    то:{(присв
        получатель:(
        перем имя:"y"
        источник:(бинор оп:"*"
            левое:(перем имя:"y")
            правое:(перем имя:"y"))))}
    иначе:{})
(присв
        получатель:(
        перем имя:"x"
        источник:(бинор оп:"*"
            левое:(перем имя:"x")
            правое:( перем имя:"x")))
(присв
        получатель:(
        перем имя:"n"
        источник:(бинор 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.

7.4 КОНТЕКСТНО-ЗАВИСИМЫЙ АНАЛИЗ

Некоторые свойства синтаксиса языка зависят от контекста и не могут быть описаны контекстно-свободными грамматиками. Но даже в тех случаях, когда такое описание возможно, оно может оказаться неоправданно сложно. 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 затрудняет понимание, да и с точки зрения реализации может оказаться менее эффективной. Поэтому в подобных

случаях обычно ограничения оставляют в виде дополнительных условий к грамматике, предполагая, что они будут проверяться дополнительным проходом по дереву разбора.

Отдельного прохода требует *идентификация объектов*, то есть 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 [АхоУльман].

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.

Наиболее существенную часть контекстно-зависимого анализа составляет *статический анализ типов*, то есть определение (вывод) типов объектов и выражений и проверка типовой правильности. Далее мы ещё обсудим понятие системы типов в языках программирования[7].

[7] 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.

7.5 СЕМАНТИКА

Теперь, когда разобрана структура программы и установлены все связи между объектами и их типы, мы можем приступать к семантике, то есть определению того, что делает данная программа. 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. 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 нём разобраться или не хочет тратить на это своё время, то он вообще остаётся 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.

7.5.1 Денотационная семантика

Целью *денотационной семантики* является определение для каждой синтаксической категории так называемой *семантической функции*, которая сопоставляет каждой 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 имён 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 $Sem[s]$[8]. Для определения семантики мы воспользуемся структурной индукцией, то есть семантика сложной конструкции будет определяться через семантику её составных частей. Для вычисления выражений семантическая функция задаётся 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,

---

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

Func(>) (*x*,*y*) = если *x*>*y* то 1 иначе 0

Вычисление пустой последовательности операторов не изменяет состояние памяти:

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

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 вычислить значение источника присваивания, а новое состояние памяти отличается от данного только для переменной-получителя:

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 цикл возвращает данное состояние памяти. В противном случае, необходимо вычислить тело 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, то есть, 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.

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 будет 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 значение n на n/2. Как и ожидалось.

7.5.2 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,

   @ (биноп оп:">"

   левое: (перем имя:"n") правое:(конст знач: "0"))

   будет означать, что машина собирается вычислять применение бинарной операции,

   (биноп 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 дополнителный 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 точкая переставляется в конец выражения:

@((конст знач:*val*))

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

При вычислении переменной в результат помещается значение, извлекаемое по имени переменной из памяти[9], 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 её 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 сохранённые 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*) @)

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

=> (бинop 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...}

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 её 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 не остаётся выражений.

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)

| (бинор 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.

7.5.3 Аксиоматическая семантика

Аксиоматическая семантика не даёт 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 значение 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

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

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

Для доказательства утверждений аксиоматическая семантика задаёт 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:

$\{P[var \rightarrow expr]\}$ (присв получатель:*var* 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

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

что эквивалентно

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

и

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

Ещё одна аксиома имеется для пустой последовательности операторов

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

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

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

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

Для условного оператора:

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

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 нём, в отличие от остальных случаев, одно и то же условие P появляется как в левой, так и в правой части. По-существу, необходимо отыскать *инвариант цикла*, то есть 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 определённых классов программ инвариант можно построить.

Для определения инварианта в рассматриваемом примере введём 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* предложим 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, ещё 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. $Inv \wedge n \leq 0 \Rightarrow y = X^N \wedge n = 0$
3. $\{Inv \wedge n > 0\}$ BODY $\{ Inv\}$

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

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

откуда получаем $y = X^N$.

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

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

Оно разбивается на два случая:

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

и

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

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

$$N[2^{k+1}] = b_k\ 2^k + \sum_{i=0}^{k-1} b_i\ 2^i = \left(\frac{N}{2^k}[2]\right)2^k + N[2^k] = (n[2])2^k + N[2^k]$$

Если n-чётно, т.е. $n[2]=0$, то

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

Если же n-нечётно, 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 $\lceil log\ N \rceil$.

7.6 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 чём 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 идёт о субъективном восприятии, то «правильный стиль» закрепляется корпоративными стандартами оформления программ. Наличие общего стиля особенно важно, поскольку

одну и ту же программу в течении её 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 распространённых требований к стилю.

7.6.1 «Лесенка»

Текст должен располагаться так, чтобы выявлять синтаксическую структуру программы. Это достигается за счёт 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||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; }
```

Стандартным оправданием для такой записи является то, что "теперь всё перед глазами". Иногда добавляют ещё и заботу об эффективности: мол, в таком виде программа занимает меньше места[10] и транслятору придётся меньше считывать и обрабатывать ненужной информации. Однако, если теперь попытаться выяснить к какому `if` относится `else` в предпоследней строке, то придётся потратить несколько секунд, для того, чтобы дать уверенный ответ. И это для программы из четырёх 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:

---

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

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

Напомним, что поскольку речь идёт 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, "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.

7.6.2 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 нулём всех глобальных целочисленных переменных. Поэтому, чтобы избежать излишних действий по инициализации переменной, он положился на умолчание. Всё 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, придётся 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.

7.6.3 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 подчёркнуто, 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` програмист 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 согласовано с именами тех объектов, которые они индексируют. Общее неформальное правило можно сформулировать так: "длина идентификатора пропорциональна размеру области его действия".

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

7.6.4 Комментарии

Хорошо структурированная программа с мнемоничными именами во многом самодокументирована. Однако, далеко не всегда всё, что хочется и нужно сказать про программу, удаётся 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» - ещё не вполне. Обратная ситуация у того, кто читает программу. И положение читателя в определённом смысле сложнее,

поскольку почти любая программа делает больше, чем от неё 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 задаёт условие корректности, которое можно было упустить при беглом прочтении.

7.7 ПРАГМАТИКА

Прагматика языка – его соотвествие поставленным целям. Под этим в зависимости от контекста понимаются довольно разные вещи. Так,

некоторые языки создавались для решения определённого класса задач. Например, язык Кобол разрабатывался для создания обработки экономической информации, Фортран – для реализации научных расчётов, 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, подчёркивающими 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 всё 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`. Четвёртый 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.

7.8 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. Причём 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 живёт достаточно долго, то в него включаются дополнительные возможности, отражающие новые области применения, либо популярные в текущий момент концепции. Причём причины, повлёкшие 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 нём 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 всё 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 всё как есть.

Язык С много заимствовал, в частности, от языков Фортран и Алгол-60. В свою очередь, он лёг 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 С. 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 С лишь стиль большинства синтаксических конструкций, в расчёте на то, что это облегчит освоение языка. Конечно, возможность использования накопленных библиотек должна быть обеспечена, но всё новое программное обеспечение должно разрабатываться на новом языке.

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

8 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 решениий, 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 называемы *макросы*, то есть 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);
}
```

Всё, 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 C[11].

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

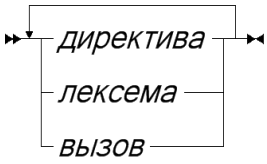
8.1 СИНТАКСИС

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 C, то есть помимо понятий идентификатора, строки, числа и т.п. в нём имеется ещё 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 её ¶.

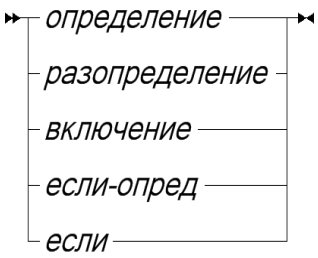
Весь текст состоит директив, вызовов и отдельных лексем. Все директивы располагаются на отдельных строках и начинаются с символа #. Если директива очень длинная, то её можно разбить на несколько строк, каждая из которых, кроме последней, завершается символом \.

