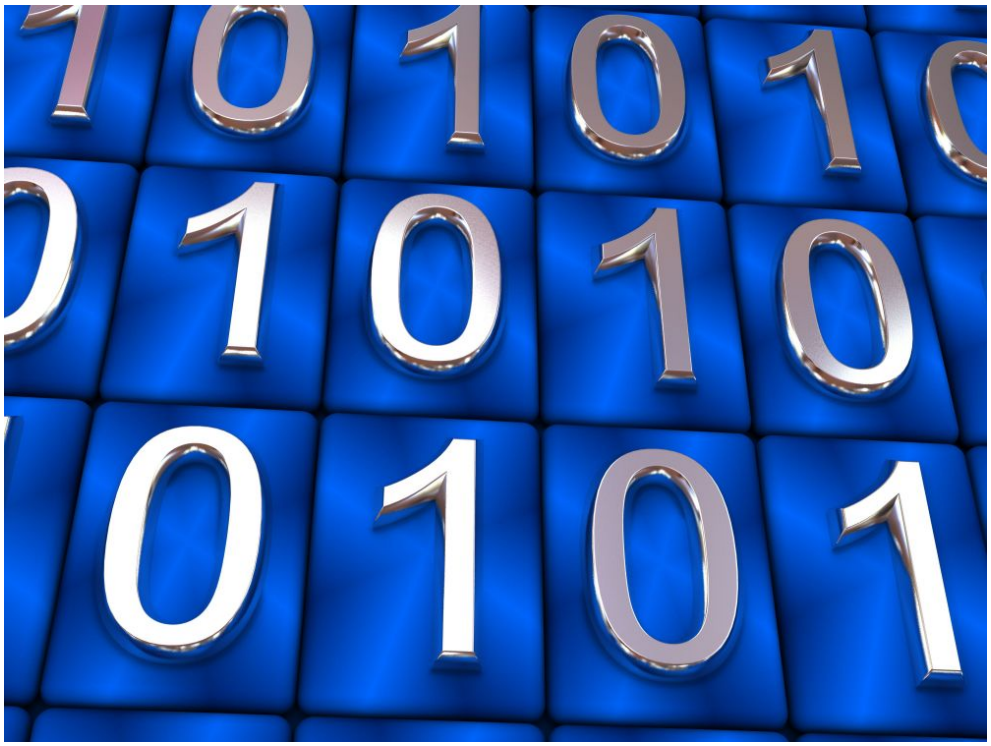
УНИВЕРСИТЕТ ИТМО

А.А. Тюгашев

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Часть II



Санкт-Петербург
2016

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

А.А. Тюгашев

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Учебное пособие. Часть II.

Санкт-Петербург

2016

А.А. Тюгашев. Основы программирования. Часть II. – СПб: Университет ИТМО, 2016. – 116 с.

Учебное пособие содержит теоретический материал и лабораторный практикум для изучения дисциплины «Основы программирования». Представлен панорамный взгляд на предметную область, с представлением не только традиционной императивной, но и функциональной, и логической парадигм программирования, исторической ретроспективы и связи с другими областями информатики. Приводится сравнение программирования на языках высокого и низкого уровней (ассемблер). Несмотря на обзорный характер, после прочтения и прохождения входящего в книгу лабораторного практикума студент будет способен писать программы средней сложности на языках C/C++. Книга содержит и специальные главы, посвященные жизненному циклу программных средств современной ИТ-индустрии, проблеме ошибок в программах и методах верификации программного обеспечения, стилю программирования.

Учебное пособие адресовано студентам, обучающимся в ИТМО на кафедре КОТ по направлению 09.03.02 «Информационные системы и технологии»; преподавателям, ведущим теоретические и лабораторные занятия по курсу «Основы программирования». В то же время издание может представлять интерес для школьников, студентов средних специальных заведений и широкого круга читателей, заинтересованных в освоении основ программирования.

Рекомендовано к печати Ученым советом факультета КТиУ 08.12.2015 г., протокол №10.

Университет ИТМО – ведущий вуз России в области информационных и

фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

Оглавление

Введение в функциональное программирование

Несмотря на богатую историю развития традиционных языков программирования, они несут на себе родовую печать фон-неймановской архитектуры. А по меткому высказыванию, приведенному в книге Л. Стерлинга и Э. Шапиро [14], «Образованному непредубежденному человеку, не знакомому с кухней проектирования компьютеров, машина фон Неймана кажется надуманным и причудливым устройством. Рассуждать в терминах ограниченного множества операций такой машины далеко не просто, и иногда такие рассуждения требуют предельных мыслительных усилий». При этом, как уже говорилось, язык оказывает существенное влияние на сам механизм мышления. Неудивительно, что недостаток был очевиден для выдающихся умов в области информатики. Вспомним, в частности, Джона Бэкуса — создателя Фортрана и его речь при вручении ему премии имени Алана Тьюринга — высшей научной награды в области информатики, аналога Нобелевской премии. «Традиционные языки программирования чудовищно разрастаются, но не становятся сильнее. Наследственные дефекты на всех базовых уровнях приводят к тому, что языки становятся одновременно “жирными” и “слабыми”: их примитивный стиль “одна операция в один момент времени” наследуется от единого для всех языков предка — архитектуры машины фон Неймана...», — сказал Бэкус. Само название тьюринговской лекции Бэкуса говорит о многом: «Можно ли освободить программиста от фон-неймановского стиля? Функциональный стиль и определяемая им алгебра программ».

С этими мыслями Бэкуса согласны Л. Стерлинг и Э. Шапиро: «Мы полагаем, что программирование может и должно быть интеллектуально благодарной деятельностью; что хороший язык программирования является мощным инструментом абстракций — инструментом, пригодным для организации, выражения, экспериментирования и даже общения в процессе интеллектуальной деятельности; что взгляд на программирование как на “кодирование” — рутинный, интеллектуально тривиальный, но очень трудоемкий и утомительный заключительный этап решения задачи при использовании компьютеров, — возможно, и является глубинной причиной явления, называемого кризисом программирования» [14].

Иными словами, главная мысль здесь заключается в том, чтобы не человека вынудить при создании программ мыслить в категориях машины, а компьютер заставить понимать язык с более высоким уровнем абстракции, удобный для формулировки и планирования решения задач человеком. Более того, в математике известны такие математические

формализации понятия алгоритма и модели вычислений, как рекурсивные функции, нормальные алгорифмы Маркова, лямбда-исчисление, не привязанные к понятиям состояния, шага программы и записи значения в ячейку, характерным для машины Тьюринга. Они могут служить мощным теоретическим фундаментом при создании языков программирования, которые можно с определенной натяжкой назвать нетрадиционными.

Функциональное программирование — одна из основных нетрадиционных парадигм (систем основополагающих взглядов, принципов) программирования, называемая также аппликативной. Причиной этого является то, что в ней вычислительный процесс сводится к применению функции к аргументу, а теоретическим базисом служит λ-исчисление (лямбда-исчисление). В качестве важного преимущества данной модели вычислений можно выделить, в частности, отсутствие так называемых побочных эффектов, что означает, что результат выполнения программы полностью определяется ею самой и исходными данными.

Хотя в определенном смысле предшественником Лиспа можно считать язык IPL (1956), пионером в этой области стал Джон Мак-Карти, который в 1958 году во время работы в Массачусетском технологическом институте разработал язык программирования LISP (от англ. LISt Processing — обработка списков). Это, вероятно, первый из языков, который основывался на серьезном теоретическом фундаменте и пытался поднять практику программирования до уровня концепций, а не опустить концепции до уровня существовавшей на момент создания языка практики [7]. Язык был описан Маккарти в статье «Рекурсивные функции над символьными выражениями и их вычисление с помощью машины».

В русскоязычной литературе используются различные варианты написания названия — LISP, Lisp, ЛИСП, мы в основном будем называть его Лисп. Лисп был изобретен как формализм для рассуждений об определенном типе логических выражений, называемых уравнениями рекурсии, как о модели вычислений.

Контрольные вопросы

1. В чем заключаются основные недостатки фон-неймановского стиля?
2. Какие известны математические формализации понятия алгоритма?
3. Где, когда и кем был создан первый функциональный язык программирования? На каком теоретическом фундаменте он основан?

Язык программирования Лисп

Общие сведения

Программа на Лиспе состоит из выражений. При этом каждое выражение является либо атомом, либо списком. Здесь *атом* — некоторый объект, представленный последовательностью алфавитно-цифровых символов. Примеры атомов:

```
7
A
ПРИВЕТ
S17
```

Списки — базовая составная структура данных языка. Под списком подразумевается упорядоченное множество некоторых элементов, например:

```
((КОРОЛЬ ФЕРЗЬ) ((ЛАДЬЯ) (СЛОН)) ПЕШКА)
```

Элементы могут быть как атомами, так и списками. Более того, в отличие от списков в ряде других языков программирования, в списках в Лиспе элементы могут иметь разную природу, что видно и в приведенном примере. Внутренняя часть списка заключается в круглые скобки, элементы обычно разделяются пробелами.

Важной особенностью языка Лисп является то, что сама программа в нем представляет собой список. Это дает возможность программам выполнять довольно интересные манипуляции над самими собой, а также передавать функции в качестве аргументов функций, иными словами, использовать функции высших порядков.

Несмотря на то что Лисп — язык, созданный почти сразу после Фортрана, на заре эпохи языков программирования высокого уровня, в нем воплощены несколько идей, лишь значительно позднее получивших распространение в наиболее широко используемых языках. Среди них помимо функций высших порядков:

- ссылочная организация памяти;
- удобство обработки символьных данных;
- сборка мусора (автоматическое освобождение системой не используемой более памяти);
- использование замыканий (локально определяемых в некотором контексте функций);
- функции проверки (RTTI) типа значения в ходе выполнения программы.

К достоинствам Лиспа относят точность, определенность, лаконичность (на многих неофитов производит впечатление краткость программ на Лиспе, выполняющих те же действия, что и значительно более длинные программы, например, на Паскале). Он является хорошим средством для представления древовидных структур. Интересно, что, начиная с первых реализаций, транслятор Лиспа принято разрабатывать методом раскрутки (простейшие операции реализуются в машинном коде, а остальные — на самом языке программирования). В составе системы сразу предусматривались и интерпретатор, и компилятор. Оба эти инструмента были весьма точно описаны на самом Лиспе, причем основной объем описаний не превышал пары страниц.

Все это привело к тому, что к середине 1970-х годов на Лиспе решались наиболее сложные практические задачи, связанные с принятием решений, построением систем искусственного интеллекта и пр. До сих пор Лисп остается языком № 1 в мире при создании интеллектуальных систем.

ЗАМЕЧАНИЕ

> В современные версии Лиспа, например Common Lisp, внесено множество дополнений императивного характера. Однако, поскольку они нарушают концептуальную целостность функционального программирования, в данной книге подобные возможности не затрагиваются.

Вернемся к рассмотрению языка. Некоторые атомы являются именами функций. Заострим внимание на том, что в соответствии с принятой в лямбда-исчислении, являющемся теоретической базой Лиспа, записью принято, что применение функции к аргументам записывается как список

(<*функция*> <*арг1*> <*арг2*> … <*аргn*>)

в отличие от принятой в классической математике и других языках программирования записи

<*функция*> (<*арг1*> <*арг2*> … <*аргn*>).

Чтобы определить, как воспринимать атом — как константу или как имя функции, значение которой надо вычислить, в Лиспе используется специальное ключевое слово QUOTE. Если перед атомом идет QUOTE, Лисп-система воспринимает его как не требующий вычисления. Часто вместо QUOTE перед именем, которое не нужно немедленно вычислять ставится апостроф: ‘L.

Некоторые символы имеют специальное назначение. Например, символ T означает логическую истину, NIL наряду с пустым списком — ложь.

Используя пустой список, структуру данных список можно определить рекурсивно. Итак, списком является или пустой список, или некоторый элемент, называемый головой списка, за которым следует сам список (так

называемый хвост).

Функции обработки списков

Взятие головы и хвоста некоторого списка является парой наиболее известных встроенных в Лисп функций — CAR и CDR. Любопытно, что свои названия они получили от двух ассемблерных команд ЭВМ IBM 704 — взятия значения по адресу и взятия значения из памяти с автоматическим декрементом (уменьшением на единицу) адреса. Приведенные далее примеры иллюстрируют применение CAR и CDR (справа после стрелки записан результат вычисления функции, немедленно выдаваемый Лисп-интеретатором после ввода соответствующей строки в диалоговом режиме):

```
(CAR '(A B C)) → A
(CAR '(A)) → A
(CAR 'A) → ошибка, A — атом, а не список
(CAR '(NIL)) → NIL
(CAR NIL) → NIL
(CAR '(NIL A)) → NIL
```

```
(CDR '(A B C)) → (B C)
(CDR '(A)) → NIL
(CDR 'A) → ошибка, A — атом, а не список
(CDR '(A (B C))) → ((B C))
(CDR '(NIL A)) → (A)
(CDR ())→ NIL
```

Для удобства головой пустого списка считается NIL.

Поскольку в программах на Лиспе довольно часто можно встретить цепочки последовательных вызовов функций CAR и CDR, в языке разрешается использовать удобные сокращенные обозначения вида CADDR, CADR и пр.:

```
(CADR S) ; то же самое, что (CAR (CDR S))
(CADR '(A B C)) ; получим B
(CADDAR S) ; то же самое, что (CAR(CDR (CDR (CAR S))))
```

Символы в программе на Лиспе, стоящие после точки с запятой и до конца строки, считаются комментарием.

Глядя на примеры, можно обратить внимание еще на одну особенность Лиспа. Это — язык без строгой системы типов. Из-за этого при вычислении функции она может быть применена к аргументам различного типа, что, в свою очередь, может привести к ошибке. Зато в Лиспе

имеются функции проверки типа аргумента при выполнении программы.

Некоторые функции Лиспа являются предикатными, иными словами, возвращают в качестве значения или ИСТИНУ — T, или ЛОЖЬ в виде NIL. Именно к ним относятся такие функции, как ATOM, проверяющая, является ли аргумент атомом, и NULL, определяющая, не является ли аргумент пустым списком:

```
(ATOM 'A) → T
(ATOM '(A B C)) → NIL
(ATOM ())→ T  ; пустой список атомарен

(NULL NIL) → T
(NULL 'A) → NIL
(NULL (CDR '(ATOM))) → T

(NULL T) → NIL
```

На основании этих примеров можно сделать вывод: функцию NULL можно использовать и как логическое отрицание. Впрочем, для этих же целей в Лиспе существует и предикатная функция NOT. Помимо нее есть и функции AND (логическое И, если среди последующих аргументов хоть раз встретится NIL, значением будет NIL, в противном случае — значение последнего из аргументов) и OR (логическое ИЛИ, если среди аргументов встречается выражение со значением, отличным от NIL, возвращается оно, иначе — NIL).

Функция CONS противоположна по действию CAR и CDR, она конструирует список из двух своих аргументов, которые должны быть элементом — будущей головой и списком — хвостом получаемого списка:

```
(CONS 'A '(B C)) → (A B C)
(CONS 'A NIL) → (A)
(CONS '(A B) '(C)) → ((A B) C)

(CONS (CAR '(A B C)) (CDR '(A B C))) → (A B C)
```

Помимо CONS списки создают функции LIST и APPEND. APPEND получает на вход набор списков — в общем случае произвольное число, что является, в частности, иллюстрацией допустимости в Лиспе функций произвольного количества аргументов, — и формирует из их элементов единый список, удаляя внешние скобки для каждого из исходных списков:

```
(APPEND '(A) NIL '((B)(C))) → (A (B)(C))
```

В отличие от этого функция LIST формирует общий список из своих аргументов (которыми могут быть и атомы), оставляя скобки:

```
(LIST '(A) NIL '((B)(C))) → ((A) ((B)(C)))
(LIST '(A) NIL) → ((A) NIL)
```

Функция SETQ позволяет связать некоторый атом со значением выражения, после чего он становится его именем:

```
(SETQ G '(A B(C D)E))
```

Теперь можно использовать это имя вместо исходного выражения:

```
(CAR G) → A
```

Важно подчеркнуть, что не следует использовать SETQ в качестве аналога присваивания в императивных языках программирования. Разрушающее присваивание, когда в ходе исполнения программы переменная изменяет свое значение, нехарактерно для аппликативного программирования. Вместо этого лучше думать, что SETQ в некотором смысле наклеивает ярлык на выражение.

Для работы со списками в Лиспе есть еще ряд полезных функций, назовем лишь некоторые из них. LENGTH вычисляет длину списка (количество элементов, число «братьев первого уровня») и возвращает целое число:

```
(LENGTH '((AB) C (D))) → 3
```

REVERSE обращает список, записывая элементы в обратном порядке:

```
(REVERSE '(A (BC) D)) → (D (BC) A)
```

Для определения того, является ли нечто «братом» первого уровня в определенном списке, используется предикатная функция MEMBER:

```
(SETQ M '(AB (CD) E))
(MEMBER 'F M) → NIL
(MEMBER 'C M) → NIL
(MEMBER 'B M) → T
```

Функция SUBST позволяет выполнить подстановку и имеет три аргумента. Результатом выполнения функции является значение третьего аргумента, в котором все вхождения второго аргумента заменены первым, например:

```
(SUBST 'A 'B '(A B C)) → (A A C)
```

Функция RPLACA имеет два аргумента. Первый из них — список. В качестве результата функция возвращает его с заменой первого элемента вторым :

```
(SETQ G '(A B C))
(RPLACA (CDR G) 'D) → (D C)
```

Функция RPLACD похожа на функцию RPLACA, но при ее использовании и второй аргумент должен быть списком. Он заменяет собой в результирующем списке хвост списка, являвшегося аргументом:

```
(RPLACD G '(1 2 3)) → (A 1 2 3)
```

Контрольные вопросы

1. Из каких составных частей строится программа на языке Лисп?
2. Какова основная структура данных языка Лисп?
3. В чем разница между атомом и списком?
4. Есть ли разница между записью программы и данных в языке программирования Лисп?
5. Почему функции взятия головы и хвоста списка носят названия CAR и CDR?
6. В чем сходство и различие функций LIST и APPEND?
7. Используется ли в языке Лисп префиксная, постфиксная или инфиксная форма записи выражений?
8. Приведите пример предикатной функции языка Лисп. Что такое предикатная функция? Если ли разница в языке Лисп между значением логическая ЛОЖЬ и пустым списком?
9. Зачем нужна функция SETQ? Приведите пример ее использования.
10. Какое значение считается головой пустого списка?
11. Зачем используется ключевое слово QUOTE или апостроф в языке программирования Лисп?

Функции для работы с числами

В Лисп встроен набор функций для обработки чисел. Как нужно их использовать, ясно из следующих примеров:

```
(+ 2 3) → 5
(+ 1 2 3 4 5) → 15
(- 5 2) → 3
(* 4 5) → 20
(/ 8 2) → 4
(MAX 5 8 7 6) → 8
(MIN 5 8 7 6) → 5
(SQRT 16) → 4 ; квадратный корень из числа
(ABS -3) → 3 ; модуль числа
```

Функция REM позволяет найти остаток от деления:

```
(REM 7 2) → 1
```

Для сравнения чисел можно использовать функции <, >, >=, <=:

```
(> 2 3)  → NIL
(>= 2 3)  → NIL
```

```
(>= 3 3)  → T
```

Для проверки на равенство (не только чисел) в Лиспе применяются функции EQUAL и EQ:

```
(EQ 'A 'B) → NIL
(EQ 'A 'A) → T
(EQ 'A (CAR '(A B)) → T
(EQ () NIL) → T
```

Следует помнить, что предикат EQ применим лишь к атомарным аргументам и не может быть использован для списков. Отличным в этом плане от него и более общим для списков является предикат EQUAL, позволяющий сравнивать два списка:

```
(EQUAL 'A 'A) → T
(EQUAL '(A B) '(A B)) → T
```

Предикатные функции NUMBERP и ZEROP проверяют свои аргументы. NUMBERP возвращает истину, если аргумент — число, ZEROP — если аргумент равен нулю (применим лишь числам!):

```
(NUMBERP 1) → T
(NUMBERP 'A) → NIL
(ZEROP 0) → T
(ZEROP 11) → NIL
```

Контрольные вопросы

1. Какие арифметические функции поддерживаются в Лиспе?
2. Зачем нужен контроль типов на этапе выполнения программы? Почему это требуется в программах на языке Лисп? Приведите пример использования функции NUMBERP.
3. Как используется функция ZEROP в программе на языке Лисп? Приведите пример.

Функции высших порядков

Весьма интересными возможностями обладают функции MAPCAR и APPLY. Функция MAPCAR применяет свой первый аргумент, который должен являться унарной функцией, ко всем элементам второго аргумента, как в следующем примере:

```
(MAPCAR 'ABS '(1 -2 3 -4 5)) → (1 2 3 4 5)
```

Функция APPLY применяет свой первый аргумент — функцию — ко второму:

```
(SETQ A '(4 5 8 1))
(APPLY '+ A) → 18
```

С помощью APPLY и MAPCAR, являющихся функциями высших порядков, в программах на Лиспе можно легко и элегантно переносить действия любых функций на элементы списка.

Контрольные вопросы и упражнения

1. Что такое функция высших порядков?
2. Как используется APPLY в программе на Лиспе? Приведите пример.
3. Как используется MAPCAR в программе на Лиспе? Приведите пример.

Написание программ на языке Лисп

Нас интересует не только возможность использовать встроенные функции языка программирования для вычислений. Нас интересует прежде всего возможность написания новых программ.

Написание программы на Лиспе сводится к определению новых функций. Сделать это позволяет особая функция DEFUN следующего вида:

(*DEFUN* <*атом-имя*> (<a_1>…<a_n>) <*выражение-тело*>)

Здесь первый аргумент — имя создаваемой функции, затем идет список, содержащий формальные параметры, и после завершающей список скобки — выражение, вычисление которого будет приниматься в качестве результата вычисления данной функции. Определим для примера функцию вычисления квадрата суммы двух чисел:

```
(DEFUN SUMKVAD (X Y)
    (+ (* X X) (* Y Y))
)
```

Обратите внимание на стиль написания. Аналогично программам на Си, программы на Лиспе можно сделать более удобными для восприятия человеком, применяя «лесенку», соответствующую логической структуре программы.