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

текст :



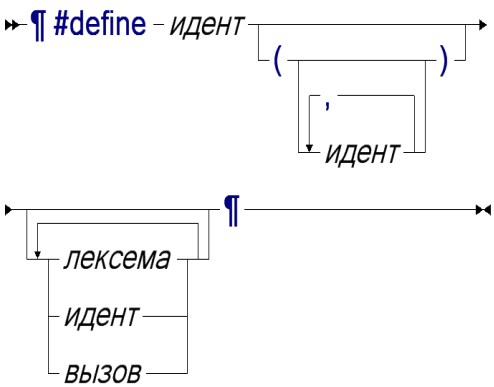
директива:



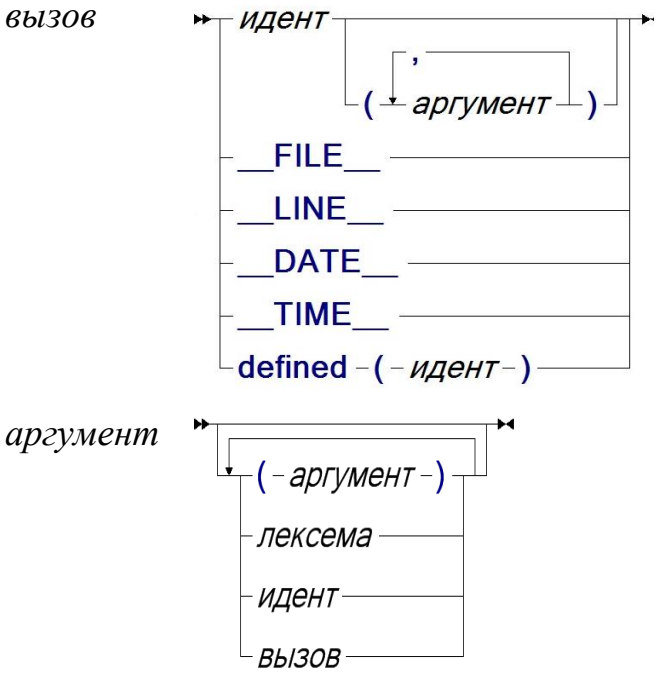
8.2 МАКРОСЫ И ВЫЗОВЫ

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

определение:



Макросы без параметров называются *макро-переменными*. Вызов начинается с идентификатора определённого макроса, за которым могут идти параметры через запятую.



Заметим, что слово "определённого" 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 ещё не определён, 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 определён просто как последовательность лексем, то препроцессор при определённом 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 приём:

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

даст в результате

```
a , b
```

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

Есть несколько предопределённых псевдо-макросов 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 определён макрос с данным *именем*. Полезность этого псевдо-макроса станет понятна ниже, когда мы будем обсуждать условную компиляцию.

Следущий 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
```

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 распространённое 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 констаные 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 С в некотрых 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 даёт 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 приведёт 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 отличии 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 всё, начиная с этой скобки, попадёт в тело макроса[12]. 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 С-программе, но понять и найти их бывает очень нелегко, поскольку с точки зрения препроцессора всё прошло нормально.

Ещё один типичный пример неправильного использования препроцессора, но уже не вызывающий и синтаксических ошибок. Пусть определён 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))
```

Ещё большие неприятности может вызвать то, что препроцессор не согласован с языком С 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 С, 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` определён как макрос, реально будет выполняться следующий текст:

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

[12] Типичный пример неустойчивого синтаксиса!

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. Ещё 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. Расмотрим, 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 всё, 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 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 мы не будем углубляться в их изучение.

Дублирование кода препроцессором может привести к тому, что программист потеряет контроль над размером получаемого кода. Определим, например, функцию Фибонначи с помощью макросов. Уже зная, что макрос в языке С 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);
```

приведёт 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 даёт возможность транслятору обнаружить, что всё 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[13]. 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.

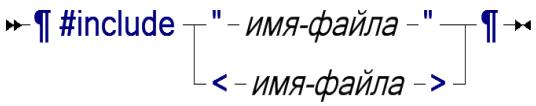
Директива-разопределение имеет следующий синтаксис

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

8.3 ВКЛЮЧЕНИЕ ФАЙЛОВ

Директива-включение задаётся следующим синтаксисом

включение:



и предназначена для того, чтобы подставить в текущее место содержимое указанного файла, которое затем обрабатывается препроцессором. В этом

[13] В общем случае вычисление чисел Фибонначи в таком виде требует экспоненциального количества сложений, а после такой оптимизации - линейного.

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 её файловой системой, в которой работает препроцессор; для MSDOS, Unix и OS2200 они могут быть устроены по-разному. Это может сделать программу не переносимой из одной обстановки в другую. Если имя файла указано в угловых скобках "<" и ">", то файл ищется в системных директориях, которые указываются либо в конфигурации системы программирования, либо параметрами при запуске препроцессора. В противном случае, если имя файла заключено в кавычки, то поиск осуществляется относительно расположения текущего обрабатываемого файла. Если файл найти не удаётся, 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"
```

8.4 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 С допускал лишь очень простые условия, заключающиеся в определённости макросов. Обычно для этой цели используются специальные макро-переменные, трактуемые как логические, причём директива

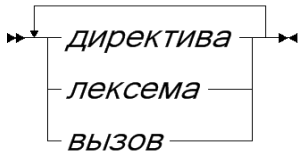
```
#define X
```

присваивает такой переменной истинное значение, а директива

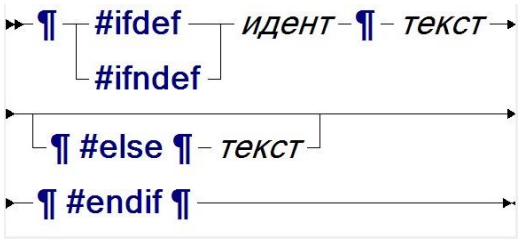
```
#undef X
```

- 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 определён (не определён) 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, объёма 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 макро-переменную, 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 путём заведения нескольких макро-переменных.

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

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

Макро-переменную можно задать и специальным параметром, передаваемым препроцессору при запуске. Тогда отпадает необходимость менять исходные тексты и обеспечивать согласованность определения макро-переменных в разных файлах.

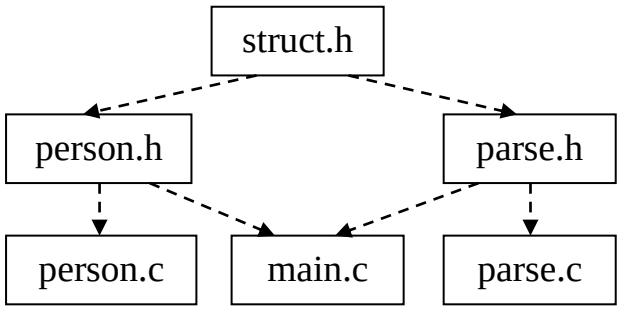
Ещё одним типичным использованием условной трансляции является решение проблемы повторного включения одного и того же файла. Рассмотрим следующую ситуацию. Пусть мы определили два модуля, каждый из которых определяет некоторый набор функций и расположен в своём файле, скажем, `parse.c` и `person.c`. 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 найдёт 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 приём. В начало любого включаемого файла, для которого следует избежать повторного включения, вставляется директива `#ifndef` с уникальным, предусмотренным специально для этого файла именем, традиционно имеющий 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 C с некоторыми

ограничениями[14], 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 C 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 псевдо-макроса `defined` делает избыточными условные вида `#ifdef` и `#ifndef`. Определённость макросов можно комбинировать как между собой, так и с другими условиями:

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

8.5 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 C две операции, которые

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

порождают новые лексемы. Эти операции могут использоваться лишь в теле макросов.

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

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

будет развёрнуто 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 C 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 макро-определения

```
#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 С-программ. Современные языки программирования, включая С++, имеют более развитые средства метапрограммирования.

9 ОБЪЕКТЫ И ТИПЫ

Изучение конструкций языка программирования начнём с понятия именования. Возможность дать имя некоторой конструкции, определённой 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.

9.1 ОБЛАСТИ ВИДИМОСТИ

Поскольку количество используемых программой имён может быть очень большим, то для избежания 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 определённом 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 задаёт *правило видимости*: имя, определённое в некотором блоке, может быть использовано в нём 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 её параметры.

Хотя правило видимости и позволяют переопределять во вложенном блоке имя, определённое 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 имён во вложенных блоках запрещается в некоторых языках программирования, например в C#.

Имя, определённое 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 VisualBasic, имеются *присоединяющие операторы*, которые являются блоками, делающими доступными в данной точке программы множество имён 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 С-подобном синтаксисе может быглядеть 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 использование имен, определённых во внешних библиотеках. Язык С 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 произойдёт, 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 остаётся никакой возможности использовать эти библиотеки одновременно. Поэтому создателям библиотек на языке С 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 подчёркивания, 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, определённого 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`, причём здесь перед точкой может быть только имя таблицы, а после - только имя столбца `select`, который должен быть определён в таблице `select`;
3. после ключевого слова `from` должно идти имя таблицы и не может появляться имя колонки и т.д.

9.2 ТИПЫ ДАННЫХ

Перейдём теперь к объектам, которые используются во время исполнения программы, то есть обрабатываемым данным. Доступ к ним осуществляется через имена, определённые 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 определённых в программе имён. Таким образом, существуют *анонимные объекты*, не имеющие собственного имени и 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 нобходимости 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 - как отобразить 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. 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, даёт 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 полтора десятка лет назад сведением к проблеме распознавания эквивалентности детерминированных магазинных автоматов.

9.2.1 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 распространённой перегруженной операцией является присваивание. Если семантика присваивания сводится к пересылке (копировании) данных из одного места в другое, то необходимо знать размер копируемой области, определяемой типами источника и получателя присваивания. Возможно также, что при присваивании происходит нетривиальные преобразования значений из одного типа в другой. Другим распространённым примером перегруженной операции является сложение (+), 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 идёт об определённых 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[15], 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 - вещественнное 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.

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

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

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 необязательными 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 дойдёт)

```
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 ведёт к написанию ненадёжных программ. Удобство и экономия усилий, которые даёт 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 придётся 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 определённой 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 контектста их использования. Это обеспечивает возможность проверки корректности применения всех операций.