Теперь мы можем использовать построенную функцию:

```
(SUMKVAD 2 5) → 29
```

Однако эта функция не включает возможность обработки разных случаев при выполнении программы. В императивных языках программирования подобная обработка обычно реализуется с помощью операторов ЕСЛИ — ТО — ИНАЧЕ или операторов выбора (`if` и `switch` в Си). Лисп дает возможность реализации обработки разных ситуаций, в том числе с помощью мощной конструкции COND. Это функция, записываемая особым образом:

(*COND* (<t_1> <v_1>)
(<t_2> <v_2>)

```
      .
      .
      .
    (<t_n> <v_n>)
)
```

При ее использовании применяются пары (<t_i> <v_i>). Значение COND вычисляется по следующей схеме. Сначала вычисляется значение выражения t_1. Если оно отличается от NIL, вычисляется значение выражения v_1 и возвращается в качестве результата выполнения всей функции COND. Если результат вычисления t_1 — NIL, вычисляется значение t_2. Если оно отлично от NIL, вычисляется и возвращается в качестве итогового результата значение v_2 и т. д. Если все t_i будут иметь значение NIL, оно будет окончательным значением COND. Некоторые выражения v_i могут отсутствовать. В этом случае, если t_i будет первым отличающимся от NIL, в качестве значения COND будет возвращено это значение. В качестве t_i обычно используют предикатные функции, возвращающие значение ИСТИНА или ЛОЖЬ. Часто в последней паре в качестве t_n используют просто константу T, чтобы, если не сработала ни одна из вышележащих пар, в любом случае нечто выполнить — аналог else или default в языке Си. Следующая программа иллюстрирует использование COND. Это аналог подобной программы на Си, переводящей оценку, выраженную в баллах, в ее словесное обозначение. Кроме того, программа замечательно иллюстрирует возможности и особенности Лиспа, связанные с обработкой символьной информации и нестрогой динамической типизацией, — она производит и обратное преобразование словесного выражения оценки в баллы:

```
; функция "Оценка"
; 2- неуд/ 3- уд/ 4- хор/ 5- отл
(DEFUN MARK (A)
  (COND
     ((EQUAL A 5) 'ОТЛИЧНО)
     ((EQUAL A 4) 'ХОРОШО)
     ((EQUAL A 3) 'УДОВЛЕТВОРИТЕЛЬНО)
     ((EQUAL A 2) 'НЕУДОВЛЕТВОРИТЕЛЬНО)
     ; перевод из строки в баллы
     ((EQUAL A 'ОТЛИЧНО) 5)
     ((EQUAL A 'ХОРОШО) 4)
     ((EQUAL A 'УДОВЛЕТВОРИТЕЛЬНО) 3)
     ((EQUAL A 'НЕУДОВЛЕТВОРИТЕЛЬНО) 2)
     (T 'НЕПОНЯТНО)
  )
```

```
)
```

Итак, функция COND применяется как аналог условного оператора и оператора выбора императивных языков программирования. Что, если в программе на Лиспе необходимо неоднократно выполнить некоторые действия? В императивном языке для этого наиболее часто применяется итерация, реализуемая с помощью того или иного оператора цикла. В чистом функциональном программировании циклы не используются. Возникает законный вопрос: что же применяется вместо них?

Базовый механизм повторения в функциональной парадигме программирования — рекурсия. Напомним, что она сводится к вызову, прямому или косвенному, функции из самой этой функции.

Использование рекурсии при программировании требует определенной перестройки мышления. Необходимо забежать вперед и представить, что мы уже имеем функцию, которая выполняет нужное нам, но не совсем для полной совокупности исходных данных — например, за исключением одного элемента списка. Далее нужно представить, как мы можем использовать ее для решения всей задачи.

Кроме этого, при написании рекурсивных программ нужно помнить о необходимости избегать бесконечного повторения. В той или иной ситуации нужно остановить рекурсию. Для этого следует использовать соответствующее условие (например, с помощью функции COND языка Лисп). При обработке списков таким условием часто будет проверка входного списка на пустоту.

Рассмотрим в качестве примера программу подсчета длины списка. Подобная функция, естественно, имеется в стандартной поставке Лиспа, но мы ее повторим в учебных целях:

```
;подсчет длины списка
(DEFUN DLINA (S)
     (COND
            ((NULL S) 0)
            (T (+ 1 (DLINA (CDR S))))
     )
)
```

Поскольку для подсчета количества элементов в списке нужно их перебрать, а их несколько, нам потребуется повторение одних и тех же действий. Следовательно, программа будет рекурсивной. Что можно использовать для останова рекурсии, какое условие? Видимо, в данном случае это будет ситуация, когда на вход поступает пустой список. Проверка этого организуется с помощью функций COND и NULL. В этом случае дальнейшие вычисления не проводятся, а в качестве результата

возвращается нуль. Далее длину всего списка можно определить путем прибавления единицы к результату вычисления длины списка без одного элемента (например, головы).

Рекурсивные программы на Лиспе писать легко и просто за счет того, в частности, что базовая структура данных языка — список — рекурсивна изначально, по своей природе.

Рассмотрим еще один пример — программу, обращающую список, иными словами, записывающую элементы исходного списка в обратном порядке:

```
(DEFUN ZERKALO (S)
     (COND
            ((NULL S) S)
            (T (APPEND (ZERKALO (CDR S))(LIST (CAR S))))
     )
)
```

Программа подобно предыдущей будет являться рекурсивной в силу необходимости обработки нескольких элементов исходного списка. Останов рекурсии производится в случае наличия на входе пустого списка — его можно считать собственным обращением. Результат получаем приписыванием первого элемента исходного списка к обращенной остальной части этого списка. Поскольку все аргументы APPEND должны являться списками, используем функцию LIST для преобразования первого элемента исходного списка в список из одного этого элемента.

Следующая программа решает задачу из книги [4], а именно генерирует числовую последовательность, начиная с заданного числа, по закону

$$u_{n+1} = \begin{cases} u_n/2, & \text{если } u_n \text{ четное,} \\ \text{иначе } 3u_n + 1. \end{cases}$$

```
; PIMP – ГЕНЕРАЦИЯ ЧИСЛОВОЙ ПОСЛЕДОВАТЕЛЬНОСТИ
(DEFUN PIMP (U)
  (PRINT U)
  (COND
     ((EQUAL U 1) NIL);останов рекурсии
     ((ZEROP (REM U 2)) (PIMP (/ U 2)))
     (T (PIMP (+ 1 (* U 3))))
  )
)
```

В программе используется еще одна функция Лиспа — PRINT, печатающая значение своего аргумента. Таким образом, при очередном рекурсивном вызове функции PIMP она сначала печатает свой аргумент. В качестве условия останова применяется равенство аргумента единице.

Проверка четности осуществляется с помощью вызова функции REM, подсчитывающей остаток от деления на 2.

Следующий пример программы — функция объединения множеств на Лиспе. Множества представляются списками, подаваемыми на вход в качестве аргументов. Особенностью множеств по отношению к спискам является то, что они, во-первых, в отличие от списков, не упорядочены, а во-вторых, элементы в списке могут повторяться, а в множество в общем случае элемент может входить лишь один раз. Поэтому программа должна выявлять элементы, входящие и в первый, и во второй список. В связи с этим реализуем сначала вспомогательную функцию APP, проверяющую вхождение элемента в список. Затем она используется в функции UNI, объединяющей аргументы E и F, с помещением результата в F:

```
; Объединение множеств на Лиспе
; вспомогательная функция — поиск атома A в списке X
(DEFUN APP (A X)
   (COND
       ((NULL X) NIL)
       ((EQUAL A (CAR X)) T)
       (T (APP A (CDR X)))
   )
)

; собственно объединение множеств
(DEFUN UNI (E F)
   (COND
     ((NULL E) F)
     ((APP (CAR E) F) (UNI (CDR E) F))
     (T (UNI (CDR E) (CONS (CAR E) F)))
   )
)
```

В следующем примере разберем уже приводившуюся программу подсчета суммы нечетных элементов списка:

```
; подсчет суммы нечетных элементов списка
(DEFUN SUMNECH (X)
  (COND
     ((NULL X) 0)
     ((ZEROP (REM (CAR X) 2)) (SUMNECH (CDR X)))
     (T (+ (CAR X) (SUMNECH (CDR X))))
  )
)
```

Поскольку необходимо перебирать несколько элементов списка, программа является рекурсивной. Условие останова — пустой список на входе: ясно, что сумма нечетных в нем равна нулю. Для проверки четности используются встроенные функции Лиспа ZEROP и REM. Если первый элемент списка четный, он просто отбрасывается и продолжается выполнение для хвоста списка, в противном случае он добавляется к результату обработки хвоста списка.

Впрочем, программа подсчета суммы нечетных на Лиспе может быть написана и по-другому.

В Лиспе присутствует возможность использования так называемых *лямбда-функций*, или безымянных функций. Лямбда-функция определяется следующим образом: ((*LAMBDA* (<*пер*>) <*выр1*>) <*выр2*>) и возвращает значение выражения <*выр1*>, в котором все вхождения формального параметра <*пер*> замещены значениями выражения <*выр2*>. Первое выражение — тело лямбда-функции, второе выражение — фактический аргумент. Примеры:

```
((LAMBDA (X) (ATOM X)) 123) → T
(MAPCAR (LAMBDA (I) (+ I 10)) '(1 2 3 4 5)) → (11 12 13 14 15)
```

На сладкое — совсем короткая программа подсчета суммы нечетных элементов числовой последовательности на Лиспе с телом функции, состоящим из одной строки. Программа использует трюк со свойствами умножения и сложения, лямбда-функцию и функции высших порядков:

```
(DEFUN SUMN (S)
  (APPLY '+ (MAPCAR (LAMBDA (X) (* X (REM X 2))) S))
)
```

Обратите внимание на то, что в данной программе в явном виде рекурсия не используется.

Следующий пример программы относится к классу программ сортировки, а именно сортировке перестановкой. Идея, воплощенная в этой программе, заключается в том, что вначале на первую позицию списке перемещается минимальный элемент, а затем процедура повторяется для оставшейся части списка:

```
;Сортировка перестановкой на Лиспе

;вспомогательная функция возвращает минимальное значение в списке
(DEFUN MINS(X)
  (COND
    ((NULL (CDR X)) (CAR X));В списке один элемент – он же min
    ((< (CAR X) (MINS (CDR X))) (CAR X)); рекурсия
    (T (MINS (CDR X)))
```

```
  )
)

;вспомогательная функция удаляет первое вхождение элемента в список
(DEFUN REMV (EL S)
  (COND
    ((NULL S) NIL)
    ((EQ EL (CAR S)) (CDR S)); если первый — возвращается хвост
    (T (CONS (CAR S) (REMV EL (CDR S))));рекурсия
  )
)

;собственно функция сортировки списка по возрастанию перестановкой
(DEFUN SORTS(X)
  (COND
    ((NULL (CDR X)) X)
    (T (CONS (MINS X) (SORTS (REMV (MINS X) X))))
  )
)
```

Еще один вариант программы сортировки основан на идее вставки, начиная с первого, элемента в список на подобающее ему место в зависимости от величины, своего рода выполнение команды «Равняйся!»:

```
;Сортировка вставкой на Лиспе

;вспомогательная функция вставки элемента
;в список на подобающее место
(DEFUN INS(X S)
  (COND
    ((NULL S) (LIST X)); если был пустой список
    ((< X (CAR S)) (CONS X S) );если меньше первого,вставляется
перед ним
    (T (CONS (CAR S) (INS X (CDR S))))
  )
)

;функция сортирует список по возрастанию
(DEFUN SORTS(X)
  (COND
    ((NULL X) X)
    (T (INS (CAR X) (SORTS (CDR X))))
  )
)
```

Следует упомянуть также о том, что в функциональном программировании используется концепция «ленивых вычислений», когда значение функции вычисляется лишь в момент, когда (и если) оно потребуется.

Порядок вычисления сложных выражений в Лиспе следующий:

1. Аргументы функции вычисляются в порядке перечисления. Пример: `(FUN A B C)`.
2. Композиции функций вычисляются от внутренней к внешней. Пример: `(FUN1 (FUN2 A) B)`.
3. Представление функции анализируется до того, как начинают вычисляться аргументы, так как в некоторых случаях аргумент можно и не вычислять.
4. При вычислении лямбда-выражений связи между именами переменных, а также между именами функций и их определениями накапливаются в так называемом ассоциативном списке, пополняемом при вызове функции и освобождаемом при выходе из функции.

Читателю предоставляется возможность в качестве упражнения самостоятельно разработать программы на Лиспе, реализующие другие алгоритмы сортировки (например, сортировку Шелла или быструю сортировку).

Завершая данный подраздел, остается заметить, что автор ни в коей мере не претендовал на полное описание возможностей Лиспа — более того, некоторые из них намеренно были исключены из рассмотрения с целью соблюдения чистоты функционального подхода. Автор лишь надеется на то, что читатель смог в некоторой степени ощутить особый вкус функционального программирования.

Контрольные вопросы

1. Какая функция языка программирования Лисп используется для определения новых функций?
2. Что такое безымянная функция в Лиспе? Приведите пример.
3. Применяются ли при написании программ на функциональных языках программирования средства структурирования текста, принятые в императивных языках: отступы (лесенка), пустые строки, комментарии? Уместны ли они в функциональных программах?
4. Лаконичнее ли программы на Лиспе программ на императивном языке с подобной функциональностью? Почему?
5. Опишите функцию `COND` языка Лисп. Почему она широко в нем используется?

6. Каковы причины использования рекурсии в программах на языке Лисп?
7. Что такое условие останова рекурсии и зачем оно нужно?

Современное состояние и перспективы функционального программирования

Подчеркнем, что в настоящее время функциональное направление в языках программирования активно развивается, и несмотря на наибольшую известность Лиспа, он далеко не единственный — и не самый концептуально чистый! — представитель семейства аппликативных языков, то есть языков, в которых главным действием остается применение функции к аргументу. Среди наиболее известных — Hope, Miranda, целое семейство языков, порожденных Haskell (назван в честь математика и логика Хаскелла Карри), — Curry, Clean, Gofel. Еще одно семейство функциональных языков, ведущих начало от языка ML, включает Standard ML, Objective CAML, Harlequin's MLWorks, F# и др. Языки Nemerle и F# построены на платформе .Net.

Язык Joy базируется на композиции функций, а не на лямбда-исчислении. Это язык, родственный Форту, хотя и не является его прямым наследником. В настоящее время Joy считается каноническим примером языка *конкатенативного программирования*. Все в Joy является функциями, принимающими стек как аргумент и возвращающими стек в качестве результата.

Бывает, что создатели изначально функционального языка программирования стремятся позволить использовать те или иные возможности других подходов. Так, языки Scala и Oz относятся к так называемым *мультипарадигмальным*, или языкам, в которых поддерживается более одной систем взглядов на программирование, но в них обоих функциональная парадигма занимает достойное место.

Язык Erlang выбивается из общего ряда функциональных языков, создаваемых в основном в исследовательских и научных организациях. Он разработан и активно используется в своих продуктах телекоммуникационным гигантом — компанией Ericsson. Отличительной особенностью является то, что Erlang включает в себя средства порождения параллельных процессов и их коммуникации. Программа транслируется в байт-код, исполняемый виртуальной машиной, что обеспечивает переносимость. Программы, написанные на Erlang, способны работать на нескольких узлах. Узлами могут быть процессоры, многие ядра одного процессора либо целый кластер машин. Таким образом

поддерживается распределенное программирование. При этом чем сложнее приложение на Erlang и чем больше оно создает процессов, тем легче его масштабировать.

Весьма любопытно семейство языков, порожденное APL (пример программы на этом языке содержится во введении к настоящей книге). Сам APL с его весьма оригинальной системой знаков сейчас является лишь страницей в истории программирования, но на нем основаны такие реально используемые языки, как J и K.

Среди отечественных разработок функциональных языков упомянем Пифагор (параллельный информационно-функциональный алгоритмический (Красноярск, 1995)) и Фактор.

Все же наиболее многочисленным остается Лисп-семейство, включающее сегодня многочисленные ответвления (MuLisp, Common Lisp, Scheme, Closure, Sisal, FP, FL и др.). В качестве примеров применения Лиспа в индустрии помимо систем искусственного интеллекта назовем AutoLisp — диалект, используемый как встроенный базовый язык программирования в системе автоматизации проектирования AutoCAD. На Лиспе написан популярный в UNIX-сообществе текстовый редактор Emacs. Встроен язык и в свободно распространяемый аудиоредактор Audacity, и в не менее свободный графический редактор Gimp.

Контрольные вопросы и упражнения

1. Какие функциональные языки программирования кроме Лиспа вам известны?
2. Существуют ли применяемые в промышленности языки функционального программирования?
3. Существуют ли отечественные языки функционального программирования? Перечислите.
4. Что такое гибридные (мультипарадигмальные) языки? Приведите примеры.
5. Каково ваше мнение о перспективах функциональных языков программирования в будущем?

Введение в логическое программирование

Логическое программирование — наряду с функциональным — еще одна из наиболее известных альтернатив традиционному императивному программированию. Можно сказать, что парадигма логического

программирования еще более повышает уровень абстракции.

Достаточно сказать, что языки логического программирования, в частности Пролог, заслуженно могут быть отнесены к декларативным. Иными словами, Пролог-система может находить решение задачи самостоятельно, без четкого предписания со стороны программиста, как это сделать. Программисту необходимо лишь указать, что мы хотим получить, и четко описать предметную область на некотором формальном языке.

Например, мы знаем, что $x \cdot 15 - 1 = 224$. Как найти x? На языке Пролог-Д для этого достаточно написать:

```
?УМНОЖЕНИЕ(x,15,-1,224).
```

Система дает немедленный ответ: `x=15`. Но самое замечательное здесь то, что мы можем использовать одну и ту же программу для решения другой задачи: сколько нужно отнять от произведения $x \cdot 15$, чтобы получить 224? Для этого следует лишь переставить переменную:

```
?УМНОЖЕНИЕ(15,15,x,224).
```

и в ответ получим: `-1`. Не правда ли необычно? В традиционных языках программирования мы привыкли к тому, что программа имеет четко определенные входные и выходные данные. Более того, мы можем использовать Пролог, например, для нахождения всех пар аргументов, дающих заданный ответ. Например, мы можем спросить у системы GNU Prolog, как сформировать список `[a,b,d]` слиянием двух подсписков, следующим образом (на особенностях синтаксиса читателю предлагается пока не останавливаться, хотя все довольно прозрачно):

```
| ?- append(X,Y,[a,b,d]).
```

и получить набор ответов:

```
X = []

Y = [a,b,d] ? ;


X = [a]

Y = [b, d] ? ;


X = [a,b]

Y = [d] ? ;


X = [a,b,d]

Y = [] ? ;
```

Нажатие на клавишу ; используется для получения очередного варианта ответа.

В основу языков логического программирования, как следует из самого названия, положена логика. А логика, как известно, это «наука о правильном мышлении», «правила рассуждения».

С точки зрения приверженцев этого стиля, язык программирования должен быть инструментом мышления. Л. Стерлинг и Э. Шапиро [14] пишут: «Мы склонны думать, что программирование может и должно быть частью процесса собственно решения задачи; что рассуждения можно организовать в виде программ, так что следствия сложной системы предположений можно исследовать, “запустив” предположения; что умозрительное решение задачи должно… сопровождаться работающей программой, демонстрирующей правильность решения и выявляющей различные аспекты проблемы».

Некоторые исследователи не совсем корректно относят к декларативному программированию также функциональные языки. Но в них четко указывается, что является аргументами, а что — результатом, то есть приведенный ранее трюк невозможен. Сходство и различие языков программирования Лисп и Пролог приведены в табл. 1.

Таблица 1

Сходство	**Различие**
Оба ориентированы на символьную обработку	В Прологе теоретической основой является логика предикатов, а в Лиспе — лямбда-исчисление
Широко применяется рекурсия	Лисп активнее используется в США, а Пролог — в Европе и Японии
Активно используются при разработке систем искусственного интеллекта	В Лиспе вычисляется единственное значение функции, а в Прологе система может самостоятельно искать множество значений, удовлетворяющих наложенным ограничениям, причем как от аргументов к значению функции, так и наоборот, от значения к приводящим к его получению аргументам

Непредубежденному человеку, не знакомому с кухней создания ЭВМ, машина фон Неймана покажется причудливым существом. Рассуждения в терминах машины требуют значительных мыслительных усилий. Эта особенность программирования на компьютерах фон Неймана приводит к разделению создателей программ на изобретающих методы решения задач алгоритмистов и реализующих алгоритм на ЭВМ знатоков особенностей машинного языка — кодировщиков. При этом в программировании, как и в логике, требуется явное выражение знаний с помощью некоторого формализма. Подобная формализация утомительна, однако в логике полезна, поскольку обеспечивает понимание задачи. Для машины фон

Неймана это вряд ли приводит к подобному эффекту. Компьютерам все еще далеко до того, чтобы стать равными партнерами человека в интеллектуальной деятельности. Однако использование логики при программировании представляется естественным и плодотворным, поскольку она сопровождает процесс мышления.

Логическое программирование сильно отклоняется от основного пути развития языков программирования. Здесь используется не некоторая последовательность преобразований, отталкивающихся от архитектуры фон Неймана и присущего ей набора операций, а теоретическая модель, никак не связанная с каким либо типом машины.

Логическое программирование базируется на следующем убеждении: не человека надо учить мышлению в терминах компьютера, а напротив, компьютер должен уметь выполнять действия, свойственные человеку. Логическое программирование подразумевает, что конкретные инструкции не задаются, вместо этого используются логические аксиомы и правила вывода, с помощью которых формулируются сведения о задаче и предположения, достаточные для ее решения.

Постановка задачи формализуется в виде логического утверждения, подлежащего доказательству. Такое утверждение называется целевым утверждением или вопросом. Программа — это множество аксиом и правил вывода, а исполнение — попытка логического вывода целевого утверждения из программы. При этом используется конструктивный метод доказательства целевого утверждения, то есть в процессе доказательства находятся аргументы, которые делают истинным данное целевое утверждение.

Покажем, как можно в логическом языке задать некоторое правило рассуждений, например: «любит — значит, дарит цветы». На Прологе-Д можно записать:

```
дарит_цветы(x,y):-любит(x,y). %логическое правило
любит(Вася, Маша). %факт
любит(Коля, Даша). %факт
?дарит_цветы(x,y). %вопрос Пролог-системе.
```

Будет сгенерирован набор ответов:

x=Вася

y=Маша

x=Коля

y=Даша

Пролог можно «научить» и другим правилам, например: «любит — значит, дарит брильянты» или, согласно поговорке, «бьет — значит, любит». Читатель, возможно, уже догадался, что мы подходим к тому, что логика бывает разной. Это действительно так. Кроме классической, существуют различные виды логик, Приложение Б содержит описания некоторых логик, или так называемых *формальных систем*. Наиболее бурное развитие логика получила с начала XX века, когда математикам удалось логику формализовать.

Примерами неклассических логик являются (приводятся характерные для них формулировки):

- темпоральные («когда-нибудь пойдет снег», «снег будет идти всегда»);
- многозначные (пример трехзначной: истина/ложь/неопределенность, пятизначная — система школьных оценок и т. п.);
- нечеткая (размытая);
- вероятностная («с вероятностью 0,7 Вася любит Машу»);
- логика возможных миров («X истинно в возможном мире К» (варианты развития событий)).

Изначально логика считалась частью философии. Среди ученых, фиксировавших законы правильного мышления, можно назвать Аристотеля, Авиценну, П. Абеляра, И. Канта, Г. В. Лейбница. Со временем логика становилась все более точной. Значительный вклад в этот процесс внесли такие ученые, как Дж. Буль и де О. Морган.

Можно сказать, что, начиная с трудов Дж. Пеано, а также классической книги Б. Рассела и А. Уайтхеда «Основания математики», формальная логика выделилась в самостоятельную дисциплину. Здесь, однако, возникают некоторые вопросы, например:

- В какой степени уместно распространение математических методов, принятых в теории чисел и алгебре, на логику?
- Каким образом переходить от логической символики к конкретной интерпретации, связанной с той или иной предметной областью? Иначе говоря, истинна ли некоторая формальная логика применительно к данной предметной области?

Математика всегда считалась областью знаний с точной системой рассуждений, однако лишь в 1882 году были сформулированы несколько положений, на которых основаны «Начала» Евклида, например, что между двумя несовпадающими точками всегда найдется третья. В 1895 году Д. Гильберт сформулировал свой знаменитый список проблем и предложил построение всей математики на основе набора аксиом. Но в 1931 году К. Гедель показал, что это невозможно. Он в своей знаменитой

теореме о неполноте (см. [2]) указал на неустранимые внутренние ограничения довольно мощных формальных систем.

Получается следующее. Логика лежит в основаниях самой математики. Леммы, теоремы, доказательства — неотъемлемые составляющие математического знания. Математика, в свою очередь, лежит в основе множества наук, более того, по степени значимости применения математического аппарата мы выделяем точные науки.

Классическая логика Аристотеля получила свою формализацию в виде *исчисления высказываний* и *исчисления предикатов первого порядка*. Однако возможны различные логические системы. Какая из них «правильная»? Доказано ли, что наш мир действительно соответствует логике предикатов? На самом деле мы *верим*, что наш мир устроен в соответствии с классической логикой. Получается, что в основании всего научного знания лежит вера в некоторые начальные условия.

Происхождение информатики довольно сильно связано с математической логикой. Теоретические основы и той, и другой наук были заложены в 20–30-х годах XX века такими выдающимися учеными, как Э. Пост, А. Черч, К. Гедель, Б. Рассел, А. Тьюринг. Не все знают о роли отечественных ученых, например М. Шейнфинкеля, заложившего основы комбинаторной логики.

Неудивительно, что с появлением ЭВМ и созданием для них первых программ возникало желание «скрестить» логику и программирование. К 1969 году относятся первые получившие довольно широкую известность попытки создания логических языков программирования — это Absys, созданный в университете Абердина в Шотландии, и Planner — фунционально-логический язык, разработка К. Хьюита из лаборатории искусственного интеллекта Массачусетского технологического института (США). Planner впоследствии породил целый ряд потомков, среди которых QA4, Popler, Conniver, QLisp и Ether.

Одним из ранних языков логического программирования стал *эквациональный* язык Golux в котором программа представляет собой совокупность равенств (1973). Его автор П. Хайес сформулировал идею «вычисления = контролируемая дедукция».

Однако наибольшую популярность завоевал Пролог и его диалекты [14], разработанный в 1972 году в университете Марсель-Экс. Авторами Пролога являются А. Кольмрауэр и Р. Ковальский. Среди наиболее известных диалектов Пролога, поддерживаемых соответствующими трансляторами и средами разработки, отметим GNU Prolog, Turbo Prolog, Prolog-10, SW-Prolog, IC Prolog, Edinburgh Prolog, Visual Prolog, MU-Prolog, P# для платформы .Net. Были созданы версии для параллельного

программирования, например Parlog и Parallel Prolog. Отметим реализации Пролога на базе русского синтаксиса встроенных предикатов, например уже упоминавшийся Пролог-Д.

Впоследствии были разработаны и другие языки логического программирования, среди которых в первую очередь можно назвать Mercury, Strand, ALF, Fril, Gödel, XSB, KLO, ShapeUp, Hayes.

Весьма популярными стали и так называемые гибридные языки — языки, поддерживающие сразу несколько парадигм программирования. Функционально-логическим можно считать широко используемый для создания многопоточных приложений в телекоммуникационной отрасли Erlang, разработанный фирмой Ericsson. Интересен язык Oz с поддерживающей его интегрированной средой программирования Mozart.

Поскольку Пролог распространен наиболее широко, именно он рассматривается здесь как пример языка логического программирования.

Контрольные вопросы

1. Относится ли логическое программирование к декларативному подходу?
2. Что такое логика?
3. Существуют ли другие языки логического программирования помимо Пролога? Были ли у него предшественники?
4. В чем состоят основные сходства и различия между Лиспом и Прологом?
5. Какие неклассические логики существуют? Приведите примеры.
6. Что такое конструктивный метод доказательства?
7. Фиксируются ли четко в Прологе данные-аргументы и результат или можно подходить к этому гибко?

Язык программирования Пролог

Как уже отмечалось, язык Пролог был создан во французском городе Марселе. Целью работы было создание языка, который мог бы делать логические заключения на основе заданного текста. Название Prolog является сокращением от Programming in logic. Главными разработчиками являются Р. Ковальский и А. Кольмрауэр.

Перед этим Ковальский работал над программой на Фортране, предназначенной для доказательства теорем. Эта программа должна была обрабатывать фразы на естественном языке. Первая реализация языка Пролога с использованием компилятора Вирта ALGOL-W была закончена

в 1972 году, а основы современного языка были заложены в 1973 году.

Пролог постепенно распространялся среди ученых, занимавшихся логическим программированием, однако недостаток эффективных реализаций сдерживал его распространение.

ЗАМЕЧАНИЕ

Синтаксис различных реализаций Пролога довольно существенно различается.

Следует отметить, что, как и в случае с Лиспом, прогресс в области производительности средней ЭВМ позволил в значительной степени снять проблемы Пролога, связанные с не столь высокой скоростью обработки, как, например, в Си.

Программа на языке Пролог состоит из предложений (выражений). Предложения строятся из атомарных элементов — переменных, констант, а также *термов*. Для обозначения переменных в GNU Prolog используются латинские буквы и цифры, начиная с прописной латинской буквы. В некоторых реализациях допускается использование особой *безымянной переменной*, обозначаемой символом подчеркивания _. Константы обозначаются с помощью латинских букв и цифр. Пролог-Д для этих же целей помимо латинских позволяет использовать русские буквы. В качестве констант большинство реализаций Пролога допускает также использование чисел.

Терм определяется рекурсивно:

- константы и переменные — термы;
- термами являются также составные конструкции, содержащие имя функтора и последовательность из одного или более заключенных в скобки аргументов, также являющихся термами. Функтор задается своим именем и арностью — количеством аргументов.

Примеры термов:

- `петя` — простой терм, состоящий из одной константы `петя`;
- `J(0)` — терм с арностью 1 с функтором `J` и аргументом константой `0`;
- `Горячий (молоко)` — терм с функтором `Горячий` арности 1 и аргументом `молоко`;
- `Имя (Ваня,Коля)` — терм с функтором `Имя` арности 2 и аргументами `Ваня` и `Коля`;
- `List(a,li(b,n))` — терм с функтором `List` арности 2 с первым аргументом константой `a` и вторым аргументом составным термом `li` арности 2 и аргументами `b` и `n`.

Термы, в которые не входят переменные, называются основными. Подразумевается, что функтор позволяет получить некоторое значение. Функторы могут быть разными.

Фундаментальную роль в языке Пролог играют *предикаты*. Напомним математическое определение: предикат — это функция, принимающая значения из двухэлементного множества, например: ИСТИНА и ЛОЖЬ, 0 и 1, ДА и НЕТ. Существует тесная связь между предикатами и отношениями. Если некоторые предметы A_1, A_2, …, A_n вступают между собой в отношение P, можно сказать, что, будучи использованными в качестве аргументов соответствующего предиката арности n, они дают в качестве результата значение ИСТИНА и ЛОЖЬ — в противном случае.

Программа на языке Пролог может включать несколько видов выражений:

- факты;
- правила вывода;
- вопросы.

Также допустимы комментарии. В качестве комментария воспринимается любая последовательность символов, начинающаяся с символа % и заканчивающаяся символом конца строки.

Факты в Прологе — это предикаты, заканчивающиеся точкой.

Примеры фактов в программе на Прологе:

```
Отец (Алексей,Иван).      %Алексей – отец Ивана
Плюс(2,3,5).   %2+3=5
любит(Вася,Маша).
любит(Коля,Маша).
любит(Вася,яблоки).
```

ЗАМЕЧАНИЕ

Как правило, тот или иной факт из жизни (предметной области) можно сформулировать разными способами. Искусство программирования на Прологе заключается, в частности, в их правильном выборе.

Примеры записи одного и того же факта из жизни в программе на Прологе:

```
дарит_цветы(Вася,Маша).
дарит(Вася,Маша,цветы).
дарит(цветы,Вася).
получатель(цветы,Маша).
```

ЗАМЕЧАНИЕ

Если нужно отразить факт $f(X, Y) = Z$, можно использовать предикат FP(X,Y,Z).

Вопрос записывается так же, как и факт, то есть обычно представляет собой предикат. Как правило, вопрос, в отличие от факта, предваряется символом ?. Допускаются так называемые конъюнктивные вопросы, содержащие несколько имен предикатов, разделенных запятой. В конце вопроса ставится точка, как и при оформлении факта.

Вопрос является целью логического вывода для Пролог-системы. Система должна попытаться найти ответ на вопрос в виде ДА или НЕТ.

В программе может формулироваться несколько вопросов-целей. Примеры вопросов в программе на Прологе:

```
?любит (Вася,Маша). % а любит ли Вася Машу? Да или нет?
?любит (Вася,X). % перечислить все X, которых любит Вася
?любит (Вася,_). % любит ли Вася хоть кого-нибудь? Да или нет?
?любит (X,яблоки) % выдать всех, кто любит яблоки
?любит (X,Y) % вывести всех на чистую воду
?любит (X,Маша),любит (X,яблоки). %найти любящего Машу яблокоеда
```

Правило — это утверждение, включающее две части, разделенные символами :- (в некоторых реализациях Пролога может использоваться <- или иная комбинация символов):

$A:-B_1,B_2,..,B_n.$

В конце правила вывода также присутствует точка. Выражение A называется заголовком, или заключением правила, в правой части идет его тело, состоящее из B_n — посылок, или гипотез. Должно соблюдаться условие $n \geq 0$, и заголовок, и посылки должны представлять собой предикаты.

Примеры правил вывода языка Пролог:

```
сын (X,Y):-отец (Y,X),мужчина(X).
дочь (X,Y):-отец (Y,X),женщина(X).
```

ЗАМЕЧАНИЕ

Допускается несколько правил вывода с одним и тем же заголовком.

В простейшем случае программа на Прологе может состоять из одного лишь вопроса (цели). Читатель может поинтересоваться: а о чем спрашивать систему, если мы не задали ей ни одного факта? Дело в том, что большинство реализаций Пролога поставляются с заранее определенными встроенными предикатами, иногда простейшими арифметическими и логическими, а иногда формирующими довольно богатую библиотеку (например, в GNU Prologe).

Контрольные вопросы и упражнения

1. В каком году, где и кем был создан язык программирования Пролог? Какие задачи решали его создатели? Различается ли синтаксис разных реализаций Пролога?
2. Из чего строится программа на языке Пролог?
3. Что такое терм в Прологе?
4. Что такое предикат в Прологе? Приведите пример.
5. Что такое факт в программе на языке Пролог? Приведите пример.
6. Что такое вопрос в программе на языке Пролог? Приведите пример.
7. Что такое правило вывода в программе на языке Пролог? Приведите пример.

Написание баз данных и знаний на Прологе

Весьма хорошо Пролог подходит для написания баз данных и знаний. Попробуем проделать это на примере базы знаний о родственных связях. Иногда подобная база может оказаться весьма полезной — не всегда Интернет может оказаться под рукой, когда потребуется найти значения загадочных понятий «деверь», «золовка» или «внучатый племянник»…

Побудительным мотивом для написания этой программы может служить то, что родственная связь — тоже отношение, соответственно, легко переводится в базовые для Пролога термины предикатов. Для начала необходимо определиться, от каких базовых родственных связей (отношений) мы будем отталкиваться, задавая более сложные отношения. Логично выделить самую близкую кровную связь, а именно «быть родителем» («быть ребенком»). Сразу договоримся для определенности, что первым аргументом будет идти родитель, а вторым — его ребенок, и станем придерживаться этого же принципа для всех последующих предикатов.

Весьма важной в человеческой цивилизации является половая принадлежность (не будем вставать на позиции властей некоторых современных европейских стран…). Итак, можно выделить также предикаты для «быть мужчиной» и «быть женщиной». Мы здесь не имеем в виду переносный смысл этих понятий, а подходим к делу чисто формально.

Немаловажен для нас и институт брака. Введем базовый предикат для обозначения отношения «быть супругами».

ЗАМЕЧАНИЕ

Можно было и по-другому выбрать базовые предикаты, например, отталкиваться от «мама» и «папа».

Предположим, у нас имеются факты, описывающие некоторую семью:

```
родитель(Коля,Петя).
родитель(Коля,Маша).
родитель(Коля,Володя).
мужчина(Коля).
мужчина(Володя).
женщина(Маша).
супруги(Коля,Нина).
```

Программа на Прологе, содержащая одни лишь факты, является прямым аналогом *базы данных* — информационной системы, в которой можно запоминать некоторую информацию и из которой по запросу пользователя извлекать ее. Например, в данном случае мы можем задать системе вопрос: кто является детьми Коли?

```
? родитель(Коля,X).
```

Иными словами, формулируем к нашей базе следующий запрос: «выдай все X такие, для кого Коля является родителем». В настоящее время наиболее распространены так называемые *реляционные* базы данных, опирающиеся на понятие «отношения». Пролог построен на базе предикатов, а предикаты неразрывно связаны с отношениями. Поэтому неудивительны явные аналогии между реляционной базой данных и программой на Прологе. Имя предиката является аналогом имени отношения (таблицы). Каждый факт соответствует строке отношения (таблицы) в реляционной базе.

Однако Пролог обладает более богатыми возможностями. На нем мы легко можем создавать не просто базы данных, а *базы знаний*. Определение базы знаний подразумевает активный характер ее содержимого, возможность на основании одной имеющейся в базе информации получать (выводить) другую. Поддерживающий этот процесс механизм в Прологе основан на использовании правил вывода.

Попробуем научить программу определять, кто кому является матерью, отцом, дочерью, сыном, братом и сестрой. Можно сделать безусловно верный вывод о том, что если некто является родителем и при этом мужчиной, он будет отцом. Правило вывода на Прологе может быть сформулировано следующим образом (в примере, поскольку для установления факта отцовства несущественно, кто именно является ребенком, используется безымянная переменная _):

```
отец(A,B):-родитель(A,_),мужчина(A).
```

Аналогичным образом формулируется правило вывода для матери. Для получения на основании фактов о том, что B является женщиной и некоторый A является родителем B, заключения о том, что B — дочь, можно использовать правило вывода

```
дочь(B,A):-родитель(A,B),женщина(B).
```

Подобным же образом можно построить правила вывода для получения информации о сыне, братьях и сестрах. Получится следующий набор правил вывода:

```
отец(A,B):-родитель(A,_),мужчина(A).
мать(A,B):-родитель(A,_),женщина(A).
сын(A,B):-родитель(B,A),мужчина(A).
дочь(B,A):-родитель(A,B),женщина(B).
брат(A,B):-родитель(Y,A),родитель(Y,B),мужчина(A).
сестра(A,B):-родитель(Y,A),родитель(Y,B),женщина(A).
```

На следующем этапе перейдем к другому поколению — опишем родственные связи вида «бабушка», «дедушка», «внук» и «внучка»:

```
внук(A,B):-родитель(Z,Y), родитель(Y,A),мужчина(A).
внучка(A,B):-родитель(Z,Y),родитель(Y,A),женщина(A).
бабушка(A,B):-родитель(A,Y), родитель(Y,B),женщина(A).
дедушка(A,B):-родитель(A,Y), родитель(Y,B),мужчина(A).
```

Перейдем к тетям и дядям, племянникам и племянницам:

```
племянник(A,B):-родитель(Y,A),брат(Y,B),мужчина(A).
племянник(A,B):-родитель(Y,A),сестра(Y,B),мужчина(A).
племянница(A,B):-родитель(Y,A),брат(Y,B),женщина(A).
племянница(A,B):-родитель(Y,A),сестра(Y,B),женщина(A).
тетя(A,B):-племянник(B,A),женщина(A).
тетя(A,B):-племянница(B,A),женщина(A).
дядя(A,B):-племянник(B,A),мужчина(A).
дядя(A,B):-племянница(B,A),мужчина (A).
```

В этих примерах продемонстрированы такие особенности Пролога, как существование нескольких правил вывода для одного и того же заключения (Пролог-система перебирает их все в процессе попытки вывода цели и, если какое-то правило подходит по посылкам, использует его) и получение симметричного отношения из ранее определенного, например, «тетя» на базе «племянник» или «племянница».

Следующие правила позволяют описать некоторые нетривиальные родственные связи:

```
стрый(x,y):-папа(z,y),брат(x,z). % дядя по отцу (брат отца)
уй(x,y):-мама(z,y),брат(x,z). % дядя по матери (брат матери)
свекор(x,y):-муж(z,y),папа(x,z). % отец мужа
```

```
свекоровь(x,y):- муж(z,y), мама(x,z). % мать мужа
тесть(x,y):-жена(z,y),папа(x,z). % отец жены
теща(x,y):-жена(z,y),мама(x,z). % мать жены

зять(x,y):- муж(x,z),дочь(z,y). % муж дочери
невестка(x,y):-жена(x,z),сын(z,y). % жена сына
сноха(x,y):- жена(x,z),папа(y,z). % жена сына для его отца

деверь(x,y):-муж(z,y),брат(x,z).% брат мужа
золовка(x,y):-муж(z,y),сестра(x,z). % сестра мужа
шурин(x,y):-жена(z,y),брат(x,z). % брат жены
свояченица(x,y):-жена(z,y),сестра(x,z). % сестра жены
```

Пример взят из реальной написанной студентом программы, основанной на несколько иных базовых предикатах.

Наконец, сформулируем более глубокое понятие, а именно *предок*. Правда, в русской разговорной речи не принято относить к предкам собственно родителей, а также бабушек и дедушек. Однако в молодежном жаргоне это принято: кто не слышал выражения «предки на даче»? Мы последуем этому примеру:

```
предок(A,B):-родитель(A,B). %граничное условие
предок(A,B):-предок(A,C),родитель(C,B).
```

Контрольные вопросы

1. Существует ли связь между программированием на Прологе и реляционными базами данных? Почему? Удобно ли программировать базы данных на Прологе?
2. В чем отличие базы знаний от базы данных? Можно ли программировать базы знаний на языке Пролог?
3. Почему отношение «предок» строится как рекурсивное? Как задается в программе на Прологе граничное условие?

Введение арифметики через логику в Прологе

Весьма интересным является использование Пролога для демонстрации мощи логики. Например, мало кто задумывается о том, что обычная школьная арифметика может быть определена логическим путем. Мы будем рассматривать это в несколько упрощенном виде, интересующихся более строгой теорией отсылаем к *арифметике Пеано* [2].

Для начала необходимо логически определить понятие «натуральное число». Примем, что ноль является натуральным числом. На Прологе это может быть записано, например, как

```
nat(0).
```

ЗАМЕЧАНИЕ

> В отечественных школах обычно принято начинать ряд натуральных чисел с единицы, однако в мире допускается начинать и с нуля. Нам удобнее поступить именно так.

Однако натуральный ряд с нуля всего лишь начинается, а уходит он в бесконечность! Что мы можем сделать для того, чтобы вместить понятие натурального числа в компьютер? Как уже отмечалось, не все понимают, что ЭВМ с конечной памятью просто не может работать с, если можно так выразиться, «настоящими» математическими числами. Имеют дело с конечными приближениями чисел — как действительных, так и дробных. Именно отводимое под стандартное представление числа в компьютере количество бит называется его разрядностью. Если при выполнении вычислений должно быть получено большее число, чем допускает разрядность ЭВМ, как правило, происходит так называемая *ошибка переполнения*. Но строгая, точная математическая логика должна предоставлять инструмент для описания бесконечного ряда чисел (оставляя пока за скобками то, что логика в Прологе реализуется на реально существующей ЭВМ с присущими ей ограничениями)!

Действительно, логический подход позволяет использовать функтор `s(X)`, смысл которого заключается в том, что мы вводим для натурального числа операцию взятия следующего за ним по порядку числа (эквивалентную прибавлению единицы). Например, для нуля это будет единица, для единицы — двойка и т. д. Во введенных обозначениях единица — число, следующее за нулем, — может записываться как `s(0)`, следующее за единицей число два — как `s(s(0))` и т. д.

ЗАМЕЧАНИЕ

> Фактически, мы воспользовались рекурсией. Натуральные числа — простейшая рекурсивная структура данных.

Следующий важный шаг состоит в том, что мы вводим логическое правило вывода о том, что если некоторое число является натуральным, то следующее за ним по порядку (получаемое с помощью функтора `s(X)`) также будет являться натуральным числом. На Прологе (GNU Prolog) это может быть записано как

```
nat(s(X)): – nat(X).
```

Данный пример служит наглядной иллюстрацией ранее упоминавшегося факта о широком использовании рекурсии при программировании на Прологе. Остановом рекурсии служит предложение, в котором в качестве аргумента фигурирует `0`.

Итак, программа на Прологе, определяющая натуральные числа (пока без операций над ними), выглядит следующим образом:

```
nat(0).
nat(s(X)):- nat (X).
```

Кому-либо это может показаться удивительным, но может быть строго математически доказано, что порождаемые этой Пролог-программой сущности `0`, `s(0)`, `s(s(0))` и т. д. эквивалентны натуральным числам, по крайней мере до момента, пока программа не прервется вследствие исчерпания наличествующей памяти компьютера.