9.2.2 Классификация типов

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

Прежде всего все типы можно разбить на *предопределённые*, 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*.

Перечеслимым (или *интегральным*) называются тип, множество значений которого отображается в диапазон целых чисел. Любой перечеслимый 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 предопределёнными типами, считая наиболее важным то, сколько места в памяти они занимают, мы начнём 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.

9.2.3 Логические

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 совпадается 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 несколько дополнительных команд. Поэтому обычно на

представление логического типа отводится по крайней мере один байт, а вопросы упаковки решаются отдельно.

9.2.4 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 заключёнными 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 начились 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 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 двухбайтой 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[16]. 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 четырё 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.

9.2.5 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.

---

[16] Заметим, что то, как символ изображается, является несомненно важной, но всё-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.

Основная проблема при моделировании целых чисел состоит в том, что их бесконечно много. В независмости о 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 представлюятся 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 начнём 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. Например, для того, чтобы сделать сложение, мы должны будем проверить одинаковы ли знаки числа, и, если 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 приведённые 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 рассчётов 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 системы разбивает двоичное представление на группы длины *n*. Например, в восьмеричном числе 275 цифра 5 обозначает три младших бита, 7 - следующие три, и т.д. Остаётся только запомнить, какие именно

комбинации из трёх бит кодирует каждая из цифр от 0 до 7, чтобы легко определить значение любого бита. Наибольшее распространение получили восьмеричная, шестнадцатеричная и двоичная системы.

9.2.6 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 "e", 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[17].

Вариации.

Как и целые числа, вещественные могут быть разного размера. Наиболее распространёнными способами представления вещественных чисел при заданном размере являются представления с фиксированной и с плавающей точкой. Начнём 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 ещё 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

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

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

Например, при n=8, p=2:

- 0 00000 00 = 0
- 1 00000 00 = -0
- 0 01111 10 = 8+4+2+1+1/2 = 15.5
- 1 00001 00 = -(1) = -1
- 1 11111 11 = -(16+8+4+2+1+1/2+1/4) = 31.75

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* даёт возможность задавать как существенно большие, так и существенно меньшие по абсолютной величине числа путём 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 её 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 ещё 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

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

Пример, *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 отличии 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, то есть существует (относительно небольшое) число ε>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 фиксированно 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 серьёзными 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.

9.2.7 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, позвляющий перечислить через запятую элементы множества, заключив всё это перечисление в квадратные скобки: `[]`, `['H','e','l','l','o']`, `[(i+1),(i-1)]`. При этом допускаются указывать диапазоны элементов `['A'..'Z', 'a'..'z']`, поскольку базовый тип должен быть перечеслимым. 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 "жёстком" 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 дизъюнкций. 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.

9.2.8 Перечисления

Тип перечисления предназначен для представления конечного множества значений, у каждого из которых есть своё имя. Единственное, что требуется от этих элементов - возможность сравнивать их на равенство и неравенство. Тип перечисления задаётся перечнем определяемых значений. Например, в языке Паскаль тип, представляющий цвета светофора, задаётся 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 C

```
#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 C считается, что все имена элементов перечисления видны в том блоке, в котором определён 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 ещё один тип для кодирования RGB представления цвета:

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

то при упоминание 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`.

9.2.9 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 операций является выборка поля, изображаемая точкой, за которой следует имя поля. То есть если переменная `p` имеет приведённый выше тип, то результатом выражения

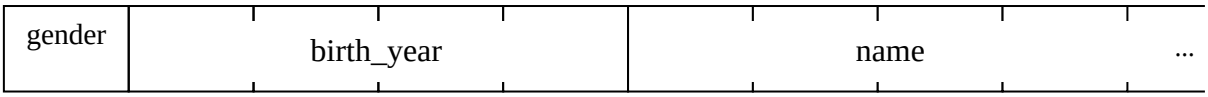
```
2020 - p.birth_year - 1
```

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

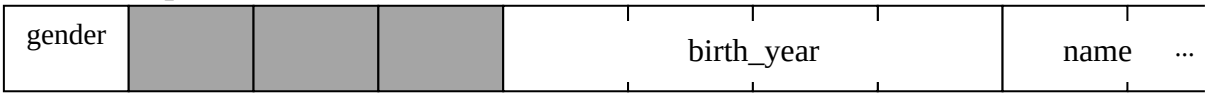
```
p.surname := 'Сидорова';
```

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 называется *выравниванием*.В приведённом выше примере перед полем `birth_year` необходимо зарезервировать 3 неиспользуемых байта.

Без выравнивания:



С выравниванием:



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 призвольном доступе к памяти, и поэтому такие языки, как Паскаль и С считают порядок фиксированным.

9.2.10 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 даёт этой же переменной новое имя, но уже определённого 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 неё 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 её 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.

9.2.11 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 всё 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.

9.2.12 Массивы

Тип массива служет для представления занумерованной последовательность 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[`*TInd*`]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*-ый элемент массива. 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 отчёт, 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.

$$i_{min} \le i \le i_{max}$$

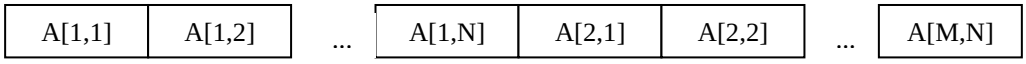
где i_{max} - максимальное значение в типе индекса. При нарушении этого условия исполнение программы должно прерваться с диагностикой *выхода за границы индекса*. Это приводит к дополнительным накладным расходам на доступ к элементам массива[18], но исключает одну из наиболее распространённых и опасных 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



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
```

Некоторые диалекты языка Паскаль вообще полагают эти типы эквивалентными. Однако, с семантической точки зрения во втором случае элементы массива являются самостоятельными объектами, которые можно

[18] 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.

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 всё равно не хватит. Кроме этого, контроль индексов в 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 придётся 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*.

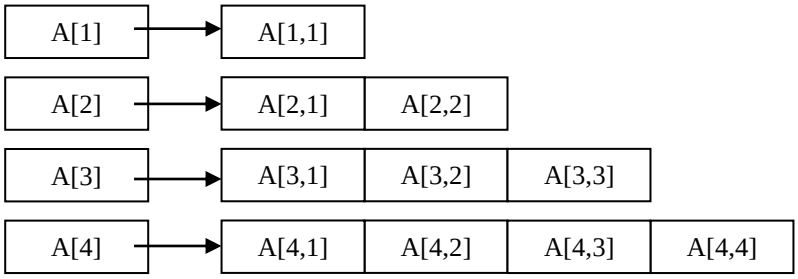
Пусть помимо добавления новых элементов в конец массива нам потребовалось вставлять или удалять элементы из середины массива. Понятно, что если настаивать на том, что все элементы массива расположены подряд, то удаление первого элемента потребует сдвига всего содержимого массива. В этом случае могут оказаться полезными списочные или древовидные структуры, которые разбивают содержимое массива на части "разумной" длины. Однако, заплатить за это придётся 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 даёт 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 хранится элементы треугольной матрицы. Если размер исходной матрицы равен N, то общее количество элементов в треугольной матрицы равно N*(N+1)/2.

```
var B : array [1.. N*(N+1) div 2];
```

Если элементы исходной матрицы расположены в нём 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
```

то есть

- 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 идёт 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 надёжнее 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.
- Траслятор 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) [???]. Язык предоставляет следующие возможности:

- богатый набор операций на 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 последовательного 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*, заданого массивом коэффициентов A:

```
/+ (A * (x   ⍳(n+1)))
```

где стандартная операция ⍳ порождает вектор целых чисел заданной длины, начиная с 0. Тогда

/+ (A * (x (0 1 2 … n)))

/+ (A * (x^0 x^1 x^2 … x^n))

/+ (A_0*x^0 A_1*x^1 A_2*x^2 … 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 её 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 вернёмся позже.

9.2.13 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 даёт возможность реализовать её 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 жёстко 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 решётки #:

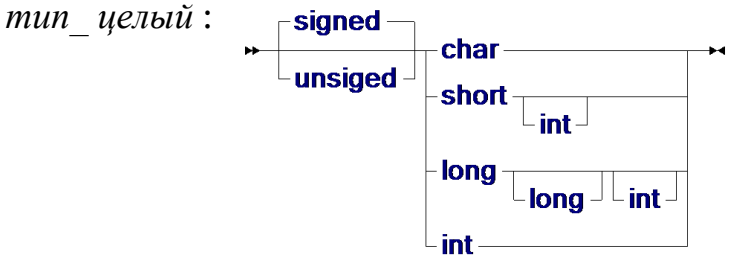
```
''''
'A8#65#56#0#12'
```

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.

9.3 ТИПЫ ДАННЫХ В ЯЗЫКЕ С

9.3.1 Целые и символы

В языке С имеется несколько целочисленных типов, отличающихся размером и тем, допускают ли они отрицательные значения. Синтаксис целочисленного типа задаётся следующей диаграммой:



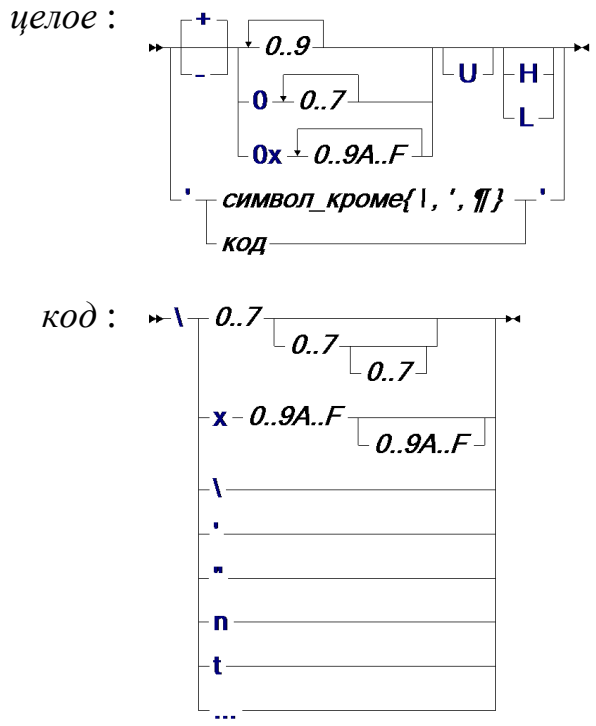
Какие имеено размеры соответствуют типам, в значительной степени зависит от конкретной реализации, что потенциально может сказаться на переносимости программ. Следующая таблица определяет минимальные размеры для разных типов в предположении, что размер байта равен 8 бит:

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 задаётся следующей синтаксической диаграммой:



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 распространённых спецсимволов - перевода строки, табуляции и т.п.

Такой подход делает возможной символьную арифметику. Например, i-ая буква латинского алфавита может быть найдена как

```
'A' + i - 1
```

что, конечно, компактнее, чем аналогичное выражение в Паскаль

```
chr(ord('A') + i - 1)
```

Однако, это же делает легальными и слабо осмысленные выражения типа

```
('A' + 'B'*'2') / '3'
```

9.3.2 Логические

В языке С нет логического типа в явном виде. Вместо этого язык считает, что все значения, в представлении которых есть ненулевых 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 чём мы ещё поговорим далее.

Примеры логических выражений:

```
!1 || 'A' && 0x12L
```

истинно, поскольку истино `!1` и `&&` имеет больший приоритет, чем `||`, а

```
'\0' || ('A' == 'B')
```

– ложно, поскольку и `'\0'`, и `'A'=='B'` - ложны.

9.3.3 Битовые шкалы

Вместо теоретико-множественных операций язык C 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, причём `y>=x`, то

Паскаль	C
`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[19]:

```
#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 используются 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

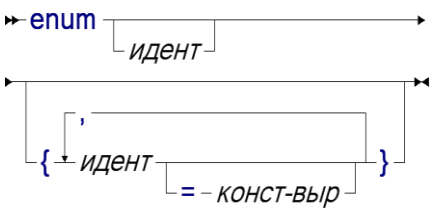
1 0000100 >> 2 = **1 11**00001 = -128 + 64 + 32 + 1 = -31 = -124/4

[19] Конечно, правильнее было бы ввести для этого соответствующий тип перечисления, а не пользоваться директивами препроцессора.

9.3.4 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

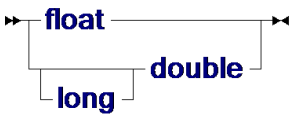
```
(Flag_Bald - Tue) | Green
```

хотя смысла в нём нет, и скорее всего такое выражение ошибочно.

9.3.5 Вещественные числа

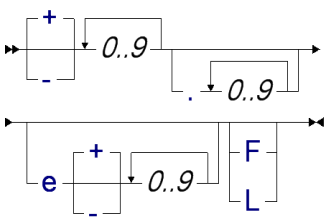
Язык С предоставляет три вещественных типа: 32-разрядный `float`, 64-разрядный `double` и `long double`, определяемых стандартом IEEE 754, о котором говорилось выше:

тип_вещественный:



Синтаксис вещественных констант задаётся 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 четырёхбайтным (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

9.3.6 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[20]. Подробности можно узнать в описании стандарта языка С.

Для явного преобразования типов результирующий тип указывается в скобках перед выражением, вычисляющим исходное значение. Например, при вычислении

```
(int) (3 + 0.5)
```

сначала 3 приведётся 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.

9.3.7 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

[20] А может быть и отрицательным, если в данной реализации `int` совпадает с `short`.

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 совпадается с адресом ни одного описанного в програме или созданного в процессе исполненния 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;
```

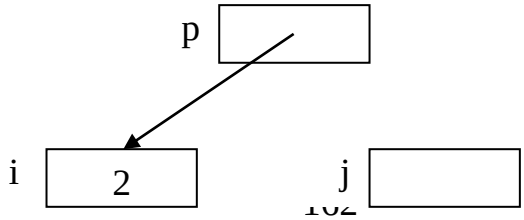
Первые две строки описывают три переменные: две целого типа и одну типа указатель на целое. Оператор

```
p = &i;
```

берёт 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 ведёт стрелка.

Следующий оператор

```
j = *p + 1;
```

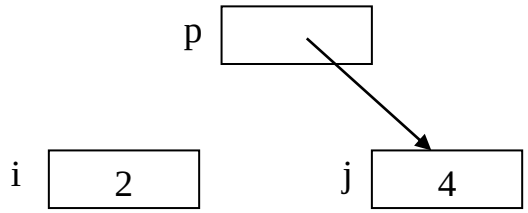
извлекает из ячейки `i` через указатель `p` значение 2, прибавляет к ней 1 и помещает результат 3 в ячейку `j`. Далее,

```
p = &j;
```

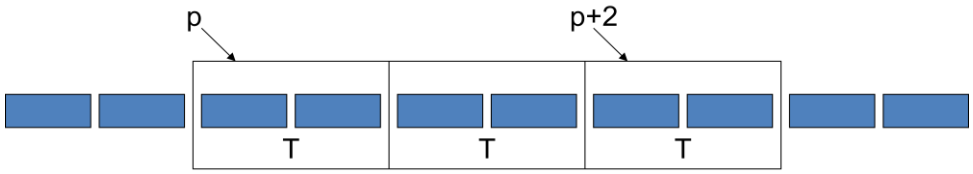
«перекидывает» стрелку из `p` на ячейку `j`: с этого места `*p` будет означать уже не `i`, а `j`. Наконец, оператор

```
*p = *p +1;
```

увеличивает значение переменной `j` на единицу, приводя к следующему состоянию пямяти:



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 даёт 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, то есть 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.

9.3.8 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 всём 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]`[21].

Язык С 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} };
```

9.3.9 Строки

Строки в языке С представляются указателем на первый символ строки. Более того, любой указатель на символ является в языке С 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 её определения необходимо "пробежаться" по строке, что делает эту операцию весьма неэффективной. Достоинство такого подхода

[21] Хотя, казалось бы, можно было заметить, что переменная A описана именно как массив, а не просто указатель, и для неё выполнять контроль индексов.

состоит в том, что нет явных ограничений на длину строки, в отличии, скажем, от Паскаль.