ЗАМЕЧАНИЕ

> Программа, состоящая из факта и итерации, иногда называется минимальной рекурсивной программой.

Продолжим развитие нашей логической программы, описывающей натуральные числа. На множестве натуральных чисел существует естественный порядок. Программа может задать отношение ≤ (меньше или равно) следующим образом:

```
mir(0,X):–nat(X).
mir(s(X),s(Y)):–mir(X,Y).
```

Это тоже рекурсивная программа, основывающаяся на том, что если $X \leq Y$, то $X + 1 \leq Y + 1$. Останов рекурсии, или граничное условие, здесь базируется на том, что ноль меньше любого натурального числа или равен ему.

Перейдем к арифметическим действиям. Фундаментальным является сложение. На Прологе логически оно может быть введено как

```
suma(0,X,X):–nat(X).
suma(s(X),Y,s(Z)):-suma(X,Y,Z).
```

Попробуем разобраться, на чем строится данный фрагмент программы. Поскольку операция сложения двухместная, ей может быть сопоставлен трехместный предикат `suma`. Будем подразумевать, что предикат истинен, если первый аргумент плюс второй аргумент дают третий аргумент.

Первая строка означает, что добавление нуля не изменяет любое натуральное число. Вторая — что если у нас есть тройка чисел, таких, что $X + Y = Z$, то $(X + 1) + Y = (Z + 1)$. Возможно, кому-то покажется удивительным, но этого достаточно, чтобы логически определить сложение. Более того, эта же программа умеет выполнять вычитание. Убедиться в этом несложно — достаточно задать Пролог-системе несколько вопросов (здесь для краткости ответ системы записывается в той же строке после стрелки), например:

```
?suma(0,s(0),s(0)). → ДА (yes)
?suma(0,s(0),X). → s(0)
?suma(s(s(0)),s(s(0)),X). → s(s(s(s(0))))
?suma(s(0),X,s(s(s(0)))). → s(s(0))
?suma(X,Y,s(s(s(0)))). %ищет все пары X,Y такие что X+Y=3
```

Здесь сначала проверяется 0 + 1 = 1?, во второй строке вычисляется сумма нуля и единицы. Помимо прочего — на всякий случай! — мы проверяем, чему у нашей программы равно 2 + 2.

Две последующие строки иллюстрируют мощь Пролог-системы — мы используем определение сложения фактически для вычитания, или решаем уравнение $1 + X = 3$, а в следующей строке вообще автоматически получаем все пары натуральных чисел, дающих в сумме 3.

ЗАМЕЧАНИЕ

> Естественно, в приводимых здесь программах для записи чисел-аргументов и интерпретации аргументов используется принятая нотация 0 — 0, 1 — s(0), 2 — s(s(0)) и т. д.

Получив такие обнадеживающие результаты для сложения, перейдем к умножению. Аналогично предикату suma определим трехместный предикат um:

```
um(0,X,0). % любое число при умножении на ноль дает ноль
um(s(X),Y,Z):-um(X,Y,W),suma(W,Y,Z).%(X+1)*Y=Z,если X*Y=W, Z=X+W
```

Предикат определяется рекурсивно, как и suma, но при определении умножения мы дополнительно опираемся на ранее определенное логически сложение. Проверим, чему равно 2 · 2 у нашей программы:

```
?um(s(s(0)),s(s(0)),X). → s(s(s(s(0))))
```

На волне успеха перейдем к определению операции возведения в степень:

```
step(step(X),0,0).
step(0,step(X),s(0)).
step(step(N),X,Y):-step(N,X,Z),um(Z,X,Y).
```

Далее приводятся чуть более сложные программы (тем не менее опирающиеся на логическое введение натуральных чисел), сопровождаемые примерами вопросов для их проверки (предоставляем это читателю выполнить самостоятельно):

```
%факториал
fakt(0,s(0)).
fakt(s(X),Z):-fakt(X,Q),um(s(X),Q,Z).

?fakt(s(s(s(0))),X).
```

```
%минимум
mini(X,Y,X):-mir(X,Y).
mini(X,Y,Y):-mir(Y,X).

?mini(s(s(0)),s(0),X).

%строго меньше
men(0,s(X)):-nat(X).
men(s(X),s(Y)):-men(X,Y).

? men(X,s(s(0))).

%остаток от деления
ost(X,Y,X):-men(X,Y).
ost(X,Y,Z):-suma(X1,Y,X),ost(X1,Y,Z).

?ost(s(s(s(0))),s(s(0)),X).


%наибольший общий делитель числа
nod(X,0,X).
nod(X,Y,G):-men(0,X),men(0,Y),ost(X,Y,Z),nod(Y,Z,G).

?nod(s(s(s(s(0)))),s(s(0)),X).
```

Контрольные вопросы и упражнения

1. Можно ли сказать, что натуральные числа — простейшая рекурсивная структура?
2. Эквивалентны ли сущности, порождаемые приведенной программой на языке программирования Пролог, числам в математике?
3. Что такое минимальная рекурсивная программа на языке Пролог?
4. Приведите пример определения операции сложения логическими средствами.
5. Приведите пример определения операции умножения логическими средствами.
6. Приведите пример определения отношения «меньше или равно» логическими средствами.
7. Напишите другой вариант программы на Прологе для нахождения наибольшего общего делителя.

Обработка списков на языке Пролог

В языке программирования Пролог, как и в Лиспе, активно используется рекурсия. Есть в нем и удобные средства для обработки такой базовой рекурсивной структуры данных, как список.

Для задания списка в Прологе достаточно заключить в квадратные скобки некоторую последовательность термов, перечисленных через запятую. В предельном случае список может быть пустым и обозначается просто `[]` (или `nil`). Список может включать в свой состав вложенные списки, как и в языке программирования Лисп.

Примеры списков на Прологе:

```
[a,b,c]
[1,2]
[]
[[a,b],2,[[a]]]
```

Весьма удобным средством Пролога, превосходящим по краткости средства Лиспа CAR и CDR, является способ взятия начального элемента и хвоста списка.

Предположим, что в списке надо выделить голову и обозначить ее X и хвост, обозначив его Y. В Прологе достаточно написать:

```
[X|Y]
```

Просто поставив вертикальную черту, можно сразу выделить и первый элемент, и все остальные. Более того, Пролог допускает записи вида

```
[X,Y|Z]
```

В приведенном примере выделяются первый и второй элементы списка X и Y и хвост Z.Заметим, что нам для этого не потребовались ни цикл, ни переменная-итератор, которые, скорее всего, нужны были бы в аналогичной программе на императивном языке.

Логическими средствами Пролога список может быть определен как

```
список([]).
список([X|S]):-список(S).
```

Разберем приведенную программу. Итак, сначала утверждается, что пустой список есть список. И во второй строке добавление в начало списка S произвольного элемента X оставляет структуру списком.

Приведенный пример позволяет в первом приближении почувствовать стиль написания программ обработки списков на языке Пролог. Пойдем дальше. Вхождение элемента в список проверяет следующая программа:

```
членсп(X,[X|_]).
```

```
членсп(X,[Y|S]):-членсп(X,S).
```

Перейдем от проверки вхождения одного элемента в список (хотя этим элементом может быть и подсписок) к проверке того, является ли некоторая последовательность элементов подсписком некоторого списка. Для упрощения решения задачи построения предиката `подсписок` удобно решать ее поэтапно и сначала реализовать два других предиката: `префикс` и `суффикс`. Под префиксом здесь понимается подсписок в начале другого списка, под суффиксом — подсписок, завершающий другой список.

ЗАМЕЧАНИЕ

> В программировании на Прологе использование самостоятельных содержательных отношений в качестве вспомогательных предикатов — типовая техника.

Произвольный подсписок может быть потом определен как суффикс некого префикса или префикс некого суффикса:

- `[А,Б,В,Г,Д,Е]` — исходный список;
- `[А,Б,В]` — префикс;
- `[Б,В,Г,Д,Е]` — суффикс;
- `[Б,В,Г]` — подсписок.

Приведем определение подсписка через суффикс и префикс на Прологе:

```
подсписок(X,Y):-префикс(P,Y),суффикс (X,P).%префикс суффикса
подсписок (X,Y):-префикс(X,S),суффикс(S,Y). %суффикс префикса
```

Сами предикаты `префикс` и `суффикс` могут быть определены следующим образом:

```
префикс([],_).
префикс([X|S],[X|Y]):-префикс(S,Y).
```

В приведенной программе утверждается, что, во-первых, пустой список может считаться префиксом любого списка, а во-вторых, если некоторый подсписок S является префиксом списка Y, добавление одинаковых элементов в начало S и Y не изменит ситуации:

```
суффикс(X,X).
суффикс(X,[_|S]):-суффикс(X,S).
```

Здесь сначала определяется, что список может считаться собственным суффиксом, а затем рекурсивно определяется суффикс при добавлении в начало рассматриваемого списка произвольного элемента.

Однако определение подсписка через префикс суффикса или, наоборот, суффикс префикса не является единственно возможным. Вообще при программировании на Прологе часто встречается ситуация, когда одно и то

же понятие можно определить разными способами. Как и при программировании на других языках, мы имеем дело с ситуацией *семантически* одинаковых, но *синтаксически* различающихся программ. Но разница обычно не только в записи — разные реализации могут иметь *существенно различающиеся* показатели эффективности, например количество операций, необходимых для достижения поставленной цели.

ЗАМЕЧАНИЕ

> Можно сказать, что искусство программирования состоит в написании простых, понятных и легко модифицируемых, но при этом эффективных программ.

В следующей программе предикат `подсписок` определяется через префикс и рекурсию:

```
подсписок(X,Y):-префикс(X,Y).
подсписок(X,[Y|S]):-подсписок(X,S).
```

Сначала говорится, что префикс — частный случай подсписка, а затем — что подсписок некоторого списка остается им при добавлении в начало этого списка элемента X.

Следующей интересной операцией над списками является слияние списков (приписывание, или конкатенация). Определим предикат `слияние`, имеющий три аргумента и становящийся истинным, если при приписывании второго аргумента-списка к первому получится третий список:

```
слияние([],Y,Y).
слияние([D|X],Y,[D|Z]):-слияние(X,Y,Z).
```

Данная программа рекурсивная, в ней говорится сначала о том, что приписывание пустого списка не меняет любой список (граничное условие рекурсии), а затем — что если два списка давали при слиянии некоторый список, добавление в начало результата и одного из аргументов одинаковых элементов не изменит ситуации.

Любопытно, что, имея предикат `слияние`, можно еще одним способом определить предикат `подсписок`, весьма кратко:

```
подсписок(X,S):-слияние(A,W,S),слияние(X,B,W).
```

Сравним эту строку с воображаемой программой на Паскале или Си, делающей то же самое.

Следующая программа выделяет в списке концевой элемент:

```
последний(X,S):-слияние(_,[X],S).
```

Интересным является написание — по аналогии с ранее созданной программой на Лиспе — и сравнение с ней программы обращения списков

на Прологе:

```
обрат([],[]).
обрат([X|S],Z):-обрат(S,Y),слияние(Y,[X],Z).
```

Возможен и другой вариант этой программы — так называемое обращение с накоплением. Здесь вводится одноименный предикат `обрат` с тремя аргументами, истинный, если третий аргумент `результат` получается приписыванием второго к элементам обращенного первого. Переменная `A` используется в качестве некоего аккумулятора:

```
обрат([],Y,Y).
обрат(X,Y):-обрат(X,[],Y).
обрат([X|S],A,Y):-обрат(S,[X|A],Y).
```

ЗАМЕЧАНИЕ

> В Прологе допускается создание одноименных предикатов, различающихся количеством аргументов. Прослеживается аналогия с объектно-ориентированным программированием, где допустимы одноименные методы, различающиеся типом и количеством аргументов.

Рассмотрение программ обработки списков было бы неполным, если бы мы не рассмотрели программу подсчета суммы нечетных членов числовой последовательности на языке Пролог:

```
СУМНЕЧ([],0). %останов рекурсии и сумма для пустого списка =0
СУМНЕЧ([X|Y],S):-ЧЕТ(X), СУМНЕЧ (Y,S). %четные — пропускаем
СУМНЕЧ ([X|Y],S):-НЕЧЕТ(X),СУМНЕЧ(Y,L),СЛОЖЕНИЕ(L,X,S).%нечетные
складываем
```

Правда, в случае отсутствия в стандартной библиотеке используемой версии Пролога предикатов `ЧЕТ` и `НЕЧЕТ` их придется определить отдельно (версия для Пролога-Д):

```
ЧЕТ(X):- УМНОЖЕНИЕ(2,Y,0,X).
НЕЧЕТ(X):- УМНОЖЕНИЕ(2,Y,1,X).
```

Еще короче будет программа с использованием ранее показанного трюка, основанного на алгебраических свойствах нуля и единицы:

```
%Сумма нечетных "хитрым" способом
СУМН([],0).
СУМН([x|y],w):-
УМНОЖЕНИЕ(r,2,z,x),УМНОЖЕНИЕ(x,z,0,p),СУМН(y,q),СЛОЖЕНИЕ(p,q,w).
```

Рассмотрим программы сортировки списков. В программировании на языке Пролог плодотворно применяется метод «образовать и проверить». Дело в том, что, как мы видели ранее, Пролог-система в процессе вычислений способна перебирать возможные значения аргумента. В методе «образовать и проверить» один из предикатов служит для

генерации возможных вариантов ответа, а другой — для проверки того, является ли полученное решение искомым — удовлетворяющим заданным требованиям. Данный подход применим для решения переборных задач, встречающихся в задачах создания интеллектуальных систем, например логических игр и пр.

Если подходить к задаче сортировки с позиций метода «образовать и проверить», можно создать предикат `упорядочен`, который будет проверять список, полученный в качестве аргумента, на отсортированность. Другой же предикат может предназначаться для генерации всех возможных перестановок списка. Несомненно, среди них будет и искомая — когда элементы выстроены по порядку. Начнем с предиката `упорядочен`:

```
упорядочен([X]).
упорядочен([X,Y|S]):-МЕНЬШЕ(X,Y),упорядочен([Y|S]).
```

Первая строка программы говорит о том, что мы можем считать список, состоящий из одного элемента, упорядоченным. Вторая строка показывает, что если мы имеем некоторый упорядоченный список, начинающийся с элемента Y и имеющий хвост S, и добавляем в его начало меньший Y элемент X, список останется упорядоченным.

Для предиката `перестановка` нам потребуется вспомогательный предикат `вырез`, который будет извлекать из списка некоторый элемент, точнее, его первое вхождение (операция вырезания). Поскольку Пролог не реализует непосредственно функций, предикат будет иметь следующие аргументы: первый — элемент, подлежащий извлечению из списка, второй — список, подлежащий обработке, и наконец — результирующий список:

```
вырез(X,[X|S],S).
вырез(X,[Y|S],[Y|Z]):-вырез(X,S,Z).
```

В первой строке говорится, что если извлечь первый элемент из списка с головой X и хвостом S, получается список S. Вторая строка рекурсивная и показывает, что если в результате вырезания элемента X из списка S получается список Z, то в случае рассмотрения списков, полученных путем добавления головы Y к S и Z, ситуация не изменится.

Напишем теперь предикат `перестановка`, основанный на операции `вырез`:

```
перестановка([],[]).
перестановка(X,[Z|S]):-вырез(Z,X,Y),перестановка(Y,S).
```

Первая строка показывает, что как ни переставляй содержимое пустого списка, получишь лишь пустой список. Вторая строка говорит, что,

например, получить из списка X его перестановку можно, взяв некоторый элемент Z из списка и переставив его на первое место. Наконец, сам предикат сортировка, который и будет собственно процедурой сортировки, выглядит весьма просто:

```
сортировка(X,Y):-перестановка(X,Y),упорядочен(Y).
```

Приведем еще два варианта программ сортировки на Прологе: сортировку вставками и быструю сортировку, — предоставив читателю самостоятельно разобраться в них:

```
%Программа сортировки вставкой на Прологе-Д
СОРТ([X|W],Y):-СОРТ(W,Z),ВСТАВКА(X,Z,Y).
СОРТ([],[]).
%Вспомогательный предикат — вставка
ВСТАВКА(X,[Y|w],[Y|z]):-БОЛЬШЕ(X,Y),ВСТАВКА(X,w,z).
ВСТАВКА(X,[],[X]).
ВСТАВКА(X,[Y|W],[X,Y|W]):-РАВНО(X,Y).
ВСТАВКА(X,[Y|W],[X,Y|W]):-МЕНЬШЕ(X,Y).

%быстрая сортировка на GNU Prolog
%слияние меньших, чем X, затем X и затем больших X
quick([X|S],Y):-
part(S,X,M,Bs),quick(M,L),quick(Bs,B),append(L,[X|B],Y).
quick([],[]).
%вспомогательный предикат разбиения упорядоченного списка с
головой X на подсписки
%содержащие меньшие и большие выделенного элемента Y
part([X|S],Y,[X|L],B):- X =< Y,part(S,Y,L,B).
part([X|S],Y,L,[X|B]):- X > Y,part(S,Y,L,B).
part([],Y,[],[]).
```

Автор надеется, что благодаря приведенным примерам читатель уже смог в достаточной мере почувствовать вкус программирования на Прологе, его особенности и отличия от программирования на императивных языках. Можно сказать, что здесь написание программы напоминает формальную постановку задачи. При этом программист избавлен от мелких деталей, связанных с реализацией решения на ЭВМ, таких как размер массива ячеек памяти, отведенных под данные, или индекс в этом массиве. Иными словами, Пролог поднимает программирование на более высокий уровень *абстракции*.

Контрольные вопросы

1. Как записываются списки в программах на языке Пролог?

2. Как взять голову и хвост списка в программе на Прологе? Сравните синтаксис с синтаксисом Лиспа. Какой вам больше нравится и почему?
3. Допускаются ли в Прологе несколько одноименных предикатов? Чем они должны отличаться друг от друга?
4. Что означает символ подчеркивания _ в программах на языке Пролог?
5. Взятие суффикса префикса и префикса суффикса списка даст одно и то же? Что получается в результате?
6. Что такое метод «образовать и проверить»? Приведите пример программы на язык Пролог, использующей данный подход.
7. Лаконичнее ли программы на Прологе аналогичных по функциональности программ на императивном языке? Почему? Программ на языке программирования Лисп? Что такое уровень абстракции в языке программирования?

Задача о ханойской башне

Согласно легенде, где-то в джунглях в окрестностях Ханоя спрятан древний монастырь. Монахи в нем делают свою работу, заключающуюся в следующем. Имеется три золотых стержня, назовем их условно А, Б и Д. На стержни нанизывают золотые диски, имеющие различный диаметр (рис. 1). Всего у монахов 64 диска.

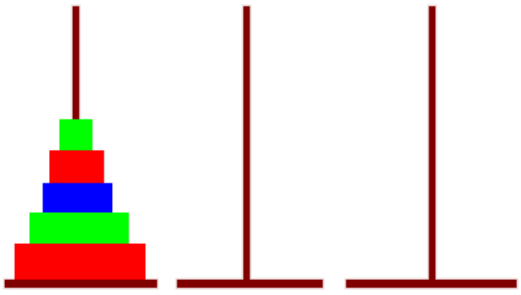


Рис. 1.

В первоначальном положении все диски находятся на стержне А в допустимом положении: можно, чтобы на диске большего диаметра был расположен меньший диск, на нем — еще меньший и т. д. Необходимо перенести все диски на стержень Б, используя при необходимости стержень Д в качестве вспомогательного. За один ход переносится лишь один диск, на каждом ходе на всех стержнях должно соблюдаться изначальное условие упорядоченности дисков.

Согласно легенде, когда монахи завершат свой труд, наш мир исчезнет. Несведущий в математике человек может испугаться подобной перспективы, но не стоит беспокоиться. Самый быстрый способ решения этой задачи требует $2^{64} - 1$ перемещений диска. Если для перемещения не

самого легкого золотого диска монахам требуется всего пара секунд, то при условии отсутствия перерывов на сон и еду они потратят на работу примерно 2^{65} секунд. Задумаемся, что это значит по человеческим меркам. В часе 3600 секунд, в сутках 24 часа, или 86 400 секунд, год продолжается примерно $2^5 \cdot 2^{10} \cdot 2^{10} \approx 32$ млн секунд, или $\approx 2^{25}$ секунд. Получается, что 2^{65} секунд $\approx 2^{40}$ лет $\approx 2^{10} \cdot 2^{30} \approx 1000 \cdot 2^{10} \cdot 2^{10} \cdot 2^{10} \approx 1$ трлн лет. Напомним, что, по данным современной физики, возраст Вселенной составляет менее 14 млрд лет.

ЗАМЕЧАНИЕ

> Мощь человеческого разума позволила придумать обозначения для невероятно больших чисел — достаточно использовать степени. Числу 10^{100} дано название гугол, которым воспользовалась компания интернет-поиска Google. Но назвать — еще не означает осознать.

Данная задача часто приводится как пример проблемы, имеющей естественное и красивое рекурсивное решение. Неудивительно, что на Прологе она решается изящной и довольно лаконичной программой. Наша программа будет генерировать ходы — перемещения диска с одного стержня на другой.

Главным предикатом программы пусть будет предикат `Ханой`. При вызове `?Ханой(3,"А","Б","Д").` он генерирует цепочку ходов вида «с А на Б, с А на Д, с Б на Д…». Первый аргумент у данного предиката – число дисков, изначально находящихся на первом стержне. Второй, третий и четвертый – наименования дисков.

```
Ханой(1,а,б,_):-ВЫВОД("с ",а," на ",б),ПС.
Ханой(N,а,б,д):-БОЛЬШЕ(N,1),Ханой(#N-1#,а,д,б),ВЫВОД("с ",а," на ",б),ПС,Ханой(#N-1#,д,б,а).
```

Разберем эту программу.

Вторая строка, служащая для останова рекурсии, говорит о том, как решить вырожденную задачу — перенести один диск со стержня `а` на стержень `б`, используя в качестве вспомогательного третий. Ясно, что для этого достаточно сделать единственное перемещение. Встроенный в Пролог-Д предикат ВЫВОД используется для информирования пользователя о произведенном переносе диска (ПС означает перенос строки).

Вторая строка использует рекурсию дважды. Перенести N дисков со стержня `а` на `б` используя в качестве вспомогательного стержень `д`, можно, перенеся N-1 диск со стержня `а` на `д` (как вспомогательный при этом

можно использовать стержень б), сделав единственный ход с а на б, и перенеся N-1 диск со стержня д на стержень б (теперь как вспомогательный можно использовать стержень а).