Не следует путать строки в языке С 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), заключённой 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";
```

определит длину массива равной 6. Строки, в отличии 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 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 её 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[22]:

```
char * PasCopy(char *source, int i, int l)
{
  char *dest = (unsigned char *)malloc(l+1);
  char *d = dest;
  char *s = &(source[i]);
```

[22] Этот пример демонстрирует в основном то, каким образом можно можно манипулировать указателями. Подробности, касающиеся управляющих структур и динамического размещения памяти (malloc) будут рассмотрены далее.

```
  while ((*d++ = *s++) && l--)
    ;
  d[-1] = '\0';
  return dest;
}
```

Также как и для массивов, язык C 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"`.

9.3.10 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 выдаёт указатель на его начало. Сама процедура выделения не знает, сколько значений и какого типа будут размещены в отведённой памяти, поскольку её 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 копирования одного отрезка памяти в другой, для чего требуется знать только указатели на начала отрезков и размер копируемого отрезка. В языке C такие указатели описываются с помощью псевдо-типа 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 надёжности. 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 надёжности и переносимости программ требуется передать подобной универсальной процедуре знание о типе аргументов. Для этой цели в языке С используется псевдо-функция `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 определён статически, вызов `sizeof` является константным выражением и заменяется на конкретное значение во время трансляции. Таким образом, более надёжным 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 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

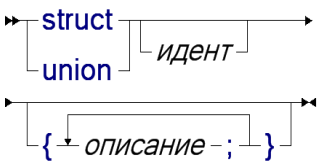
```
float f = 1.0;
int x = * (int *) (&f);
```

"взглянуть" на представление вещественного числа как на представление целого. Переменная `x` в результате может получить в зависимости от реализации, например, значение 1065353216.

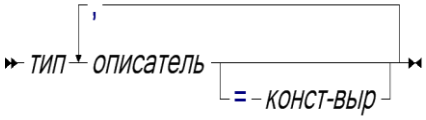
9.3.11 Описания, структуры и объединения

Тип структуры и объединения в языке С имеют почти одинаковый синтаксис, который задаёт перечисление полей, описание которых в свою очередь практически совпадает с описанием переменных:

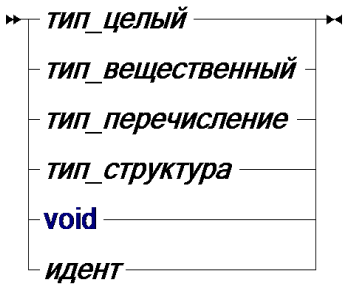
тип_структура:



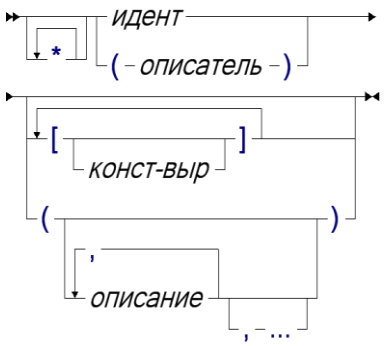
описание:



тип:



описатель:



Последняя альтернатива в диаграмме *тип* соответствует имени определяемого типа, а в диаграмме *описатель* - типу функции, которые обсуждаются ниже.

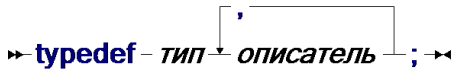
Синтаксическая категория *тип* не покрывает все возможные типы языка С: указатели, массивы и функции появляются в категории *описатель*. Это иногда приводит к неудобствам: например, не всякий тип можно непосредственно указать как аргумент операции `sizeof` и для этого приходится дополнительно описывать 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 даёт 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 задаёт 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 струкуры, 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 струкутры могут иметь поля одним и тем же именем. В этом смысле операция выборки является перегуруженной - 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 сокращённая форма: лексема `->`, за которой следует имя поля. То есть 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 ещё одной "дырой" в контроле типов. То есть 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 отличии 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`.

9.3.12 Присваивания

Целью присваивания является изменение значение 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;
```

произойдёт округление вещественного значения за счёт 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 выдаёт предупреждение.

Массивы, поскольку они являются константными указателями, присваивать нельзя, т.е.

```
int a[3], b[3];
a = b;
```

недопустимо.

С другой стороны, струкутры 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`, то есть 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 C есть несколько видов присваиваний, *совмещённых* с выполнением других операций. Так следующие два присваивания эквивалентны

```
x = x + 2;
x += 2;
```

Сокращённая 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 псевдо-операций && 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 несовмещённым 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 совмещённых 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 эквиваленен

```
x++;
x--;
```

Поскольку точно также 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. Вспомим, 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. То есть цикл эквивалентен

```
while (*p = *q)
{
   p++;
   q++;
}
```

причём 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, то есть нагляднее было бы записать его как `*q != '\0'`. Конечно, можно считать, что выбор между разными формами записи этого цикла - дело вкуса. Аргументами 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.

Вернёмся к операции инкремента. В языке C имеется как префиксная, так и постфиксная форма этой операции:

```
++x
x++
```

соответственно. Постфиксная форма выдаёт исходное значение переменной, а префиксная - новое, что семантически эквивалентно.

```
x += 1
(t = x, x += 1, t)
```

где `t` - вспомогательная переменая. В случае, если результат инкремента не используется, то две эти формы эквивалентны. То же справедливо и для операции декремента `--`.

9.3.13 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 воспринятно как

```
x = ((2 & y)  > 0 )
```

что ложно, причём x получит новое значение 0. Желаемый результат задаётся 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
```

причём все имеют разный смысл.

10 УПРАВЛЕНИЕ

Для того, чтобы действия в программе выполнялись в нужном порядке, используется управление. В простейшем случае, как, например, в машине Тьюринга, для каждого 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 задаёт частичный порядок на некотором подмножестве действий[23];
- *операторы*, целью которых является изменение состояние памяти. Последовательность выполнения операторов задаётся императивно, т.е. следующий за данным оператор задаётся 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 её многократно, возможно, меняя некоторые параметры;

[23] 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.

- *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.

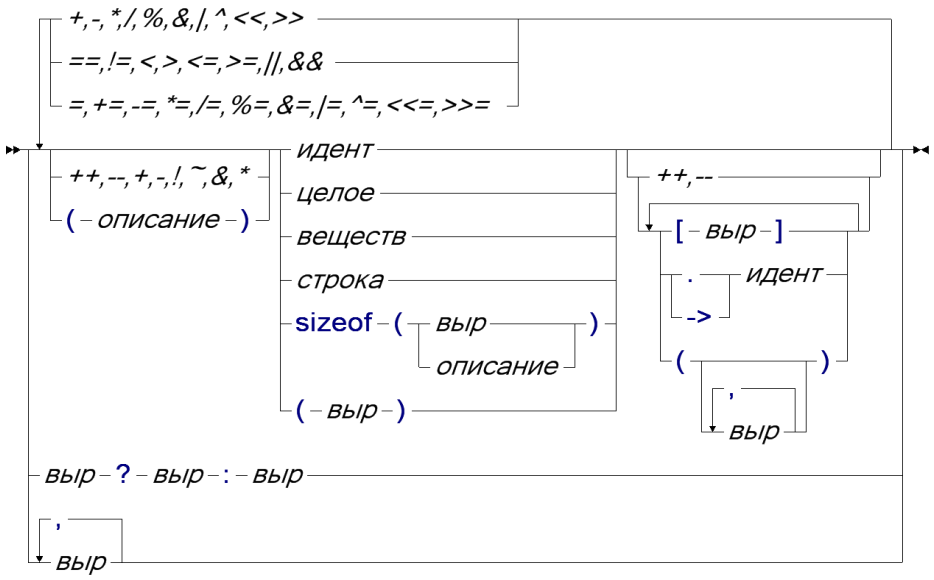
10.1 ВЫРАЖЕНИЯ

Выражения в языке С строятся из

- имён переменных;
- литеральных значений и имён 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 нём 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:

1	`() [] -> . ++ --`	Вызов функции, выбор поля или компоненты массива, постфиксные инкремент и декремент
2	`! ~ - ++ -- + * & (тип) sizeof`	Унарные, префиксные инкремент и декремент, приведение типа, sizeof
3	`* / %`	Умножение
4	`+ -`	Сложение
5	`<< >>`	Битовый сдвиг
6	`< <= > >=`	Сравнения
7	`== !=`	Равенство
8	`&`	Побитовое AND

9	^	Побитовое XOR
10	|	Побитовое OR
11	&&	Логическое AND
12	||	Логическое OR
13	? :	Условное выражение
14	= += -= *= /= %= &= |= ^= <<= >>=	Присваивания
15	,	Последовательное выполнение

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 использоваться 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 примения 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*

$e_1, e_2, \ldots, e_n$

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[24]. Например,