Две приведенные строки решают задачу.

Контрольные вопросы

1. Почему задача о ханойской башне рекурсивна по своей природе — каково ваше мнение?
2. Сколько времени понадобится монахам для решения задачи?

Программы обработки информации, записанной символами

Как говорилось ранее, Пролог, как и Лисп, изначально ориентирован на обработку символьной информации, а не на вычисления, как, например, Фортран. В языке программирования Пролог переменная может принимать как числовое, так и символьное значение.

Мощь обработки информации, записанной символами, может быть проиллюстрирована приведенными далее примерами, в первом из которых Пролог «обучается» правильно распознавать в некой последовательности символов корректно записанный в соответствии с определенными правилами многочлен, а затем выполнять дифференцирование функций (имеется в виду получение по записи исходной функции записи формулы вычисления производной). Чем не искусственный интеллект?!

ЗАМЕЧАНИЕ

> В силу некоторых ограничений версии Пролог-Д примеры в данном подразделе приводятся на GNU Prolog.

Программа, распознающая многочлены:

```
%Программа распознавания многочленов
%Вспомогательные предикаты
constant(X):-atom(X).
constant(X):-integer(X).
natural_number(X):-integer(X),X>0.
%Символьное распознавание полиномов
poly(X,X).
poly(Term,X):-constant(Term).
poly(Term1+Term2,X):-poly(Term1,X),poly(Term2,X).
poly(Term1-Term2,X):-poly(Term1,X),poly(Term2,X).
poly(Term1*Term2,X):-poly(Term1,X),poly(Term2,X).
poly(Term1/Term2,X):-poly(Term1,X),constant(Term2).
```

```
poly(Term^N,X):-natural_number(N),poly(Term,X).
```

Приведенной программе можно задать вопрос `?poly(x^2-3*x,x)` и получить ответ «да» на английском языке.

Следующей программой будет программа символьного дифференцирования (подобные преобразования иногда называют символьными вычислениями, на них специализируются системы компьютерной алгебры, например Maxima). Предикат `derivative` истинен, если последний аргумент — запись производной по второму аргументу функции, запись которой подается в качестве первого аргумента.

Программа символьного дифференцирования на языке GNU Prolog:

```
%Программа символьного дифференцирования выражений
%Справочник базовых производных для популярных функций
diff(X,X,1).
diff(X^N,X,N*X^(N-1)).
diff(sin(X),X,cox(X)).
diff(cos(X),X,-sin(X)).
diff(e^X,X,e^X).
diff(log(X),X,1/X).

%Правила дифференцирования сумм, разностей, частных
diff(F+G,X,DF+DG):-diff(F,X,DF),diff(G,X,DG).
diff(F-G,X,DF-DG):-diff(F,X,DF),diff(G,X,DG).
diff(F*G,X,F*DG+DF*G):-diff(F,X,DF),diff(G,X,DG).
diff(1/F,X,-DF/(F*F)):-diff(F,X,DF).
diff(F/G,X,(G*DF-F*DG)/(G*G)):-diff(F,X,DF),diff(G,X,DG).
```

Нетрудно убедиться в способностях программы, задавая ей вопросы вида, например, `?derivative(e^x,x,e^X)` или `diff(e^x+sin(x),x,X)`.

Контрольные вопросы

1. Что такое символьные вычисления? Где применяется компьютерная алгебра, как Вам кажется?
2. Универсальнее ли программы обработки символьной информации программ, работающих с числами?

Отрицание и отсечения в Прологе

Некоторую сложность в логическом программировании с помощью Пролога имеет рассмотрение логического отрицания. Дело в том, что используемый в Прологе механизм логического вывода на основе *метода резолюции* не позволяет точно устанавливать ложность некоторого

предложения. В силу этого «настоящего» строгого логического отрицания большинство версий Пролога не включают.

Частичной, но не строгой заменой логического понятия ЛОЖЬ может служить расценивание ложности как *невыводимости*. Иначе говоря, если, исходя из заданных программой фактов и правил вывода, заданную цель вывести не удается, она считается ложной. Однако при реализации описанного подхода нас тоже подстерегают подводные камни: в связи с ограниченностью ресурсов ЭВМ логический вывод может прерваться, не дойдя до доказательства нужного предложения.

Отрицание в смысле невыводимости может быть реализовано в Прологе с помощью так называемого *отсечения*. Под отсечением понимается заведомое отбрасывание некоторых ветвей в дереве перебора вариантов ответа. Во многих программах это имеет смысл, например, из эвристических соображений. Кроме того, у Пролога имеется проблема получения нескольких одинаковых ответов, если в программе можно логически прийти к ним разными путями. Отсечение позволяет прервать процесс логического вывода, если уже получен ответ. Записывается отсечение с помощью восклицательного знака `!`, который рассматривается как специальный предикат без аргументов, всегда возвращающий ИСТИНУ и не дающий откатиться назад, чтобы выбрать альтернативное решение для уже установленных подцелей, то есть тех, которые в строке программы расположены *левее* отсечения. На подцели, расположенные *правее*, отсечение не влияет. Кроме этого, отсечение отбрасывает все предложения программы, находящиеся *ниже* места, где сработало отсечение.

Пример программы поиска максимума с использованием отсечения:

```
maxi(X,Y,X):-X>Y,!.
maxi(_,Y,Y).
```

Нахождение максимума из трех чисел с использованием отсечения:

```
max3(X,Y,Z,X):-X>Y,X>Z,!.
max3(_,Y,Z,Y):-Y>=Z,!.
max3(_,_,Z,Z).
```

В программах на Прологе отсечение позволяет имитировать конструкцию `if…then…else…` императивных языков следующим образом.

```
%if <Y> then P
%  else P2
S:-<Y>,!,P.
S:-P2.
```

После унификации порождающей цели с заголовком предложения, содержащего отсечение, цель выполняется и фиксируется для всех

возможных выборов предложений. Конъюнкция целей до отсечения приводит не более чем к одному решению, но не влияет на те, что выводятся после него. В случае возврата они могут порождать более одного решения

Пример программы определения многочлена при использовании отсечения:

```
polinom (X,X):-!.
polinom (Term,X):-constant(Term),!.
polinom (Term1+Term2,X):-!, polinom (Term1,X), polinom (Term2,X).
polinom (Term1-Term2,X):- !, polinom (Term1,X), polinom (Term2,X).
polinom (Term1*Term2,X):-!, polinom (Term1,X), polinom (Term2,X).
polinom (Term1/Term2,X):-!, polinom (Term1,X), polinom (Term2,X).
polinom (Term1^N,X):-!, natural_number(N),polinom (Term1,X).
```

Как видите, отсечение позволяет достичь двух целей:

- получать более компактные программы;
- оптимизировать процесс вычислений за счет отбрасывания заведомо неперспективных ветвей дерева поиска.

Вернемся к определению отрицания в Прологе. Можно использовать следующую программу, использующую также встроенные системные предикат `fail` и `call` (пример на GNU Prolog):

```
not(X):-call(X),!,fail.
not(X).
```

Выполняется эта программа следующим образом. Применяется первое правило. Если предложение `X` доказуемо, происходит отсечение, соответственно, цель не выполняется. Если `X` не выводится, то происходит переход ко второй строке.

Контрольные вопросы

1. Существует ли в языке программирования Пролог логическое отрицание?
2. Что такое отсечение? Зачем оно используется в программах на Прологе? Как работает механизм отсечения?
3. Как реализовать отрицание в языке программирования Пролог на основе отсечения? Приведите пример программы.

Жизненный цикл программных средств

Все сложнее, чем кажется.

Один из законов Мэрфи

Индустрии программирования необходимо чудо-чудо, которое воплотило бы в жизнь мечту о быстрой и легкой разработке программ.

Мануэль Т.

На заре программирования часто возникала следующая ситуация. Необходимо было решить на ЭВМ некую прикладную задачу. Поскольку компьютеры были весьма редкими и дорогостоящими устройствами, к работе на них допускали лишь самых лучших инженеров и ученых — часто самих создателей ЭВМ, знающих их досконально. Автор самостоятельно выполнял математическую постановку задачи, разрабатывал алгоритм решения и записывал его на используемом на данной машине языке программирования. Проверял соответствие функционирования программы первоначальному замыслу и исправлял ошибки — производил отладку. Затем он же эксплуатировал программу и при необходимости вносил в нее изменения. Время эксплуатации и число пользователей были весьма ограниченными.

Шло время, компьютеры становились более доступными. Ряды программистов ширились, и не всегда за счет выдающихся личностей, обладающих высочайшим профессионализмом. Усложнялись решаемые задачи, стало актуальным разделение труда. Появилось деление на математиков-алгоритмистов и программистов менее высокого уровня, осуществлявших не совсем правильно звучащее по-русски «кодирование». В этих условиях правильным шагом было доверить проверку функционирования программ не самим их создателям. Родилась профессия специалистов по тестированию ПО (иногда их еще называют специалистами по качеству, Quality assurance).

Большие программы создаются большими коллективами людей. Размер некоторых современных программ (например, операционных систем ЭВМ) достигает десятков миллионов операторов. Трудоемкость их написания составляет сотни и даже тысячи человеко-лет. Сотни модулей взаимодействуют между собой посредством десятков тысяч параметров. Можно сказать, что одной из неотъемлемых черт современного

программного обеспечения является сложность.

Многие программы эксплуатируются по всему миру миллионами пользователей на протяжении нескольких лет, а некоторые — десятков лет.

Родилось понятие *жизненного цикла* программной системы. Он включает в себя все этапы, начиная от постановки задачи до вывода из эксплуатации. Рассмотрим примерный жизненный цикл заказного программного обеспечения.

На каждом из этапов в процессе участвуют люди, выполняющие разные обязанности, — можно сказать, играющие различные роли, и формируются разные виды документов, иногда называемые *артефактами*. Иногда, особенно при нехватке в организации специалистов, вызванной экономией, что встречается сплошь и рядом, один и тот же человек выступает в нескольких роля, в предельном случае — сам себе и швец, и жнец, и на дуде игрец.

Итак, сначала возникает идея (замысел) программного средства, основанный на определенной потребности. Заказчика обращается в организацию, занимающуюся созданием программных систем (иногда замысел возникает внутри этой организации — в этом случае речь идет об инициативной разработке).

Для начала неплохо оценить, разрешима ли вообще задача. Некоторые задачи вообще не имеют алгоритмического решения, и это строго математически доказано. На этом этапе в игру вступает *системный аналитик* (иногда разделяют его роль и роль *бизнес-аналитика*, но это весьма близкие понятия). Проанализировав проблему, необходимо сделать заключение о том:

- что задача в принципе разрешима;
- задача разрешима не только теоретически за миллион лет, но и практически, и организация берется за дело.

После этого начинается формализованная постановка задачи. Это весьма важный этап, завершающийся созданием документа, называющегося техническим заданием (ТЗ), *спецификацией,* иногда просто требованиями к программе. В нем дается ответ на вопрос: что нужно сделать (помимо функциональных обычно выдвигают и нефункциональные требования, например, к быстродействию, совместимости с другим ПО, переносимости и пр.)? Чем более точно будет поставлена задача, тем меньше вероятность возникновения печальной ситуации, встречающейся в жизни чаще, чем хотелось бы, когда, выполнив большой объем работы усилиями десятков людей, получают не совсем то, что было нужно, а иногда — совсем не то и все идет в корзину. Весьма наглядно это показано на известном рисунке с

качелями, появившемся в 1973 году в бюллетене вычислительного центра Лондонского университета (рис. 2).

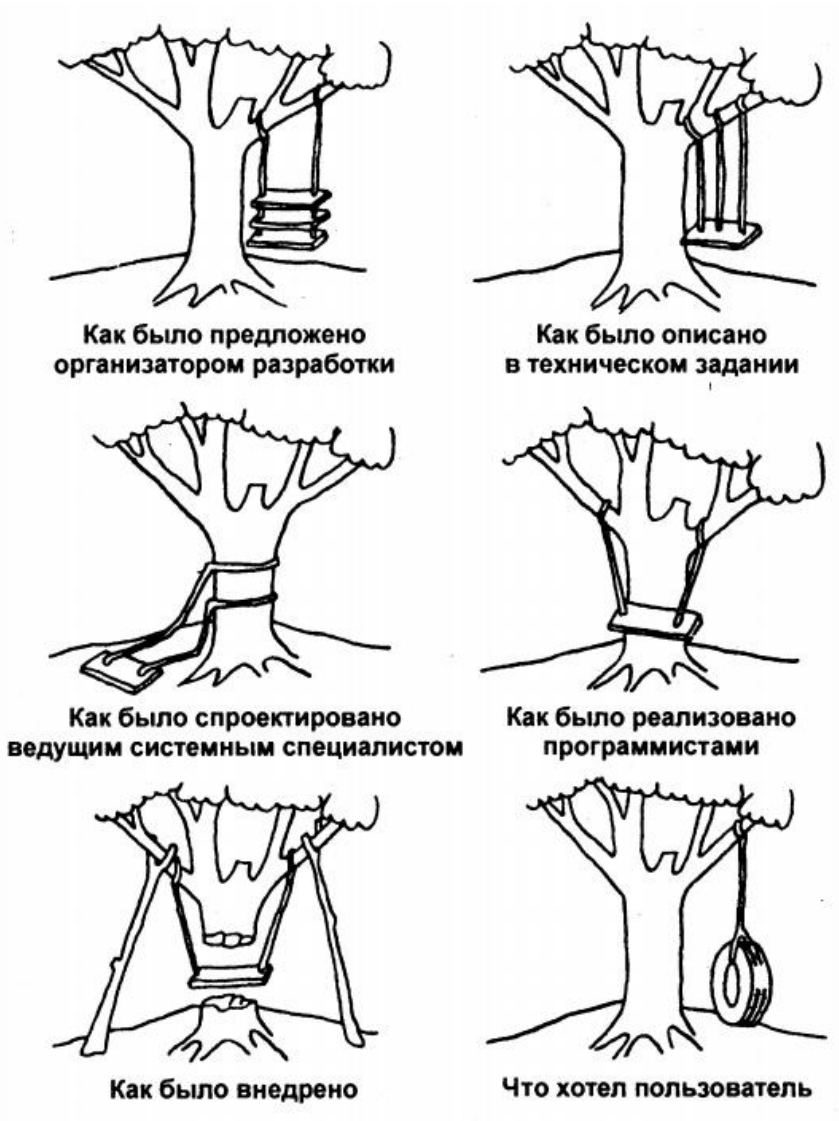


Рис. 2

Основным автором ТЗ является системный аналитик. Он должен быть специалистом высокого уровня, обладать способностью оценить, зачем и кому нужен предполагаемый продукт, уметь общаться с людьми — предполагаемыми пользователями, выявлять их нужды, владеть методами формализованного описания задачи (UML, IDEF, SADT и пр.) и т. д. Неотъемлемой его чертой, что видно даже из названия роли, должно быть системное мышление. Системный аналитик должен быть компетентен также в области методов технической реализации проекта, чтобы правильно оценить его разрешимость. ТЗ должно быть максимально четко и однозначно написано, в нем следует по максимуму использовать строгие формальные нотации, его должен подписать заказчик, что позволит в дальнейшем минимизировать споры и конфликты.

ЗАМЕЧАНИЕ

> Некоторые современные модели разработки программ, например test driven development, требуют, чтобы уже на этом этапе подключались специалисты по тестированию и для каждого требования изначально разрабатывался тест, позволяющий проверить выполнение данного требования.

После ответа на вопрос: «Что?» необходимо понять, как это делать. Ответ на этот вопрос должен содержаться в документе, который может называться техническим проектом, архитектурным проектом ПО, или Software Arcitecture Document. В нем должны быть ответы на следующие вопросы:

- Из каких частей и подсистем будет состоять программная система?
- Какие языки, библиотеки и платформы будут использованы?
- Какие важнейшие алгоритмы применять?
- Как будут организованы структуры данных?
- Как строить интерфейсы, в том числе с пользователем и другим ПО?

Основным автором этого документа и впоследствии координатором остальных этапов жизненного цикла является *технический архитектор*. Это, как правило, специалист высочайшей квалификации с системным взглядом на вещи и богатым профессиональным опытом, одновременно разбирающийся в технических деталях и способный видеть проект в целом. Однако для разработки наиболее сложных и дорогостоящих систем целесообразно привлечение узких специалистов, например проектировщика баз данных и дизайнера пользовательского интерфейса.

Четкое построение архитектуры с продуманным разделением на подсистемы и модули позволяет перейти к выдаче заданий программистам. Обычно каждый из программистов отвечает за свой модуль. Он пишет программу и проводит первый этап ее тестирования и отладки — автономную отладку, называемую также модульным тестированием. После этого осуществляется *сборка* версии программного обеспечения, и оно передается специалистам по тестированию в отдел, который в некоторых компаниях называют Quality Assurance. Не очень хорошо, если на роль тестировщиков назначают программистов. Увы, талантливые разработчики, хорошо владеющие языком Си или Java, часто не очень хорошо владеют русским...

ЗАМЕЧАНИЕ

> Специалист по тестированию и программист должны обладать разными наборами психологических качеств. Если программист должен быть оптимистом и мыслить конструктивно, то для специалиста по тестированию

полезны педантичность, внимание к деталям вплоть до мелочности, придирчивость, пессимизм.

Специалисты по тестированию выполняют тестирование программного продукта и в случае необходимости отправляют его на предыдущий этап для исправления ошибок (а в тяжелых случаях — на частичное или полное перепроектирование).

ЗАМЕЧАНИЕ

Тестирование и отладка представляют собой наиболее трудоемкий этап при создании программных средств — до 55–60 % от общей трудоемкости, тогда как само написание программы — лишь10 %.

После нескольких итераций по тестированию и отладке специалисты по тестированию дают добро на выпуск официальной версии программного обеспечения. Однако для современных систем программы как таковой недостаточно. Важным этапом является документирование, причем разработка программной документации для больших систем — весьма трудоемкий процесс. Ее также лучше не поручать программистам. Специалист, занятый документированием программ, называется *техническим писателем*. Он должен обладать весьма редким набором качеств: с одной стороны, уметь прозрачно, однозначно и понятным для неподготовленного пользователя языком описывать систему, с другой — понимать технические стороны ее реализации. Особенно важно это, если учесть, что программная документация не ограничивается инструкцией пользователя, а может включать:

- руководство пользователя;
- руководство программиста — для будущих доработок, сопровождения, интеграции с другим ПО и пр.;
- руководство по установке и настройке;
- методику испытаний;
- учебник по программной системе.

Иногда к написанию документации подключают системных аналитиков и технического архитектора. Отдельной проблемой является актуализация документации — приведение версий руководств в соответствие версиям программы .

ЗАМЕЧАНИЕ

Документирование может выполняться одновременно с разработкой и тестированием.

Помимо основных процессов жизненного цикла существует и ряд

обеспечивающих — например, руководство проекта (менеджмент) должно проходить сквозной линией от его начала до конца. Ошибки при проектировании также обходятся весьма дорого. К сожалению, при плохом проектировании может возникнуть ситуация, когда модули рождаются хаотично, интерфейсы не продуманы, все держится на честном слове. Об этом можно сказать так: «Если бы строители строили здания так же, как программисты пишут программы, первый залетевший дятел разрушил бы цивилизацию». В случае выбора неверных платформ, языков и т. д., что может быть выявлено лишь на этапе тестирования, возникает необходимость возврата на этап проектирования.

Затем наступает момент передачи заказчику программного продукта, что должно быть оформлено актом сдачи-приемки. Это весьма интересный этап, на котором начинаются взаимные претензии и конфликты. Заказчик восклицает: «Что вы наделали?! Нам нужно было совсем другое! Платить не будем!». Здесь самое время предъявить ему его подпись на техническом задании. К сожалению, поскольку в ТЗ невозможно использовать лишь строгие формальные нотации, отказавшись от естественного языка, а ему присущи такие неустранимые свойства, как неточность и неоднозначность, споры и различное истолкование одних и тех же фраз спецификации неизбежны.

Однако предположим, что споры урегулированы и заказчик соглашается принять продукт. Теперь необходимо установить и настроить его — это может быть трудоемким процессом, если, например, устанавливается корпоративная информационная система или система автоматизации проектирования с разделением доступа на десятках и даже сотнях компьютеров в большой организации. Обычно эту работу выполняют системные администраторы.

Если программой будут пользоваться большое число сотрудников, необходимо спланировать и организовать их обучение работе с ней. К этому желательно привлекать не программистов, профессиональным пороком которых, увы, часто является пренебрежительное отношение к конечным пользователям, а обладающих необходимой подготовкой и навыками преподавателей.

ЗАМЕЧАНИЕ

> Взгляд программистов свысока на людей, не являющихся профессионалами в информационных технологиях, не только неэтичен, но и показывает невысокий уровень развития самого программиста, ведь он сам может совершенно не разбираться в том, в чем пользователь разбирается прекрасно, будучи, например, врачом или инженером по ракетным двигателям.

Т. Павловская пишет: «Часто программисты упрямо считают, что их основной целью является изобретение новых изощренных алгоритмов, а не

выполнение полезной работы, и напоминают этим печально известную точку зрения “магазин — для продавца, а не для покупателя”. Надо помнить о том, что программист в конечном счете всегда работает для пользователя программы и является членом коллектива, который должен обеспечить создание надежной программы в установленный срок» [12]. К этому нечего добавить!

После внедрения ПО начинается процесс его эксплуатации и сопровождения. Для многих массово используемых программных систем необходима организация консультаций пользователей как по телефону «горячей линии», так и средствами современных компьютерных сетей. Консультант должен обладать набором специфических психологических качеств: быть корректным, вежливым, владеть навыками общения и получения информации из потока речи огорченного пользователя, — но при этом разбираться в технических деталях программы.

В случае выявления ошибок в процессе эксплуатации — а это, увы, происходит довольно регулярно — разработчик должен за свой счет исправить их, вернувшись на соответствующий этап. И будет лучше для всех, если это произойдет на этапах автономной отладки и тестирования.

Современные сложные программные комплексы эксплуатируются на протяжении нескольких лет, иногда даже десятков лет. Неудивительно, что возникает потребность в доработках, внесении изменений, адаптации к меняющимся условиям производства, законодательству и пр. В этом случае может быть выпущено частное техническое задание, которое согласовывают с заказчиком, и происходит возврат далеко назад по этапам жизненного цикла — к постановке задачи. При длительной эксплуатации затраты на сопровождение вполне могут в несколько раз превысить затраты на создание первоначальной версии программы.

Контрольные вопросы

1. Хорошо или плохо, когда в процессе создания программы принимает участие значительное число людей?
2. Сколько лет занимает жизненный цикл современной сложной программной системы?
3. На каком этапе жизненного цикла программы цена ошибки наиболее высока?
4. Каков наиболее трудоемкий этап жизненного цикла программных средств?
5. В чем состоят задачи системного аналитика?
6. Техническое задание, спецификация и архитектура системы — это один и тот же или разные документы?

7. Зачем нужны формальные нотации при написании технического задания?
8. Должны ли программисты писать техническую документацию?
9. Какими психологическими качествами должны обладать программисты и специалисты по тестированию? Хорошо ли, когда программисты сами занимаются тестированием программ?
10. Что требует наибольших затрат — разработка или сопровождение программных систем?
11. Программы для программистов или программы для пользователей? Каково ваше мнение?

О стиле программирования и красоте программ

Красота всегда права.

Бабур

К настоящему моменту читатель уже имел возможность увидеть достаточное количество программ для того, чтобы заметить, что они значительно отличаются друг от друга. Одна программа выглядит так, другая — иначе, каждая несет на себе отпечаток личности автора.

Какие-то программы выглядят более привлекательными, другие — менее. А можно ли говорить о красоте программы? Кто-то из читателей слышал, вероятно, словосочетания «красивое решение», «изящный метод» и даже «красивая формула». Уместно ли это, допустимо ли эстетические критерии применять к технике или это удел таких видов искусства, как живопись, музыка, литература? Девизом выдающегося авиаконструктора А. Н. Туполева было: «Некрасивая машина никогда не будет летать». И действительно, красивые, гармоничные ракеты, самолеты, корабли часто обладают выдающимися техническими характеристиками.

Соразмерность, гармоничность облика технического изделия, машины может быть поверена, в частности, с помощью так называемого правила золотого сечения. Так называется деление отрезка на две части, при котором отношение длины меньшего получившегося отрезка к длине большего равно отношению длин большего к исходного отрезков.

Однако так можно оценить изделия, имеющие геометрическое изображение, — чертеж, эскиз, трехмерную модель. А как отличить красивую программу от не являющейся таковой? Возможно, для того чтобы понять, что такое красивая программа, можно пойти от противного. Давайте разберемся, какую программу мы ни в каком случае не признаем

красивой. Излишне использующую память компьютера. Выполняющую бессмысленные действия. Неряшливо написанную, плохо воспринимаемую глазом человека, в которой неясна логика. Если две программы — из десяти и ста строк — семантически эквивалентны и делают одно и то же, ясно, что более лаконичная программа красивее. Избыточная сложность — это порок для программы.

ЗАМЕЧАНИЕ

> В программировании вполне уместно применение совета Эйнштейна «сделать это как можно более простым, но не проще».

Сравним два варианта программы, ставшей красной нитью данного книги, — программы суммирования нечетных чисел в последовательности, на языке Си.

```
#include <stdio.h>
main() {int f[2000],i,p,n;
n=0;p=10;for(i=0;i<10;i++)
scanf("%d",&f[i]);
for(i=0;i<10;i++)
{if(f[i]%2)n+=f[i];}
printf("результат %d",n);}
```

```
/* Суммирование нечетных */
#include <stdio.h>
#define N 10 /* количество обработанных чисел
*/

int main()
{
  int a,i,s=0; /* начальный сброс суммы */

  printf("Подсчет суммы нечетных\n");
  printf("Введите %d целых чисел:\n",N);
  for (i=0;i<N;i++)
  {
    printf(">");scanf("%d",&a);
    if (a%2==1) /* проверка на нечетность*/
      s+=a;
  }

  printf("Сумма нечетных =%d\n",s);
  return 0;
}
```

Обе они синтаксически и семантически корректны, то есть компилируются стандартным компилятором Си и решают поставленную задачу — подсчитывают сумму нечетных членов числовой последовательности. В то же время программы эти явно различаются по эстетическим критериям.

Что можно сказать о программе, расположенной слева? Она явно неряшлива. Строки сливаются, о логической структуре алгоритма трудно на первый взгляд сказать что-то определенное. Кстати, написать ее без ошибок в данном виде — несмотря на незначительный размер — у автора с первого раза не получилось, и это тоже следствие недочетов ее стиля. Программа не снабжена комментариями, и это затрудняет ее понимание.

Для выявления нечетности числа используется трюк, эксплуатирующий ту особенность языка Си, что в условном операторе `if ()` допустимо не только применять булевы логические значения, но и писать произвольное целое, причем ноль воспринимается как ЛОЖЬ, а любое другое число — как ИСТИНА. С одной стороны, это вроде бы повышает эффективность и свидетельствует о профессиональном знании языка, но с другой — затрудняет понимание программы. Текст программы не структурирован, отступы и пустые строки не применяются. Программа использует для хранения вводимых чисел массив, причем в памяти резервируется 2000 ячеек — с большим запасом, хотя реально используется лишь 10. Вероятно, так сделано на всякий случай, но примером правильного использования памяти явно не является. Кстати, 10 в данной программе — магическое число, встречающееся в ней несколько раз. В подобном случае правильным является использование именованной константы или макроопределения, которые затем достаточно изменить один раз, например, в самом начале программы. Имена переменных `n`, `p`, `f` ничего не говорят об их смысле, скорее, напротив — нарушают негласные традиции программирования, ведь в данной программе `n` используется для хранения суммы, `p` — количества обрабатываемых чисел, а `f` — в качестве имени массива целых. Традиционным и, соответственно, облегчающим понимание является использоние для суммы имени `s`, для количества чисел — `n`. В языке Си `f` — форматный символ для чисел с плавающей точкой, поэтому это имя для целочисленного массива — неудачное. Вообще, понятие стиля программирования включает и правила именования переменных и констант. Существует несколько известных рекомендаций и даже стандартов (так называемая венгерская нотация). В любом случае хорошо, когда переменная говорит сама за себя — имя подсказывает ее содержание и смысл ее использования. Итак, с позиций восприятия программы человеком расрположенная слева программа — не образец стиля.

ЗАМЕЧАНИЕ

> В книге [12] приводится пример корректной программы на Си, которую практически невозможно понять человеку. Она приводится в приложении А.

Рассмотрим теперь алгоритм программы, потратив некоторые усилия, чтобы разобраться. Он не вызывает восторгов. Сначала отдельным циклом производится ввод чисел в память. Затем вторым проходом с помощью еще одного оператора цикла выполняются перебор этих значений и их обработка. При этом более ни для каких целей в программе массив не используется, и возникает вопрос: а надо ли вообще накапливать вводимые числа в памяти? Нельзя ли совместить эти действия в одном цикле? Программа недружественна к пользователю. После запуска он видит перед

собой пустой экран с мигающим курсором, и ему остается лишь догадываться, чего от него ждет программа и вообще какую задачу она решает. В конце выводится некий результат, при этом пользователь может так и не понять, что было выполнено и зачем. Нет, эта программа не может считаться эстетичной — ни с позиций эффективности, ни с позиций алгоритма, ни с точки зрения внешнего вида.

Посмотрим теперь критическим взглядом на программу в правой части. Благодаря использованию отступов и пустых строк ее логическая структура довольно хорошо просматривается (в практике программирования существует несколько рекомендуемых стилей использования отступов — читатель может выбрать любой из них, главное, чтобы они были). Первой строкой идет комментарий, поясняющий предназначение программы (а при запуске она информирует об этом же пользователя). Память не засоряется массивом вводимых данных — здесь он просто не нужен. При этом программа легко перенастраивается на любую длину последовательности чисел изменением значения `N` в третьей строке. Ключевые моменты также пояснены в комментариях.

Кстати, дадим некоторые комментарии по поводу комментариев в программах. Чересчур большое количество, причем неуместных, комментариев тоже может испортить программу, в отличие от масла в каше из поговорки. Когда надо написать комментарий, а когда он лишний? Необходим комментарий в начале каждого программного модуля — в нем принято давать краткую справку о назначении и основных особенностях программы, а также указывать авторство и дату внесения изменений. Нужен комментарий перед каждой функцией (процедурой, классом, другой самостоятельной программной единицей), кратко поясняющий задачи, решаемые функцией, и ее аргументы. Каков принцип комментирования других строк программы? Все ключевые и не вполне очевидные моменты алгоритма должны быть прокомментированы. При этом комментарий должен быть содержательным, а не дублировать текст программы. Рассмотрим пример плохого комментария:

```
if (a[i]%2) s+=a[i];  // прибавление к s a[i]
```

Ясно, что пользы от него немного, скорее он способен вызвать раздражение. Рассмотрим другой вариант:

```
if (a[i]%2==1) // если число нечетное, остаток ==1
      s+=a[i];  // => добавляем очередное число к сумме
```

Этот гораздо более информативен.

По окончании выполнения программы пользователь получает осмысленное сообщение о том, что получено в результате, а именно, что

это — сумма нечетных чисел. Можно также отметить, что в программе — несмотря на то что компилятор пропускает и вариант, приведенный слева, — аккуратно используются декларирование целочисленного возвращаемого функцией `main` значения и оператор возврата в операционную систему значения `0` — `return 0`, что соответствует системному соглашению. В результате программу можно охарактеризовать так:

- ее легко понять;
- ее легко изменить в случае необходимости, например масштабировать количество обрабатываемых чисел;
- алгоритм программы не производит лишних операций;
- программа не тратит лишнюю память ЭВМ;
- ею удобно пользоваться.

Итак, мы можем сказать, что второй вариант программы — выполняющий то же самое, что и первый! — значительно более эстетичен.

ЗАМЕЧАНИЕ

> Вообще говоря, критерии эффективности, удобства пользования и понятности являются до определенной степени противоречивыми. Искусство программирования состоит, в частности, в том, чтобы найти правильный баланс между ними.

В современных условиях программные комплексы, достигающие весьма значительной длины — миллионов и даже десятков миллионов строк, не создаются программистами-одиночками. В этом участвуют десятки человек. Бывает, что программу, ранее написанную одним автором, передают для дальнейшего сопровождения другому. В этих условиях понятность программы приобретает первостепенное значение. А понятность определяет и удобство внесения в программу изменений и модификации. Многим программистам известен эффект, когда даже собственная программа через, скажем, полгода становится для автора не вполне узнаваемой, и иногда приходится вспоминать некоторые особенности ее устройства и детали алгоритма.

ЗАМЕЧАНИЕ

> Языки программирования используются сейчас не только для общения человека с ЭВМ, но и для общения человека с человеком.

Для лучшего понимания программы сначала ее автором, а затем последующими читателями и систематичности разработки можно рекомендовать следующий подход. Начинать писать программу со спецификации: укажите ее назначение, интерфейс, требования к ней. При

этом большая часть текста спецификации сначала идет как комментарий в тексте программы. А проект архитектуры со спецификацией интерфейсов процедур — в комментарии заголовков процедур. Затем постепенно «костяк» наполняется «мясом» программы.

В современном мире программы «живут» (создаются и используются) годами, претерпевают изменения, дорабатываются, совершенствуются. Нетрудно понять, насколько важны поэтому следующие характеристики качественной программы:

- функциональность;
- надежность;
- удобство использования;
- сопровождаемость и модифицируемость;
- переносимость и совместимость.

Никлаус Вирт, автор языков Паскаль, Модула и Оберон, ввел даже понятие *грамотного программирования* (literate programming), которое еще более приближает создание программ к написанию литературного произведения с соблюдением ряда канонов построения, которые можно назвать эстетическими.

ЗАМЕЧАНИЕ

> В литературе [7] встречается и другое понимание термина «стиль программирования», более близкое к смыслу термина «модель вычислений».

Автор языка программирования C++ Бьерн Страуструп пишет: «Вопрос “как писать хорошие программы…” напоминает вопрос “как писать хорошую… прозу?”. Есть два совета: “знай, что хочешь сказать” и “тренируйся, подражай хорошему стилю”. Оба совета годятся как для C++, так и для… прозы, и обоим одинаково сложно следовать» [19].

Стиль написания программ влияет также на возможность внесения ошибок — и удобство их поиска и исправления. Из-за этого хорошее оформление, красота программ становятся весьма серьезными факторами индустриального значения. Можно привести замечательный пример. Одним из первых документов, появившимся в самом начале проектирования истребителя пятого поколения F-35, стали «Правила написания бортовых программ на языке Си». Этот факт наглядно иллюстрирует важность программ при реализации сложных проектов и первостепенное значение стиля программирования.

Контрольные вопросы

1. Что такое красота программы?

2. Лучше или хуже работает красивая программа?
3. Удобнее ли сопровождать красивую программу?
4. Важен ли стиль программирования при разработке сложных программных комплексов? Нужна ли стандартизация стиля программирования? Приведите пример.
5. Что входит в понятие «качественная программа»?

Ошибки в программах и как с ними бороться

Человеку свойственно ошибаться.

Закон природы

Исправляя одну ошибку в программе, программист вносит две новых.

Народная мудрость

Как уже говорилось, современные программные комплексы достигли невероятного уровня сложности. Размер исходных текстов операционной системы Windows XP составляет 45 млн строк. Неудивительно, что, например, в Windows 98, уже практически вышедшей из употребления, до сих пор официально признано наличие нескольких тысяч ошибок.

ЗАМЕЧАНИЕ

Честно говоря, вообще непонятно, как при подобной сложности современное программное обеспечение хоть как-то работает. Это практически чудо (созданное умом и руками человека)!

При этом роль компьютеров и, соответственно, программного обеспечения в нашей жизни все более возрастает. Компьютеры повсюду — в телефонах, автомобилях, стиральных машинах, в поликлинике, билетной кассе и пр. От так называемых *критических приложений* (медицина, автоматизированные системы управления технологическими процессами на производстве, атомные станции, авиация, и пр.) во многих случаях зависят человеческие жизни. Как же добиться того, чтобы программы содержали как можно меньше ошибок (известная в программистских кругах шутка, в которой лишь доля шутки, гласит: каждая полезная хоть чем-то программа содержит по крайней мере одну ошибку)? Могут быть использованы два пути:

1. Сделать так, чтобы при разработке программы в нее вносилось меньше ошибок, иными словами, усовершенствовать процесс создания программ.

2. После появления программы провести проверку, выявить и устранить найденные ошибки. При необходимости процесс повторять до достижения минимально допустимого уровня качества.

Начнем рассмотрение со второго, наиболее активно используемого в настоящее время подхода.

Прежде всего определимся с базовыми понятиями. Существует ряд терминов, относящихся к проблеме обеспечения качества программ: «отладка», «тестирование», «испытания», «верификация», «валидация». Под отладкой понимается процесс, позволяющий получить программное обеспечение, функционирующее с требуемыми характеристиками в заданной области входных данных. Иными словами, *отладка* — этап разработки, на котором устраняются недостатки программного обеспечения. *Тестирование* программного обеспечения — вид деятельности, направленный на обнаружение ошибок (несоответствий, неполноты, двусмысленностей и т. д.) в программе. Отладка не является разновидностью тестирования, хотя эти два вида деятельности очень тесно связаны и обычно рассматриваются совместно.

Верификация программного обеспечения — более общее понятие, чем тестирование. Целью верификации является достижение гарантии того, что верифицируемый объект соответствует требованиям и удовлетворяет проектной документации и стандартам. Таким образом, можно сказать, что тестирование является составной частью процесса верификации.

Валидация программной системы — процесс, целью которого является доказательство того, что в результате разработки системы мы достигли целей, которых планировали достичь благодаря ее использованию. Иными словами, валидация — это проверка соответствия системы ожиданиям заказчика.

Помимо поиска ошибок в собственно исходных текстах программ имеет смысл поиск и исправление ошибок в других артефактах жизненного цикла. Например, ошибка в спецификации может обойтись гораздо дороже, нежели ошибка во второстепенном программном модуле. Неправильно выбранные проектные решения, будучи выявленными уже на этапе опытной эксплуатации, способны отбросить разработку на месяцы назад. Сами наборы тестов ПО могут быть неполными и некорректными, что приведет к некачественному тестированию и выдаче неудовлетворительного продукта.

В связи с этим верификацию рассматривают в рамках следующей общей картины (рис. 3) [20].

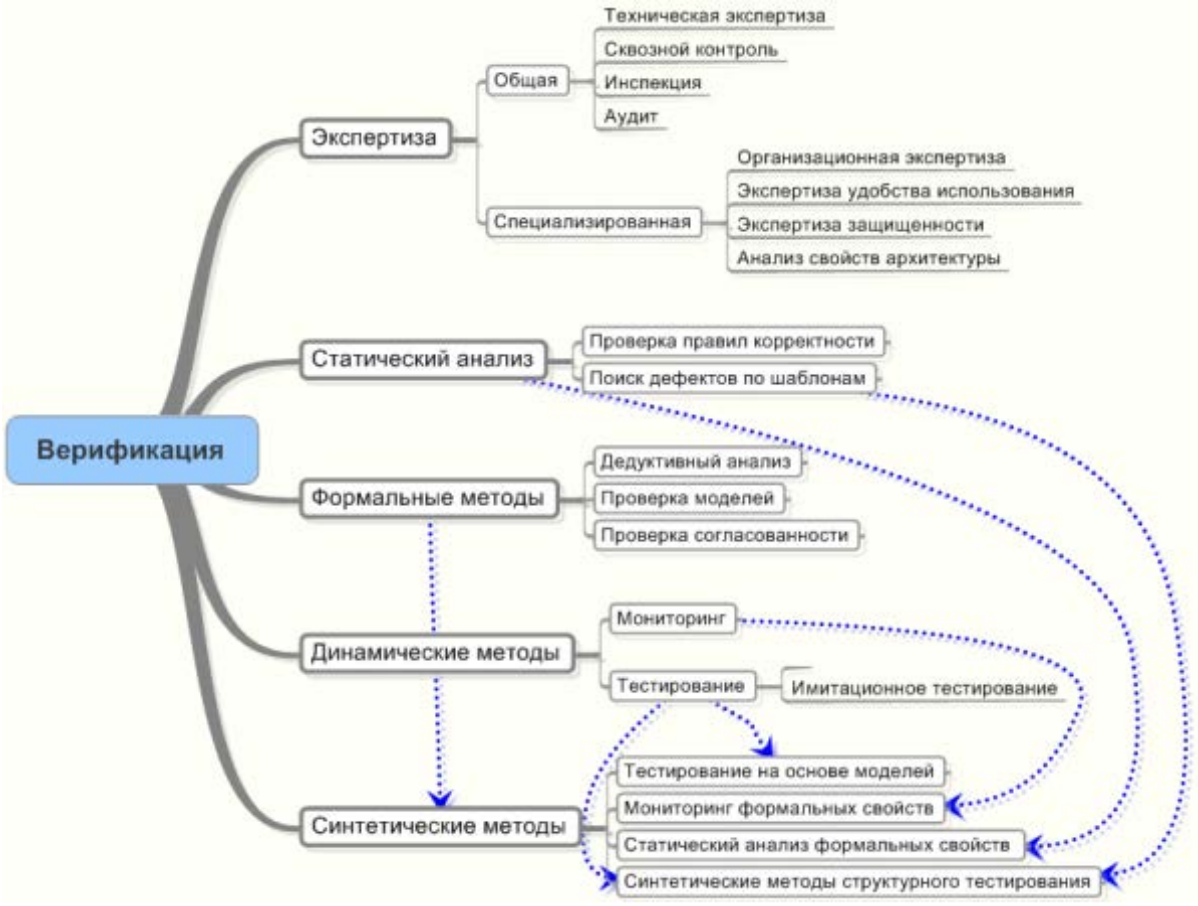


Рис. 3.

На практике сейчас наиболее широко применяют тестирование. При этом выделяют тестирование по методу «черного ящика», когда не углубляются во внутреннюю структуру программы, а лишь пытаются подать на ее вход различные исходные данные и сравнить результат работы с ожиданиями, и тестирование по методу «белого», или «прозрачного ящика», когда тесты составляют с учетом ветвлений программы таким образом, чтобы проверить ее как можно более полно.

ЗАМЕЧАНИЕ

> Существуют различные критерии полноты тестирования — покрытие всех операторов (при тестировании каждый исполняется хотя бы один раз), покрытие всех ветвей (в каждом условии пошли хоть раз и по ветви ТО, и по ветви ИНАЧЕ), покрытие всех маршрутов и пр.

Тестирование и отладка, как правило, представляют собой процесс, делящийся на этапы. Вначале выполняется *автономная отладка*, или *модульное тестирование*, когда по отдельности проверяют процедуры, классы, функции. Затем производится интеграция, или сборка, модулей, и осуществляется *совместная отладка*, или *интеграционное тестирование*. Если программная система является подсистемой более сложного

комплекса, например программно-аппаратного, проводится *комплексная отладка* с реальным или моделируемым оборудованием на специальных стендах (так происходит, например, в космической отрасли).

Исправлять ошибки в огромных и сложных программах — трудное и дорогостоящее занятие, причем цена ошибки тем выше, чем позже она обнаружена. Поэтому тестирование является хотя и необходимой, но весьма дорогой операцией. Повторим, что трудоемкость тестирования и отладки составляет до 60 % общей трудоемкости при разработке ПО.

Обеспечивает ли тестирование стопроцентную гарантию качества? Полный перебор всех возможных маршрутов исполнения программы при всех возможных сочетаниях исходных данных для промышленных программ занял бы миллионы, а то и миллиарды лет. Число маршрутов обычно на несколько порядков больше количества команд в программе. Никлаус Вирт в 1973 году сказал: «Экспериментальное тестирование программ может служить для доказательства наличия ошибок, но никогда не докажет их отсутствия» [9]. Ту же мысль высказал другой классик программирования, голландский ученый Эдсгар Дейкстра: «Тестирование программы может вполне эффективно служить для демонстрации наличия в ней ошибок, но, к сожалению, непригодно для доказательства их отсутствия» [21]. Ситуация ухудшается в случае тестирования параллельных программ, в которых необходимо учитывать аспекты взаимодействия программ и корректности разделения общих ресурсов. Еще сложнее случай систем управления реального времени, в которых система взаимодействует с непредсказуемым внешним миром и надо рассматривать сочетания параметров во все ключевые моменты времени.