```
A[i,j] = i+j
```

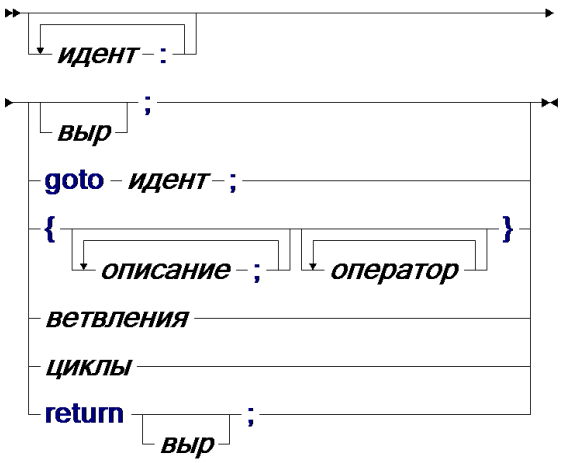
воспринимается как присваивание элементу двумерного массива, хотя на самом деле эквивалентно

```
A[j] = i+j
```

10.2 ОПЕРАТОРЫ

Управление на уровне операторов будем описывать методом раскрутки, то есть от простого к сложному. Набор управляющих конструкций в языке С весьма небольшой, как показывает следующая синтаксическая диаграмма:

оператор:



[24] Транслятор может выдать об этом предупреждение. А может и не выдать...

Начнём с первых двух альтернатив, которых, вообще говоря, вполне достаточно для реализации любой последовательности действий.

10.2.1 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 из только 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 ещё один пустой оператор.

10.2.2 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 всё, 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 введём вычисляемые метки для описания семантики других управляющих конструкций, после чего внесём их в "чёрный 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, всё управление спрятано в массиве меток `next`. Младший бит индекса этого массива соответствует положительности *n*, а второй - нечётности *n*. Кроме этого, программа была оптимизирована содержательно на основе следующих наблюдений:

1. если *n* положительно и нечётно, то после вычитания 1 оно обязательно будет чётно;
2. если *n* положительно и чётно, 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 её совершенно невозможно. Для каждого помеченного 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 ещё более серъёзной проблемой: становится невозможным задать семантику программы путём композиции, как мы это делали при описании денотационной и аксиоматической семантики.

Переменные-метки и переходы по вычисляемым меткам не является изобретением языка GNU C. Ещё в языке Фортран были сходные конструкции, называемые переключателями. Приведём 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 всё-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)
   {
```

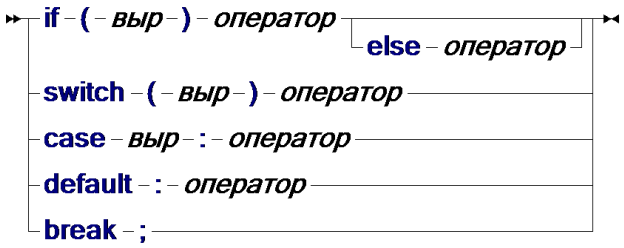
```
        int x = 2;
        l: printf("%d", x*i);
    }
}
```

что приводит к непредсказуемым последствиям. Язык Паскаль требует объявления меток, но они также относятся к функциям и не согласуются с блочной структурой.

10.2.3 Ветвления

В языке C имеется две конструкции, реализующие выбор одной из ветвей в зависимости от значения выражения - условный `if` и переключатель `switch`. Их синтакис задаётся 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 передаётся 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:

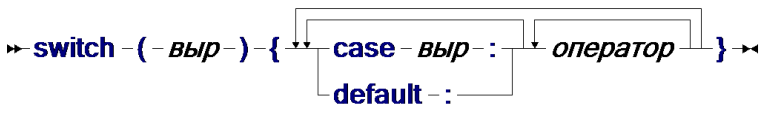
- Значение выражения в переключателе (`x%6` данном примере) вычисляется один раз;
- Выбор метки может быть реализован разными способами:
    1. в виде массива меток (как в данном примере), если известен их диапазон и он достаточно небольшой;
    2. двоичным поиском (дихотомией), если альтернатив достаточно 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 перейдёт к `x+=1;`. На практике это свойство переключателя является источником ошибок, поскольку `break;` чаще всего отстуствует не намеренно. Конечно, можно привести примеры, когда такое поведение позволяет либо не дублировать код, либо избежать использования `goto`, но в целом вреда больше, чем пользы.

В общем, переключатель в языке С недалеко ушёл от перехода по вычисляемой метке и позволяет писать плохо структурировнные 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 переключаетелях 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#:



что, видимо, можно рекомендовать и для использования в языке С.

Вариации

Условые 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 трёх 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.[25] 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 A.

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
```

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

[25] В языке Фортран метки обозначаются числами, а не идентификаторами.

используется перечисление `enum`, то при добавлении нового элемента к этому типу придётся исправлять и все места, где он используется.

Язык C требует, чтобы метки переключателя были *целого типа* (включая, естественно, тип `char` и `enum`), но не допускает строки. В том же Паскаль 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 подойдёт 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 C, то рассмотренный выше фрагмент программы двоичного поиска можно было бы реализовать следующим образом:

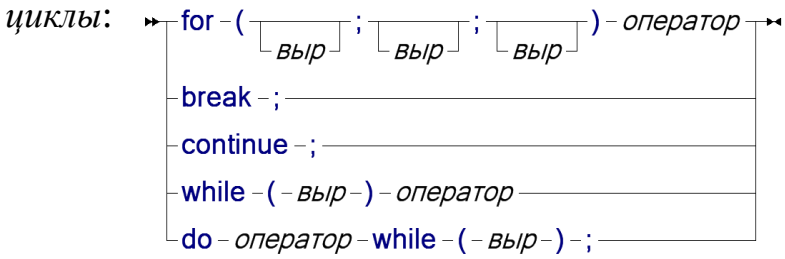
```
switch (1)
{
    case A[m]==x : break; // нашли!
    case A[m]>x  : high = m-1; break;
    case A[m]<x  : low = m+1; break;
}
```

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.

10.2.4 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:



Семантику циклов начнём 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* отусутствует, то оно полагается тождественно истинным, то есть условный оператор можно опустить. В этом случае тело цикла *body* будет выполняться бесконечно много раз, если только в нём 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 значания 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 её 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 равным `N-1`.

Операторы `break;` и `continue;`, называемые *структурными переходами*, означают передачу управления на `lDone` и `lStep`, соответственно. То есть `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 Паскаль цикл

```
repeat ... until condition
```

будет выполняться до тех пор, пока `x` не станет меньше 0.01. По каким-то причинам в Паскаль 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 C, поскольку условие завершения есть просто отрицание условия продолжения.

На практике очень часто встречается ситуация, когда надо некоторое действие повторить определённое *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 путём 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 ещё 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 возникаются 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 придётся 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 С 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 даёт 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 её 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 аналогичным цикле на языке C

```
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 Паскаль до появления в нём оператора `break;` для прерывания цикла использовался следующий приём: 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 приведёт к синтаксической ошибке, а значение переменной цикла после его завершения неопределено.

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 элемены в текущей строке, после чего управление перейдёт 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 счёт дополнительных накладных расходов, потери наглядности и надёжности, то есть 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 С такие ситуации относятся к тем редким случаям, когда оправдано применение оператора `goto`.

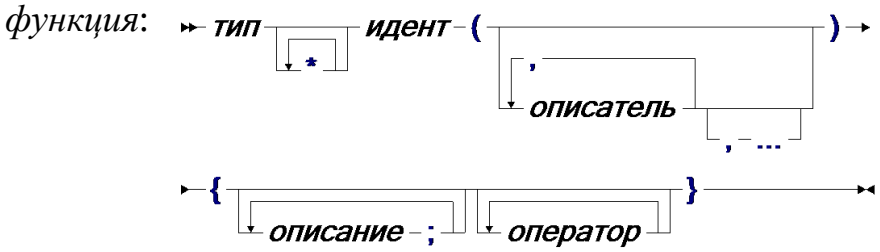
10.3 ПРОЦЕДУРЫ И ФУНКЦИИ

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, соотвественно, их вызов является оператором. Поскольку в С, 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`.

10.3.1 Описание функций

Синтаксис описания функции в языке С имеет следующий вид:



В нём отражается несколько, вообще говоря, самостоятельных аспектов:

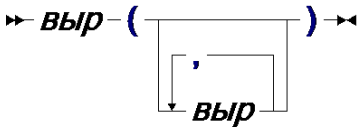
1. описание *типа функции*, которая включает тип результата, с которого начинается описание, а также типы и имена *формальных параметров*, задаваемые описателями. Если список формальных параметров завершается многоточием, то это означает, что функция может иметь дополнительные параметры;
2. *имя* функции;
3. *тело функции* - блок, определяющий последовательность выполняемых действий с указанием того, что является *результатом*.