Каков же выход? Существуют методы так называемой *формальной верификации*, или дедуктивные методики, способные математически строго доказать отсутствие ошибок. Однако их распространение и внедрение в практику существенно ограничивается следующими обстоятельствами. Эти методы применимы лишь к свойствам, которые могут быть описаны формально в рамках специальной математической модели. Для использования этих методов необходимо затратить значительные усилия на построение моделей. К тому же построить такие модели и проанализировать их могут только специалисты по формальным методам, которых не так уж много и услуги которых стоят довольно дорого. Построение формальных моделей на сегодня не автоматизировано, для этого всегда необходим человек, обладающий специфическим набором навыков и знаний в некоторых разделах математической логики и алгебры. Тем не менее в ряде областей, где ликвидация последствий ошибок в системе может оказаться чрезвычайно дорогой (авиация, космонавтика, опасные производства), формальные методы верификации используются. С их помощью можно обнаруживать сложные ошибки, практически не

выявляемые другими методами.

ЗАМЕЧАНИЕ

> Доказательная верификация была с успехом применена при программировании на языке Ада полностью автоматической линии парижского метро.

Можно отметить еще, что формализация требований и проектных решений возможна только при их глубоком понимании и поэтому вынуждает провести тщательнейший анализ этих артефактов в ходе совместной работы специалистов по формальным методам и экспертов в предметной области, а это само по себе весьма полезно для уменьшения количества ошибок.

Другой современный метод, обладающий существенным преимуществом за счет того, что он поддается автоматизации, — *проверка моделей* программ. Он активно развивается, и во внедрении его в практику программирования есть успехи — например, в лаборатории надежного программного обеспечения НАСА. Однако у этого метода также имеются два существенных недостатка. В нем проверяется не сама программа, а некоторая ее *модель*, соответственно, может оказаться, что модель неадекватна реальной программе. Кроме того, обычно человек строит модель вручную, что требует от него высокой математической квалификации.

Тестирование относится к так называемым *динамическим* методам верификации — в ходе них программа для проверки запускается на выполнение.

В то же время существует ряд *статических* методов, не связанных с прогоном программы, незаслуженно забытых и почти полностью игнорируемых в отечественных организациях, занятых созданием ПО. Программу, написанную одним человеком, может проверить другой человек. Подобный метод называется *экспертной оценкой*, или *просмотром кода*. Действительно, каким бы хорошим программист ни был, существует психологический эффект «замыленного» и «свежего взгляда». Организовать такую работу можно по-разному. Могут проводиться специальные совещания с групповым просмотром и комментированием наиболее ответственных фрагментов программы. Опытные и квалифицированные проограммисты или специалисты отдела качества могут проверять программы начинающих разработчиков (*инспекция*). Можно регулярно взаимно контролировать работу равных по статусу программистов. На практике метод показывает весьма высокую эффективность — до 90 % ошибок выявляется до компиляции программы [20].

ЗАМЕЧАНИЕ

> При данном подходе следует учитывать психологические аспекты: надо соблюсти принцип «не навреди», то есть не переборщить с недоверием и не ранить тонкую душевную организацию программистов.

В методе экстремального программирования (XP — eXtreme Programming) вообще принята методика *парного* программирования, когда за каждый модуль отвечают два человека и, соответственно, каждый находит и исправляет ошибки другого. Ведь, согласно известной поговорке, «одна голова — хорошо, а две — лучше».

Еще один метод проверки программы — так называемое *символическое выполнение*, когда осуществляется проход по модулю с записью всех присваиваний не в виде конкретных числовых значений переменных, а в символической форме. Например, анализируя

```
b=a+sin(x);
d=b*5;
res=d*d.
```

при символическом выполнении получаем `res` = $25a^2 + 50a \sin x + 25 \sin^2 x$. Полученный результат сравниваем со спецификацией модуля.

И совсем не вызовет конфликтов ситуация, когда рассмотрение текста программы будет возложено не на ревнивого коллегу или начальника, а на бездушную машину. Существует целый ряд инструментов автоматического статического анализа исходных текстов для программ на различных языках программирования — SPlint, Klockwork Insight, ESC/Java2, Boogie, Saturn, Calysto, отечественные SVaCE и Aegis и др. Они способны выявлять типичные ошибки и проводить контроль стиля программы на соответствие некоторому набору рекомендаций (*стандарту кодирования*), о роли которого говорится далее.

Поговорим теперь о влиянии процесса создания программы на результат. Начинающие программисты (например, студенты на занятиях) иногда пишут программы примерно следующим образом. Получив задание, немедленно усаживаются за компьютер и начинают набирать программу. Переменным дают первые попавшиеся имена, отражающие словарный запас и культуру автора, но не смысл. Когда программа зависает, безжалостно убивая первый порыв энтузиазма, делают перерыв, после которого уничтожают исходный вариант программы и все повторяют заново. Структурированием данных при этом не озадачиваются. Функции нервно удаляют и создают. Программу делают большим блоком, разбиение на модули выполняют лишь по настоятельному указанию преподавателя. Комментарии оставляют на потом, текст никак не форматируют.

Периодически высказывают сомнения в правильности работы компилятора, компьютера и операционной системы. Когда программа впервые доходит до стадии выполнения, в нее вводят произвольные значения, после чего экран на некоторое время становится объектом удивленного изучения.

ЗАМЕЧАНИЕ

Первый шаг при написании любой программы — сесть и подумать!

Будет ли качественным продукт, полученный в результате подобного процесса? Ответ на этот вопрос не вызывает сомнений.

Итак, при написании программы следует уделять должное внимание ее *проектированию*. И данные, и алгоритмы должны быть правильно структурированы. Необходимо применять универсальный принцип «разделяй и властвуй» для создания обозримых, прозрачных, легко отлаживаемых модулей.

Другой важнейший аспект, имеющий прямое отношение к теме данной книги, — используемый язык программирования. Одни языки изобилуют потенциально опасными конструкциями, чреватыми возможностью внесения ошибок, другие весьма строги. Статистические исследования показывают, что программы на Си в среднем содержат больше ошибок на 1000 строк, нежели программы на языках Ада или Оберон (а программы на ассемблере — больше, чем программы на Си). На Си вы можете написать:

```
for(;P("\n").R-;P("\"))for(e=3DC;e-
;P("_"+(*u++/8)%2))P("|"+(*u/4)%2);
```

В Паскале, Модуле или Обероне подобное невозможно. Другой хрестоматийный пример — уязвимость в Фортране:

```
DO 100 I = 1,10
```

Эта строка описывает циклическое выполнение последующего фрагмента программы до метки 100 в количестве 10 раз. Однако достаточно сделать ошибку в единственном символе и вместо запятой поставить точку (пробелы в Фортране игнорируются):

```
DO100I = 1.10
```

чтобы получить вместо цикла оператор присваивания некой переменной с именем `DO100I` значения 1,1.

Большое количество ошибок связано с неверным преобразованием типов данных и ручным управлением динамической памятью. Соответственно, преимущество с этой точки зрения имеют языки с автоматической сборкой мусора и строгим контролем типов на этапе компиляции.

Вообще можно отметить, что язык программирования должен

максимально способствовать правильному программированию, помогать развивать хороший стиль написания программ. В начале книги мы уже отмечали влияние языка на сам процесс мышления.

Другой важной проблемой для современных сложных программных комплексов является участие в их разработке десятков людей разных специальностей, уровня профессиональной подготовки, навыков и пр. На разных этапах жизненного цикла возникает недопонимание, например, между заказчиком и системным аналитиком, аналитиком и техническим архитектором, архитектором и программистами, программистами и специалистами по тестированию, чреватое ошибками. Было бы замечательно использовать интуитивно понятные человеку нотации (языки), позволяющие уменьшить количество ошибок, возникающих в силу подобного недопонимания (о роли визуальных методов разговор шел ранее). В настоящее время с этой целью применяются такие языки, как UML, DFD, BPML, IDEF0, IDEF1X и др. Если они при этом еще будут строгими, однозначными и поддаваться формальной верификации — вообще замечательно!

Еще одним подходом, позволяющим улучшить процесс разработки программ с точки зрения повышения надежности получаемого продукта, является так называемое *контрактное программирование*. При применении данного подхода каждые программный модуль, процедура, класс в ООП снабжаются формализованными *предусловием* и *постусловием*. Предусловие проверяется перед началом выполнения модуля и контролирует аргументы, тем самым обеспечивая корректность интерфейса, постусловие — при выходе из процедуры, позволяя проверить корректность работы модуля. В некоторых языках, например Eiffel, контрактное программирование является обязательным, в других — предоставляемой возможностью (Ада2012). На данные можно накладывать дополнительные условия — так называемые *инварианты*, система автоматически проверяет их и при выходе значения за указанные границы генерирует сообщение.

Ну и, наконец, наиболее радикальным методом повышения надежности программ — если уж человеку свойственно ошибаться — является воплощение в жизнь принципа *программирование без программистов* [22]. В этом случае по спецификации автоматически, без участия человека генерируется машинный код, готовый к загрузке в память ЭВМ. На первый взгляд это может показаться весьма удивительным и даже невозможным, но известны успешные примеры реализации данного подхода на практике! (Естественно, необходимым условием становится строгий, полный и понятный человеку язык спецификации.) Примером является технология ГРАФИТ/ФЛОКС, созданная в Научно-производственном центре автоматики и приборостроения им. академика Н. А. Пилюгина в ходе

работ по созданию многоразового космического корабля «Буран». Еще в одной подведомственной Федеральному космическому агентству России организации — МОКБ «Марс» внедрена в производство подобная методология разработки бортовых программ для космических аппаратов, причем автоматически генерируется не только программа, но и актуальная, соответствующая ее версии документация! Это позволило отказаться от услуг целого отдела бортовых программистов. Следует отметить, что в силу особой ответственности предметной области все вносимые изменения строго контролируются и утверждаются в системе электронного документооборота, интегрированной с системой контроля версий.

Среди актуальных направлений можно назвать также методы автоматизации тестирования программ с автоматической генерацией набора тестов, покрывающих все маршруты, методы model checking с автоматическим извлечением модели из исходного текста, проблемно-ориентированные языки, повышающие уровень абстракции, и др. В заключение отметим: хотя полностью безошибочных программ создать и невозможно, к этому, безусловно, надо стремиться!

Контрольные вопросы

1. Какие методы борьбы с ошибками в программах существуют?
2. Надо ли сесть и подумать, прежде чем писать программу?
3. В чем разница между верификацией и валидацией?
4. В чем разница между статическими и динамическими методами верификации программ?
5. Нужно ли верифицировать спецификации?
6. Что такое статический анализ программы? Что такое инспекции? Эффективны ли эти методы? Приведите примеры.
7. Что такое контрактное программирование? В каких языках программирования оно предусмотрено изначально?
8. Что такое метод проверки моделей программ (model checking)? Опишите его базовые идеи.
9. Какие виды отладки и тестирования программ вам известны?
10. Влияет ли используемый язык программирования на количество ошибок в программе?
11. Назовите причины, по которым формальная верификация редко используется в настоящее время?
12. Как вы относитесь к подходу «программирование без программистов»? Возможно ли его воплощение в жизнь? Перспективен ли он?

Заключение

Невозможно объять необъятное, и книга имеет ограниченный размер. За границами ее рассмотрения остались такие интересные и заслуживающие внимания языки, как Форт и Smalltalk, эквациональные Q и Golux, уникальный РЕФАЛ, созданный отечественным ученым В. Турчиным, функционально-ориентированные K и J, мультипарадигменный Oz…

Все же автор питает некоторую надежду на то, что ему удалось донести до читателя основной посыл, заключающийся в том, что языки программирования — захватывающая и бурно развивающаяся область с несколькими магистральными путями развития и парадигмами программирования, не сводящаяся лишь к всем известным императивным Паскалю, C# и Java.

Вместе с читателем мы спустились до «низкого» уровня языков ассемблера и затем поднялись на головокружительную абстрактную высоту Пролога. Рассмотрев исторические аспекты, увидели, «из какого сора» и зачастую случайно появлялись на свет завоевавшие затем мир языки.

Что можно сказать в заключение? Видимо, несколько слов о будущем, каким оно представляется автору.

Языки программирования — развивающаяся область одной из самых инновационных областей деятельности человека — информационных технологий. Какими станут языки программирования через пять лет? Десять? Займет ли достойное место Фортран в третьем десятилетии XXI века? «Заговорят» ли наконец ЭВМ на естественном языке? Не появится ли некая универсальная программирующая система, после чего всех программистов выкинут на улицу?

Автору хочется отметить следующие тенденции. Аппаратная часть, начиная с рубежа веков, стремительно развивается, магистральным путем развития стали многоядерные микропроцессоры — они уже являются нормой даже в смартфонах! В этих условиях приобретают особую актуальность проблемы параллельного программирования. Ведь хотя оборудование переросло ограничения архитектуры фон Неймана, большинство программистов традиционно проходят обучение в рамках императивной последовательной парадигмы. Это представляет собой, можно сказать, психологическое ограничение прогресса в области параллельных вычислений.

Касательно материала данной книги можно напомнить, что языки функциональной и логической парадигм — Лисп и Пролог — допускают легкое распараллеливание средствами системы без участия программиста

и возложения на него обязанностей по детальному продумыванию вопросов параллельной обработки.

Еще одна характерная особенность современных информационных технологий — массовое внедрение микроконтроллеров в бытовую технику, системы автомобилей и пр. Встроенные ЭВМ — сегодня наиболее многочисленный вид ЭВМ. Соответственно, необходимо создание разнообразного встроенного программного обеспечения. У встраиваемых приложений есть свои особенности, в частности, ограничения на ресурсы. Поэтому, видимо, в обозримом будущем ассемблер не уйдет из практики программирования.

Наконец, автор еще раз выражает уверенность в том, что визуальное программирование будет и дальше развиваться и завоевывать все более серьезные позиции в индустрии программирования.

И освоение, а может быть и развитие перспективных средств программирования — задача читателя, держащего в руках эту книгу. Успехов!

Список литературы

1. Современный компьютер: Сб. науч.-попул. Статей; Пер. с англ./Под ред. В.М.Курочкина; предисл. Л.Н. Королева — М: Мир, 1986.
2. Хофштадтер Д. Гедель, Эшер, Бах — эта бесконечная гирлянда. — Самара: Бахрах-М, 2000.
3. Тьюринг А. Может ли машина мыслить. — М.: Физматгиз, 1950.
4. Лорьер Ж.-Л. Системы искусственного интеллекта. — М.: Мир, 1991.
5. Ермаков И. Е. Лекции с обзором языков программирования.
6. Непейвода Н. Стили и методы программирования. Курс лекций. Учебное пособие. — М.: Интернет-университет информационных технологий, 2005.
7. Пентковский В. М. Язык программирования Эль-76. Принципы построения языка и руководство к пользованию. — 2-е изд., испр. и доп. — М.: Физматлит, 1989.
8. Хамби Э. Программирование таблиц решений. — М.: Мир, 1976.
9. Вирт Н. Систематическое программирование. Введение. — М.: Мир, 1977.
10. Кнут Д. Искусство программирования. В 3 т. — М.: Вильямс, 2012.
11. Керниган Б., Ритчи Д. Язык программирования Си. 2-е изд.— М.: Издательский дом «Вильямс», 2013.

12. C/C++. Программирование на языке высокого уровня / Т.А. Павловская — СПб.: Питер, 2011 .
13. Пустоваров В. И. Язык ассемблера в программировании информационных и управляющих систем. — М.: ДЕСС, 1998.
14. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог. — М.: Мир, 1990.
15. Тюгашев А.А. Графические языки программирования и их применение в системах управления реального времени. — Самара: Изд-во СНЦ РАН , 2009.
16. ГОСТ 19.701–90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. — М.: Госстандарт, 1991.
17. Коварцев А. Н. Автоматизация разработки и тестирования программных средств / Изд-во Самар. гос. аэрокосм. ун-та — Самара, 1999.
18. Харьковский З. Путеводитель автостопщика по потаенным знаниям // Компьютерра.- 2005. № 12. — С. 42–52.
19. Страуструп Б. Язык программирования C++. 3-е изд. — М.: Бином, 2011.
20. Кулямин В. В. Перспективы интеграции методов верификации программного обеспечения // Труды ИСП РАН. — 2009. — Т.16:-С.73-88,
21. Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978.
22. Паронджанов В. Д. Как улучшить работу ума. — М.: Дело, 2001.
23. Кауфман В. Языки программирования. Концепции и принципы. — М.: ДМК Пресс, 2011.
24. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация / Под общей ред. А. Матросова. — СПб.: Питер, 2002.
25. Абельсон Х., Сассман Д. Структура и интерпретация компьютерных программ. — М.: Добросвет, 2010.

Лабораторный практикум по языку Си

Примерный перечень лабораторных работ по языку программирования C/C++.

Лабораторная работа № 1. Знакомство с интегрированной средой разработки программ. Ввод, трансляция и выполнение простейшей программы на языке программирования Си. Анализ результатов (2 академических часа).

Лабораторная работа № 2. Типы данных. Математические вычисления с использованием языка программирования Си. Знакомство с функциями математической библиотеки. Основные операции ввода-вывода. Условные выражения. Написание программы вычисления не всюду определенной сложной функции (по вариантам) (6 академических часов).

Лабораторная работа № 3. Операторы цикла в языке Си. Табулирование значений функции из лабораторной работы № 2 (4 академических часа).

Лабораторная работа № 4. Передача параметров в функцию `main` из операционной системы. Написание программы подсчета суммы нечетных элементов числовой последовательности (тремя способами) (6 академических часов).

Лабораторная работа № 5. Работа с массивами и матрицами. Вывод содержимого матрицы по спирали (4 академических часа).

Лабораторная работа № 6. Написание программ сортировки матриц (по вариантам). Знакомство с отладчиком. Установка точек останова. Просмотр значений переменных в ходе исполнения программы (8 академических часов).

Лабораторная работа № 7. Низкоуровневые операции языка Си. Написание подпрограммы вывода содержимого ячейки в двоичном виде на экран с помощью битового маскирования и сдвига (4 академических часа).

Лабораторная работа № 8. Работа со строками в языке Си. Оператор выбора языка Си. Преобразование вузовской оценки из баллов в строку и наоборот (4 академических часа).

Лабораторная работа № 9. Шифрование файла. Написание программы шифрования и декодирования текстового файла (по вариантам) (4 академических часа).

Лабораторная работа № 10. Разработка базы данных с вводом, хранением, поиском и заменой информации о студентах на основе структур и текстовых файлов (12 академических часов).

Лабораторная работа № 11. Введение в объектно-ориентированное

программирование. Написание программы работы с геометрическими фигурами (по вариантам) (6 академических часов).

Лабораторная работа № 1. Простейшая программа на языке Си

Задание: написать и запустить программу вывода на экран компьютера приветственного сообщения (например — «Здравствуй, мир!»).

Возможный вариант решения:

```
#include <stdio.h>
int main()
{
  printf ("Здравствуй, мир !\n");
  return 0;
}
```

Разработка программ проводится обычно в интегрированной среде разработки программ (Integrated Development Environment, IDE).

Принцип работы с интегрированной средой базируется на концепции “редактор – транслятор - отладчик”. При программировании тех или иных задач работа программиста представляет собой, как правило, итеративный (циклический) процесс, включающий в себя следующие основные стадии:

- набор исходной программы в текстовом редакторе;
- трансляция (компиляция);
- компоновка (линковка);
- запуск на выполнение;
- отладка.

В обычном варианте при прохождении всех перечисленных стадий необходим последовательный вызов средствами операционной системы соответствующих программ: текстового редактора, транслятора, компоновщика, отладчика. При работе с интегрированной средой разработчика (IDE, Integrated Development Environment) программист общается только с ее собственным, дружественным интерфейсом, а вызов необходимых служебных программ и модулей осуществляется автоматически. Будем вести изложение на основе достаточно старой, но хорошо апробированной и великолепно зарекомендовавшей себя, вдобавок распространявшейся уже достаточно давно бесплатно IDE – Borland C++ (Turbo C++), в других средах «горячие клавиши» и пункты меню могут отличаться, но принципы работы сохраняются. Интерфейс среды разработчика Borland C++ построен на основе меню и окон. Для редактирования каждого текстовых файла открывается одно окно, при

этом возможен перенос фрагментов текста из одного файла в другой через буфер оболочки. При трансляции открывается специальное окно трансляции, существует специальное окно для вывода сообщений об ошибках, и т.д. Навигацию по окнам можно осуществлять с помощью нажатия клавиш Alt+<номер требуемого окна>. С помощью нажатия клавиши F5 возможно развернуть активное окно на весь экран. Обратное изменение возможно с помощью еще одного нажатия F5. Alt+F3 приводит к закрытию текущего окна. Ctrl+F5 включает режим изменения размеров и перемещения окон.

Доступ к большинству функций интегрированной системы возможен через главное меню оболочки, которое можно вызвать с помощью клавиши F10.

Перемещаться по меню можно, используя клавиши управления курсором – клавиши со стрелками. При нажатии на клавишу Enter происходит выбор соответствующего пункта в меню. Пункты главного меню имеют связанные с ними так называемые “выпадающие” подменю, например, с пунктом главного меню “File” связано выпадающее меню, в котором перечисляются основные операции с файлами – создание, открытие, сохранение (запись), и т.д.

Для выхода из меню (равно как и для отмены какого-либо действия), можно использовать клавишу Esc. Некоторые пункты выпадающих подменю главного меню могут иметь рядом с названием пункта многоточие (“. . .”) либо знак треугольника. Это означает, что при выборе данного пункта происходит открытие дополнительного окна диалога или нового подменю. Некоторые пункты меню можно вызвать с помощью так называемых “горячих клавиш” непосредственно, например, в процессе набора текста программы во встроенном в IDE текстовом редакторе. Эти клавиши (или их комбинации) встречаются в соответствующих пунктах подменю. Например, с помощью нажатия клавиши F3 можно вызвать стандартный диалог открытия файла. При нажатии комбинации “Alt+X” происходит быстрый выход из среды разработчика. Пункты подменю можно вызывать и с помощью комбинации клавиш Alt и буквы, подчеркнутой в названии соответствующего пункта.

Работа с меню, кроме использования клавиатуры, возможна с использованием манипулятора “мышь”.

В нижней части экрана при работе с интегрированной оболочкой Borland C++ располагается так называемая строка состояния. Она выполняет следующие функции.

- напоминает о клавишах, которые можно использовать в данный момент.

- позволяет выполнять те или иные действия с помощью манипулятора “мышь”. Достаточно указать название действия и щелкнуть на нем левой кнопкой мыши.
- информирует пользователя о действии, выполняемом в настоящий момент (например, сохранение файла).
- предлагает советы и рекомендации по выбранным пунктам меню или диалога.