10.3.2 Вызов функции

Функции в языке С 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.

Приведём 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 распространённого 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. Четвёртый вызов демонстрирует, что фактическим параметром также может быть вызов функции. Наконец, последняя строка - вызов функции `WriteLn` без параметров. Типичной ошибкой является попытка вызова функций без скобок:

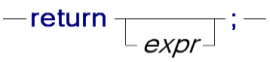
```
WriteLn;
```

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 игнорируюей 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:



Выражение должно пристуствовать тогда и только тогда, когда тип возвращаемого значения отличен от `void`. Кроме этого, оператор `return` завершает исполнение процедуры, то есть переходит к шагу 4. Точнее говоря, корректное исполнения функции, которая не является процедурой, обязано 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 всё необходимое окружение. В частности, в качестве фактических параметров передаётся 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 массиав `coef` равна `n`+1;
3. `power` - функция вычисления `x` в степени `n`.

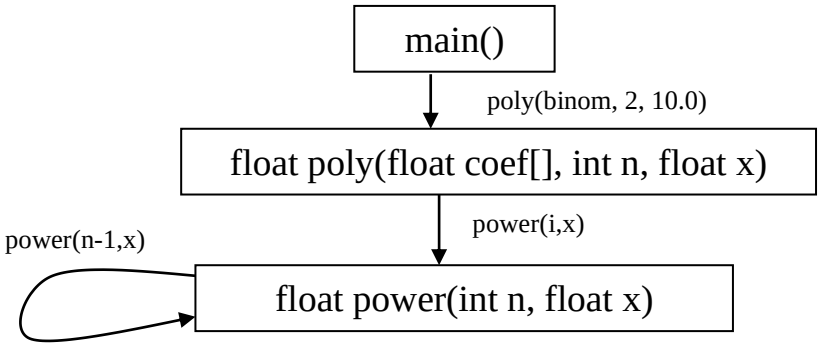
10.3.3 Рекурсия

Особенностью функции power в рассматриваемой программе является то, что при некоторых условиях она может вызывать сама себя, что соответствует рекуррентному математическому определению степенной функции:

$$x^n = \begin{cases} 1, & \text{если } n = 0 \\ x * x^{n-1}, & \text{иначе} \end{cases}$$

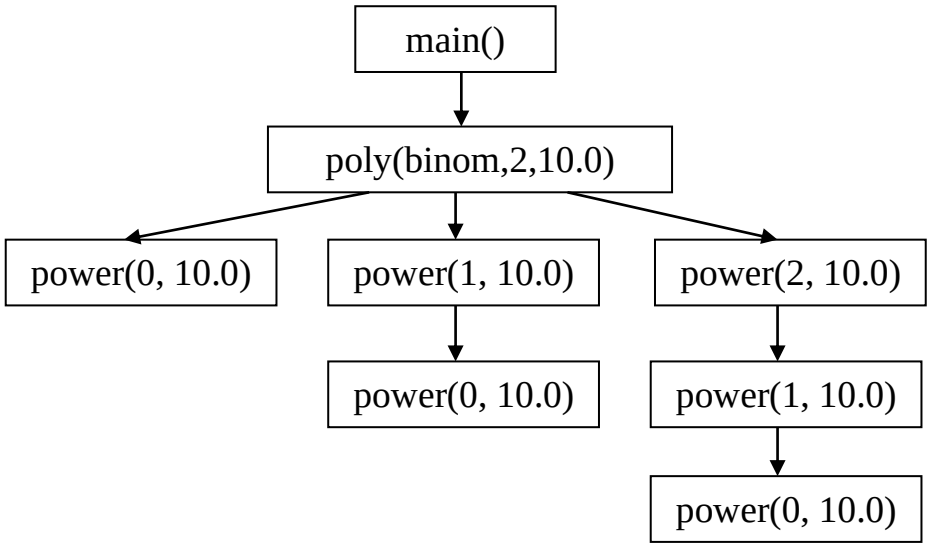
Такие функции называются *рекурсивыми*. Это определение может быть истолковано разными способами и требует более формального подхода. Рассмотрим *граф вызовов* программы. Вершинами этого графа являются функции, а дуга проводится от одной функции к другой, если в первой есть

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 привёл 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", то есть основанной на графе вызовов рекурсивности, то её легко определить, но только в случае, если вызываемые функции всегда указываются явно своими именеми. Иначе для каждого выражения, вычисляющего функцию, необходимо определить множество её 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 её вызов еще не закончился, то согласно описанному методу исполнения вызова функции создаются новые локальные объекты, когда старые ещё 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 её 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 дереву 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.

10.3.4 Реализация функций

Семантику функций мы будем описывать, используя *трансформационный* подход, постепенно преобразуя программу ко всё 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 добьёмся 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 создаётся перед вызовом процедуры и в него присваиваются значения, соответствующие 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 её тип.

На **четвёртом шаге** мы избавимся от параметров вообще. Заметим, что к этому моменту у каждой процедуры есть доступ только к одному фрейму. Заведём глобальную переменную `f`, которая будет хранить ссылку на текущий фрейм. Посколько 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, заведём 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 процедур.

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;
   float * 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 неё в ещё 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:

3. 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.
4. 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[26] рекурсивной, что ещё раз свидетельствует о достоинствах рекурсивных процедур, по крайней мере для опередлённого класса задач.

10.3.5 Хвостовая рекурсия

Преобразования не являются оптимизирующими, то есть 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 счёт 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 путём введения *накопительных* параметров. Рассмотрим этот приём на примере той же функции возведения в степень, которая уже была предварительно преобразована в процедуру:

---

[26] 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.

```
void power(int n, float x, float * result)
{
  if (n==0)
    *result = 1;
  else
  {
    float t;
    power(n-1,x,&t);
    *result = x * t;
  }
}
```

Заметим, что она не удовлетворяет требованию хвостовой рекурсии, поскольку, во-первых, после вызова `power` ещё выполняется присваивание результату, и, во-вторых, вызову передаётся ссылка на локальную переменную `t`, которая внутри вызова может быть изменена и на самом деле меняется присваиванием `*result`.

Заведём 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 нём 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;
}
```

10.3.6 Вложенные процедуры

Вернёмся к программе вычисления полинома. Можно заметить, что параметр `x`, передаваемый в вызове `power` в функции `poly` дальше передаётся без изменений в рекурсивные вызовы. В результате все поколения переменной `x` имеют одно и то же не изменяемое 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 всё время выполнения программы, и её смогут изменить и другие процедуры. Выход из этого положения дают *вложенные процедуры*, которые имеются во многих языках программирования, например, в Паскаль 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 типерь 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, пробежать 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 *свёртки*. Допустим, мы обнаружили, что некоторый

фрагмент кода нужно выполнить ещё 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, придётся 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.

10.3.7 Оптимизации

Если уж мы заговорили об оптимизации, то рассмотрим наш пример на основе анализа сложности вычислений. Использованные ниже методы достаточно характерны при оптимизации программ. Будем оценивать сложность в количестве выполненных операций умножения. В тексте она встречается дважды: один раз в функции `poly` и один - в `power`. Поскольку всё 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

$$T_{poly}(n) = \sum_{i=0}^{n-1}(1 + T_{power}(i))$$

$$T_{power}(n) = \begin{cases} 0, \text{если } n = 0 \\ 1 + T_{power}(n), \text{иначе} \end{cases}$$

Очевидно, что $T_{power}(n) = n$, и следовательно,

$$T_{poly}(n) = \sum_{i=0}^{n-1}(1 + i) = \frac{n(n+1)}{2}$$

Что касается ёмкостной сложности, то она определяется максимальной глубиной в дереве вызова, которая равна n. Преобразование функции `power` путём приведения её к хвостовой рекурсии не изменяет количество операций, но уменьшает ёмкостную сложность до константы.

Большинство умножений происходит в функции `power`. Следуя принципу оптимизации наиболее горячих точек, займёмся ею в первую очередь. Можно попытаться реализовать функцию `power` за счёт 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

$$T_{poly}(n) \leq \sum_{i=0}^{n-1}(1 + 2\log(i+1))$$

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

$$T_{poly}(n) \leq 2\, n \log n + 1$$

Теперь заметим, что на каждой итерации цикла в функции `poly` результат функции `power` отличается от предыдущего в *x* раз. Это даёт возможность вообще исключить функцию `power`, введя вместо неё 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

$$T_{poly}(n) = 2\,n.$$

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 всё это за нас сделает транслятор.

10.3.8 Функциональные значения

Как мы уже говорили, указатели на функции являются равноправными объектами в языке C. 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 C 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` задаёт 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 её 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 `op` преобразует код операции в указатель на реализующую её 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`. То есть вызывать мы её 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 дойдёт во 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 некорретным будет возвращение в качестве результата, либо присваивание в нелокальную переменную любой ссылки на локальный объект, будь то переменная, процедура или метка.

10.3.9 Подстановка параметров

В этом разделе мы рассмотрим несколько вариаций на тему подстановки параметров. В языке С 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 результатам. 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[27], 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,

[27] 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.

```
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 в им были присвоены начальные значения, не имеют никакого отношения к A и B.

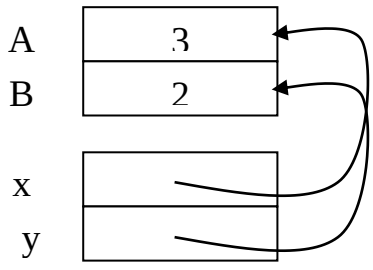
На первый взгляд получается, что при передаче параметров по значению единственным способом осуществить побочный эффект является изменение глобальных переменных. Нам необходимо, чтобы в ходе выполнения swap переменные x и y означали A и B соответственно, и любое присваивание, скажем, x означала присваивание A. Такой способ передачи параметров называется *по ссылке*. В языке C он может быть реализован следующим образом:

```
void  swap(float * x, float * y)
{
  *x += *y; *y = *x-*y; *x -= *y;
}
```

а при вызове по значению передавать указатель 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, причём последний способ является умолчательным. В языке Фортран параметры всегда передаются по ссылке, поскольку в нём типична передача в качестве параметров больших массивов[28]. При передаче по значению это потребовало бы больших накладных расходов как по памяти, так и по времени.

Вернёмся к примеру с функций `swap`. Рассмотрим случай, когда в качестве параметра передаётся 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, 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 приводит к труднообнаруживаемым ошибкам. Также оно может влиять на эффективность программ, поскольку транслятору приходится

[28] Именно массивов, а не ссылок на первый элемент массива, как в языке C.

предполагать, что параметр может указывать куда угодно, и, как следствие, ограничивать применение некоторых оптимизаций.

Избежать синонимичности позволяет передача 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 заначения 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 определённых условиях, которые в общем случае зависят от других параметров.

Рассмотрим, например, функцию, которая вычисляет квадрат либо второго, либо третьего аргумента в зависимости от некоторого условия, значение которого передаётся 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 вернёт ноль, однако это не так, поскольку согласно методу передачи параметров по значению сначала вычисляются все три значения фактических параметров, включая `1/x`, что приведёт к исключительной ситуации.

Отложить вычисление можно, если передавать не значение, а функцию, вычисляющее это значение. Такой способ называется передачей параметров *по необходимости*. Сразу скажем, что это имеет смысл только для так называемых *нестрогих* параметров, то есть тех, значения которых могут не потребоваться. В противном случае параметр называется *строгим*. В приведённом 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*[29].

Фундаментальным достоинством передачи параметров по необходимости является то, что такой вызов всегда выдаёт 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 ухушится, 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 её 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

[29] Слово thunk является субстандартной или диалектной совершенной формой глагола think (думать).

```
float power(float * x())
{
  return (*x)() * (*x)() * (*x)();
}
```

а вызов будет иметь вид

```
power3(t2_thunk)
```

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. То есть (в нотации языка С) 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 отличии от неё 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 траслятор 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 приведённом 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[30].

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));
```

Возможность передавать произвольное количество параметров не укладывается в ту схему реализации процедур, которую мы рассматривали

[30] 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. Приведённые в обсуждении примеры корректны только в предположении, что в фактическим параметре встречаются только глобальные переменные.

выше, поскольку неизвестен размер фрейма и типы передаваемых дополнительных параметров. Её можно развить, формируя фрейм не по описанию процедуры, а по конкретному вызову, что приведёт 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, определённых 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 параметр. 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 совмещён 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 назовём `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 её, 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 следующим образом. Во-первых, невозможно проверить, что количество формальных парметров соответствует формату. То есть, при вызове

```
my_printf("%f + %f = %f", x, y);
```

при обработке последнего %f произойдёт 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` приведёт 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 неё 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)
```

причём 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 C. 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 прямоуголиника и её 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 место, 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, все параметры должны стоять на соответствующих позициях. Так что в данном случае, если нужно задать прозрачность, то придётся указать и все остальные параметры.

Кардинальным решением является указание имён 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);
```

причём 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.

10.4 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[31], и возвращает значение выражения.

Имея две такие функции мы можем просто написать процедуру, реализующую *цикл "читать-вычислять-печатать"*, который по очереди вводит выражения и выдаёт 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);
}
```

[31] Для дальнейших рассуждений нам неважно, как именно устроен этот объект и как, используя его, получить значение переменной, входящей в выражение.

Всё 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)))
```

произойдёт деление 3/0. Чтобы наша программа вообще не прервалась, мы должны предусмотреть обработку исключительных 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 выдаём сообщение и передаём 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 С может не быть вложенных процедур. В таком случае нам придётся 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 позовляет 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 её 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 С 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[32] 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_buf env;
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;
```

[32] На самом деле setjmp является макросом, поскольку в противном случае она не могла бы изменить параметр, переданный по значению.

```
      case ERR_OVERFLOW :
        ...
    }
  }
  fputs("Пока.",stderr);
}
```

при вызове `setjmp` управление будет передано на альтернативу `case 0`, и если ничего не случится, то будет распечатано вычисленное значение выражения.

Функцию `eval_expr` вернём практически к её 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 вернёт 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 позоволило 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 тяжёлым 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 большинсве случаев позволяют избежать таких ситуаций. Так, в нашем пример 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.

11 РАСПРЕДЕЛЕНИЕ ПАМЯТИ

Все объекты программы можно классифицировать в зависимости от способа их создания и времени существования как глобальные, локальные (или автоматические) и динамические.

Глобальные объекты существуют всё 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 пямять, пока программа не закончится.

Автоматическими называются локальные объекты процедур и функций. Как мы уже обсуждали, они появляются только в при вызове процедуры, а после её завершения удаляются, причём 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 удалён 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 упрощённая: она отводит под стек массив фиксированной длины, не рассматривает случай переполнения стека, игнорирует выравнивание и т.п. Возможны и принципально 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 C предоставляется следующие функции для создания и удаления динамических объектов, определённые 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 выдаёт 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 её реализации. Например, можно организовать списки для каждого запрашиваемого размера (или по-крайней мере для наиболее распространённых значений). При этом, конечно, придётся решать проблему, когда имеется много свободных фрагментов одного размера, а запрашивается другой. Для избежания *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 займёт 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 счёт 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 объём 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 псевдо-процедурой

```
new(p)
```

которая "знает" о типе, на который указывает p. В языке С 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);
```

приведёт к ошибкам, поскольку при выделении памяти не учтён 0, завершающей строку source.

Каждый размещённый фрагмент памяти должен быть освобождён. Если указатель на выделенный участок памяти потерян, то теряется и возможность осводить 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.

Ещё 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 путём 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 путём 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 размещённый 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 счётчик ссылок, который увеличивается или уменьшается при присваивании указательных переменных. Когда же счётчик 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 всё целиком.

По многим причинам - наличия адресной арифметики, возможности некорректного приведения типов и др. - автоматическая сборка мусора принципиально невозможна в языке C. 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 надёжность 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 стукуры 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, предстваляющая лексему, в которой, если необходимо, помимо её кода указана дополнительная информация о значении или имени:

```
struct Token {
    enum TokenCode code;
    union {
        float value;
        char name[8];
    } choice;
};
```

Также будем считать, что определён тип `Lexer`, для которого реализована функция, вырабатывающая ссылку на очередную считанную лексему:

```
struct Token * get_token(Lexer lexer);
```

Поскольку нам потребуется «заглядывать» на шаг вперёд, 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 приведём 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` создаётся 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 выделяются по максимуму, хотя для вершины-числа её 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`, потребуется 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.

12 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 имён 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*, определённые 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`)[33] файла система поддержки исполнения выбирает свободный элемент в этой таблицы, заполняет его, устанавливая связь с физическим файлом, и выдаёт 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 связываются 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. Её 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`.

---

[33] 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.

Для минимизации количества обращений к операционной системе используется *буферизованный ввод-вывод*, типы и функции которого определёны 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 жёстком 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.

Определённая в `stdio.h` струкутра `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 всего 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 тяжёлая 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 C, 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 рассматривавшуюся 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 стандарные псевдо-процедуры `Write` и `WriteLn`:

```
WriteLn(f, x:6:2, ' + ', y:6:2, ' = ', x+y:7:2);
```

Эти псевдо-процедуры, в отличии 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 даёт 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.