Система имеет развитую систему так называемой контекстно-зависимой (то есть “понимающей”, в каком режиме происходит работа и предлагающей помощь именно по нему) подсказки. Есть соответствующий пункт в главном меню, есть и соответствующая “горячая клавиша” - F1. Есть и еще одна полезная особенность. При подводе курсора в окне текстового редактора к набранному оператору или стандартной функции языка Си и нажатии комбинации клавиш Ctrl+F1 выводится информация о данном ключевом слове.

Отладка (debugging) программ подразумевает процесс выявления и устранения ошибок в написанных программах. Как гласит известная в программистских кругах аксиома, “каждая программа содержит по крайней мере одну ошибку”. Еще одно шутливое изречение гласит, что “устраняя одну ошибку в программе, мы вносим три новых”. И, хотя в этих утверждениях есть только доля шутки, данный факт не дает нам оснований отказаться от устранения ошибок в программах.

Встроенный в IDE Borland C++ отладчик обладает возможностями по установке точек останова (Breakpoints) в тексте программы, просмотра и модификации при необходимости текущих значений переменных.

Для успешной работы с оболочкой необходимо правильное задание так называемых параметров рабочего окружения. В частности, при работе с системой Borland C++ используются стандартные файлы, и необходимо указать их расположение (каталог на диске). Доступ к этому режиму осуществляется через пункт главного меню Options, подпункт Directories. В частности, при программировании на языках C/C++ используются так называемые заголовочные файлы, и строка в тексте программы

```
#include <stdio.h>
```

означает, что в текст необходимо вставить файл `stdio.h`, а искать его надо в некотором стандартном каталоге. Этот каталог надо явно задать в оболочке, в разделе Include Directories.

Еще одна важная группа файлов – служебные библиотеки, включающие наборы стандартных функций, таких например, как сравнение строк, математические и др. Их расположение задается в разделе Library Directories.

Наконец, каталог, в который будет помещаться готовая к исполнению программа с расширением .exe, задается в разделе Output Directory.

Итак, создадим новое окно текстового редактора путем выбора в главном меню пункта File, подпункта New в выпадающем меню. Откроется новое окно текстового редактора. Наберем в нем нижеследующий текст.

```
#include <stdio.h>
int main()
{
  printf ("Здравствуй, Мир!\n");
  return 0;
}
```

Эта программа выводит на экран компьютера текстовое сообщение "Здравствуй, Мир !".

Разберем текст созданной программы. Первая строка содержит одну из так называемых директив препроцессора. Препроцессор – это специальная программа, обрабатывающая тексты на языке Си до того, как они передаются транслятору языка программирования, и осуществляющая некоторые предварительные действия в соответствии с директивами, которые для отличия от собственно операторов языка Си начинаются с символа "#". Директива `#include` в частности, указывает на необходимость включения в программу стандартного заголовочного файла `stdio.h` для подключения к программе библиотеки стандартных функций ввода-вывода (STandarD Input - Output). Это необходимо для возможности дальнейшего использования библиотечной функции `printf`.

Следующая строка содержит заголовок идущей далее в тексте функции `main`. Надо сказать, что все программы на языке Си являются наборами функций. Функция определяется как некоторый модуль, имеющий в общем случае некоторые аргументы, записываемые в круглых скобках, и могущий возвращать какое-то значение. Одни функции могут вызывать на выполнение другие, и т.д. При этом система начинает выполнение любой программы на языке Си с функции со стандартным именем – `main`. Соответственно, каждая программа на языке Си должна включать ровно одну функцию с именем `main`. Перед названием функции написано "`int`", что означает, что функция возвращает целое число. Поскольку функция `main` – основная функция программы, то ее возвращаемое значение подразумевает возвращаемое в операционную систему программой значение. Существует соглашение, что если

программа завершается нормально, без возникновения нештатных ситуаций, то она возвращает ноль, в противном случае – ненулевое значение. После названия функции `main` идут пустые круглые скобки, что означает, что наша программа не обрабатывает никаких входных значений, передаваемых через командную строку при вызове программы на выполнение (а при вызове с помощью команды Run интегрированной оболочки есть специальный пункт Arguments в соответствующем подменю).

В следующих строках после заголовка идет собственно тело функции, в котором содержатся действия, которые она выполняет, то есть операторы языка Си и вызовы функций. Тело функции ограничивается фигурными скобками – `{` и `}`.

Наша функция вызывает библиотечную функцию форматного вывода на стандартное устройство (в данном случае экран дисплея)-функцию `printf`. Аргументом данной функции является текстовая строка, которые в языке Си принято заключать в двойные кавычки. В конце строки можно заметить символы `“\n”`, это специальная последовательность символов (на это указывает знак ‘\’), означающая, что после вывода строки на экран надо перейти к новой строке. Точка с запятой – стандартный разделитель языка Си, разграничивающий отдельные действия в тексте функции.

Следом идет оператор `return 0`. Этот оператор возвращает некоторое значение в качестве результата выполнения функции. В данном случае мы передаем ноль операционной системе как свидетельство того, что функция завершается нормальным образом.

Последняя строка программы – закрывающая фигурная скобка, указывающая на конец функции `main`.

Теперь необходимо осуществить следующие действия:

1. Сохранение текста программы в некотором файле на диске.
2. Трансляция текста программы на языке Си в машинные коды.
3. Компоновка объектного модуля программы с модулем стандартной библиотеки.
4. Запуск готовой исполняемой программы.

Для выполнения первого этапа нажмем кнопку F2. Откроется стандартное окно сохранения файла. В нем можно видеть содержимое текущего каталога на диске, перемещаться по каталогам путем выбора соответственно “..” для перемещения на уровень вверх по дереву каталогов, и название поддиректории для спуска вниз по дереву, задавать имя сохраняемой программы. По умолчанию IDE предлагает для

программы название “NONAME00.C”. Исправим его, например, на “first.c”. Необходимо обратить внимание на то, что в каталоге показываются не все файлы, а только те, которые имеют расширения, задаваемые соответствующим пунктом диалога. И при сохранении файла предлагается одно из стандартных расширений - .c, .h или .cpp для программ на языке C++.

Указав имя файла и расширение .c, которое покащывает системе, что данный файл содержит текст программы на языке Си, выбираем пункт Ok диалога сохранения.

Этапы 2-4 можно выполнять по отдельности, но мы воспользуемся возможностью отдать интегрированной среде команду “сделай все необходимое и запусти программу” с помощью нажатия комбинации клавиш Ctrl+F9. Система тогда автоматически выполняет трансляцию, компоновку (о чем свидетельствует специальное окно, возникающее на экране), и в случае отсутствия ошибок, запускает программу на выполнение.

Не стоит удивляться, если вы не заметите результатов выполнения программы. Дело в том, что включенная в IDE система исполнения программ имеет свое собственное окно, отличное от основного рабочего окна разработки. Для перехода в окно выполнения необходимо использовать комбинацию Alt+F5. Тогда на экране появится строка “Здравствуй, Мир!”, свидетельствующая о том, что наш эксперимент завершился успешно, и программа работает.

Лабораторная работа № 2. Вычисление значения математического выражения

Задание: написать программу, осуществляющую математические вычисления с использованием стандартных функций математической библиотеки языка программирования Си (по вариантам). Аргументы для производства вычислений вводятся пользователем с клавиатуры. При этом программа должна проверять принадлежность введенных значений (чисел с плавающей точкой) области допустимых значений для данного выражения и в случае необходимости выдавать на экран диагностическое сообщение о том, что аргументы не принадлежат области определения.

Варианты:

1. $\frac{1}{z}\ln\left(\frac{\sin x}{\cos^2 y + 1}\right)$.

2. $e^x \sqrt{\frac{\mathrm{tg}^2 y - 2}{|z-3|}}$.

3. $\frac{\ln(z-5y)}{\sqrt{\cos x - \sin y}}$.

4. $\sqrt{\frac{\cos^2 x + \sin^2 y}{\ln 3z}}$.

5. $\frac{\sqrt{\sin x + \mathrm{tg}\, z}}{\ln(5x - |3y|)}$.

6. $\sqrt{\frac{\sin x}{\cos z} - \frac{\ln(3z)}{\mathrm{tg}(5y)}}$.

Справочная информация: для вычисления тригонометрических функций используйте стандартные функции математической библиотеки sin(*x*), cos(*x*), tan(*x*), при этом вместо *x* подставляется аргумент соответствующей функции. Для вычисления модуля используется функция abs(*x*). Рассчитать e^x можно с помощью exp(*x*). Квадратный корень извлекается с помощью стандартной функции sqrt(*x*), натуральный логарифм определяется с помощью log(*x*).

Возможный вариант решения (для первого варианта):

```
#include <math.h>
#include <stdio.h>

int main()
  {
     float x,y,z,rez;

     printf("Программа расчета значения математического
выражения\n");
     printf("Введите три аргумента (вещественные числа)->");
     scanf("%f %f %f",&x,&y,&z);

     // Проверяем допустимость введенных аргументов
     if (z==0 || sin(x)<=0)
     {
printf("Вы ввели недопустимые значения аргументов\n");
      return 1; // Выход с ненулевым признаком завершения
     }

rez=(1/z)*log(sin(x)/(cos(y)*cos(y)+1));
printf("Результат расчета =%7.2f",rez);
```

```
    return 0;
}
```

Обратите внимание на то, что программа предварительно, перед тем как вводить исходные данные, выводит на дисплей сообщение о выполняемых ею задачах. Это является хорошим тоном при написании программ — программы создаются для пользователей, которые в общем случае не знают, для каких функций написана программа и какие данные в нее необходимо вводить. Каждый ввод данных с клавиатуры должен быть сопровожден содержательным приглашением, в котором желательно указывать число и тип вводимых аргументов.

В первой строке объявляются четыре переменные типа «число с плавающей точкой» — `float` в языке Си. Переменные `x`, `y` и `z` используются для хранения значений аргументов, переменная `rez` — для сохранения значения результата, подсчитанного по формуле (вообще-то в данном случае переменная `rez` является избыточной и добавлена для повышения наглядности программы).

В языке Си идентификаторы, то есть имена переменных и функций, представляют собой последовательность букв латинского алфавита, символов подчеркивания и десятичных цифр. Первым символом в идентификаторе обязательно должна быть буква. При этом прописные и строчные буквы в языке Си различаются, то есть `NAME`, `Name` и `name` — три разных идентификатора. Имена не должны совпадать с операторами языка и названиями стандартных функций. Базовые типы языка Си: `int` — целое число, `float` — вещественное число (число с плавающей точкой), `char` — одиночный символ, `void` — отсутствие типа, `double` — вещественное число двойной точности.

Перед названием типа могут встречаться так называемые модификаторы. Например, `unsigned` для чисел означает, что данное число не может хранить отрицательное значение. Таким образом, если при 16-разрядном целом числе тип `int` может хранить числа от –32 767 до 32 768, то `unsigned int` — от 0 до 65 535.

Используются также модификаторы `short` и `long`, указывающие на короткое (занимающее меньше памяти) и длинное (для хранения больших значений) целое соответственно. При этом можно писать просто `short` и `long` вместо `short int` и `long int`.

Каждая используемая в программе переменная должна быть предварительно объявлена с указанием типа, и в дальнейшем она может использоваться для хранения значений только указанного типа. Допустимо объявлять переменные одного типа как через запятую в одной строке, так и в нескольких строках.

Обратите внимание также на правильный синтаксис вызова стандартной функции ввода данных языка Си `scanf`. Форматная строка для ввода трех вещественных значений выглядит как `%f %f %f` — три значения типа `float`. Важно отметить то, что между форматными символами находится ровно по одному пробелу. Для ввода в стандартную функцию `scanf` передаются не сами переменные, а их адреса, в связи с этим при вводе переменных любого типа, кроме строкового (который сам является массивом, другими словами, указателем), перед именем переменной обязательно присутствует знак & (амперсанд).

Далее осуществляется проверка на то, что введенные с клавиатуры значения попадают в область определения вычисляемой функции. Помним, что знаменатель не может быть равным нулю (деление на ноль запрещено), а также на то, что логарифм определен только на положительных значениях. Знаменатель $\cos^2 y + 1$ проверять на ноль не требуется, поскольку результатом всегда будет положительное число. Поэтому достаточно проверить `z` на отличие от нуля и то, что sin(*x*) будет положительным. Для других вариантов заданий, возможно, проверка допустимости введенных значений окажется более сложной. Есть вероятность, что там окажется целесообразным использовать вспомогательные переменные для хранения промежуточных значений. Для проверки используется сложное условие, включающее в себя логический оператор ИЛИ (две вертикальные черты `||`). Логическое И записывается как `&&`, логическое отрицание — знаком `!` (восклицательный знак). Обратите внимание также на то, что в языке программирования Си одиночный знак равенства `=` используется для записи оператора присваивания, а в записи условий равенство обозначается как `==`. Для записи сравнений используются также знаки `<` — меньше, `>` — больше, `<=` — меньше или равно, `>=` — больше или равно. При условии, что введенное значения переменных `x` или `z` попадает в недопустимую область, выдается диагностическое сообщение и происходит выход из программы с ненулевым значением, передаваемым в операционную систему (`return 1`). В противном случае выполнение программы продолжается, и после расчета значения выражения оно печатается с помощью функции `printf`. При этом применение форматного символа `%7.2f` позволяет отвести на экране под печать результата ровно семь позиций, причем две из них отводятся под печать цифр после запятой (в языке Си вместо запятой пользуются десятичной точкой).

Интересным в тексте программы является также использование комментариев. Для улучшения восприятия программы человеком помимо правильного стиля использования пробелов и отступов полезно применять комментарии. Комментарии (примечания) в языке Си — произвольный

текст (возможно, включающий несколько строк), заключенный между последовательностями символов `/*` и `*/`. В применяемой нами при выполнении лабораторных работ интегрированной среде разработки Турбо Си разрешается также (как и в языке программирования C++) использование однострочных комментариев, при этом комментарий в строке начинается с пары символов `//`. Примечания должны носить содержательный характер, то есть, например, оператор `s=a*b` не должен комментироваться как «присваивание `s` произведения `a*b`», вместо этого в данном случае комментарием может быть «вычисляем площадь прямоугольника».

Лабораторная работа № 3. Табулирование функции

Задание: написать программу, которая выполнит табулирование (формирование на экране таблицы значений) функции из предыдущей лабораторной работы с сохранением варианта. Табулирование проводится по переменной *x, y* или *z* в зависимости от наличия разрывов в области определения (в таблице значений в случае неопределенности функции для данного аргумента должно стоять «не определена»). Границы изменения значения задаются пользователем, как и фиксированные значения двух других переменных (если при них функция не определена, выполнять табулирование не имеет смысла).

Возможный вариант решения:

```
/* Программа табулирования функции f(x,y,z)=
1/(sqrt(sin(x)))+cos(y)/log(z) */
#include <stdio.h>
#include <math.h>

int main()
{
float x,y,z,f;
float xn,xk,sh;

printf("Программа табулирования функции
f(x,y,z)=1/(sqrt(sin(x)))+cos(y)/log(z)\n");

printf("Введите y>");scanf("%f",&y);
printf("Введите z>");scanf("%f",&z);
/* Сначала проверяем, что z>0, чтобы можно было проверять log(z)! */
if (z<=0) {printf("Значение z не входит в ОДЗ!\n\7");return -2;}
```

```
if (log(z)==0) {printf("Значение z не входит в ОДЗ!\n\7");return -
3;}

printf("Введите начальное значение x>");scanf("%f",&xn);
printf("Введите конечное значение x>");scanf("%f",&xk);
printf("Введите шаг по x>");scanf("%f",&sh);

/* Печатаем "шапку" таблицы */
printf("+-----+--------+\n");
printf("!  x  !f(x,y,z)!\n");
printf("+-----+--------+\n");
for (x=xn;x<=xk;x+=sh)
{
    printf("!%5.2f!",x);
    if (sin(x)>0) /* Если x входит в ОДЗ, выводим значение функции
*/
    {
      f=1/(sqrt(sin(x)))+cos(y)/log(z);
      printf("%8.2f!\n",f);
    }
    else /* Иначе выводим сообщение, что функция не определена */
      printf(" не опр.!\n");
}
printf("+--------------+\n");

return 0;
}
```

Лабораторная работа № 4. Сумма нечетных

Задание: написать программу, которая с использованием цикла `do` языка Си вводит с экрана и накапливает в некоторой переменной сумму нечетных в последовательности целых чисел тремя способами.

- с помощью оператора `if`;
- с использованием условного выражения языка Си;
- с передачей числовой последовательности через командную строку в качестве параметров функции `main` и использования алгебраических свойств нуля и единицы.

Возможные варианты решений:

```
/* Третий вариант суммы нечетных с командной строкой */
```

```
#include <stdio.h>

int main(int argc,char* argv[])
{
  int i,n,s=0;
  for(i=1;i<argc;i++)
  {
    sscanf(argv[i],"%d",&n);
    s+=n*(n%2); /* s+=((n%2)?1:0);*/
          }
  printf("\ns=%d\n",s);
          return 0;
}
```

Обратите внимание на выражение `s+=n*(n%2)`. К переменной `s`, накапливающей сумму нечетных, с помощью операции `+=` прибавляется произведение очередного числа на остаток от его же деления на 2, вычисляемый с помощью операции `%`. Для нечетного числа остаток равен 1; произведение равно самому `n`, которое и будет прибавлено к `s`; для четного остаток равен 0, в этом случае произведение будет равным нулю и прибавление нуля не изменит `s`. С использованием условного выражения Си можно написать для этого же `s+=((n%2)?1:0);`.

Лабораторная работа № 5. Сортировка матрицы

Задание: написать программу, которая задает при объявлении (в тексте программы) матрицу вещественных чисел заданного размера, сортирует строки или столбцы матрицы (переставляя их местами целиком) в соответствии с заданным критерием (по вариантам) и выводит результат — отсортированную матрицу (предварительно вывести на экран исходную матрицу для контроля).

Варианты:

1. Отсортировать столбцы матрицы 5×5 по убыванию по критерию максимальной суммы элементов столбца.
2. Отсортировать строки матрицы 4×4 по возрастанию по критерию минимальной суммы элементов строки.
3. Отсортировать столбцы матрицы 3×4 по убыванию по критерию среднего значения элементов в столбце.

4. Отсортировать строки матрицы 5×3 по возрастанию по критерию среднего значения элементов в строке.
5. Отсортировать столбцы матрицы 5×5 по возрастанию минимальных элементов внутри столбца.
6. Отсортировать строки матрицы 4×3 по убыванию максимальных элементов внутри столбца
7. Отсортировать столбцы матрицы 5×5 по убыванию по критерию среднего значения элементов столбца.

Возможное решение (для первого варианта):

```
#include <stdio.h>

// Вспомогательная функция, используется для вывода матрицы
int print_Mat(float x[5][5])
{
  int i,j;

  for(i=0;i<5;i++)
  {
    for(j=0;j<5;j++)
      printf("%7.2f",x[i][j]);
    printf("\n");
  }

}

// Основная функция программы — точка входа
int main()
{
  int i,j,jj;
  float z,MaxSum,Sum,a[5][5]= { {1    ,2    ,3    ,4    ,0.5},
                       {2    ,5    ,-1   ,-7   ,100},
                       {5    ,4    ,3     ,2   ,1},
                       {-100 ,11   ,23.81,-5   ,1},
                       {1    ,2    ,3    ,4    ,5} }; //z -буфер
printf ("Программа сортировки столбцов матрицы 5×5 по убыванию
максимумов\n");

  // Печатаем матрицу до обработки
  printf("Исходная матрица:\n");
  print_Mat(a); // Передаем матрицу в функцию print_Mat для печати
```

```
  // Начало алгоритма сортировки перестановками
  for (j=0;j<4;j++) // Цикл с первого столбца до предпоследнего
  {
      MaxSum=0;
for (i=0;i<5;i++) MaxSum+=a[i][j]; // Находим сумму элементов
текущего столбца-максимума
      for (jj=j+1;jj<5;jj++)
      {
        Sum=0;
        for (i=0;i<5;i++) Sum+=a[i][jj]; // Сумма столбца-кандидата
        if (MaxSum<Sum)
        {
       MaxSum=Sum; // Новый максимум суммы
       for(i=0;i<5;i++) // Переставляем местами столбцы
       {
         z=a[i][j];
         a[i][j]=a[i][jj];
         a[i][jj]=z;
       }
        }
      }
  }

  // Печать матрицы после сортировки
  printf("Результирующая матрица:\n");
  print_Mat(a);

  return 0;
}
```

Обратите внимание на то, как на языке Си задаются значения элементов двумерной матрицы при объявлении. В программе используется алгоритм сортировки с перестановкой элементов (в качестве элементов здесь фигурируют целые столбцы). Сначала берем первый столбец и рассматриваем его в качестве столбца с максимальной суммой элементов. Затем перебираем столбцы от второго до пятого, находим суммы элементов в них и в случае, если они превышают сумму элементов первого столбца, переставляем столбцы (не забывая при этом поменять и значение текущего максимума суммы элементов столбца). Пройдя таким образом все столбцы с первого до предпоследнего, получаем матрицу с отсортированными столбцами. Для печати значений элементов матрицы здесь применяется специально написанная функция `print_Mat`. Обратите внимание на правильную запись формальных аргументов при объявлении

функции и на правильную передачу в функцию `print_Mat` матрицы из основной функции.

Лабораторная работа № 6. Низкоуровневые операции языка Си

Задание: написать и отладить программу наложения вводимой с клавиатуры битовой маски на заданное число. Вывести аргумент, маску и результат в двоичном виде с помощью специально разработанной функции, опирающейся на низкоуровневые возможности языка Си.

Возможный вариант решения:

```
/* Низкоуровневые операции языка Си */
#include <stdio.h>

/* Функция выводит аргумент на экран в двоичном виде */
int prn2(int a)
{
  int i;

  /* поразрядный вывод 0 или 1 */
  for(i=sizeof(int)*8-1;i>=0;i--) /* перебираем разряды справа налево */
    printf("%d",(a>>i)&1); /* наложение маски &1 оставляет нужный бит */

  return 0;
}

int main()
{
  int n,mask;

  printf("Программа наложения битовой маски на число\n");
  printf("Введите число:");scanf("%d",&n);
  printf("Введите маску:");scanf("%d",&mask);
  printf("В двоичном виде:\n\n");
  printf("Исходное число: ");prn2(n);printf("\n");
  printf("    маска: ");prn2(mask);printf("\n");
  printf("==========================\n");
  printf("результат: ");prn2(n&mask);printf("\n");

  return 0;
```

```
}
```

Можно, напротив, сдвигать не число, а маску, сначала взяв ее для старшего бита (например, `unsigned m=32768` для 16-разрядных): `printf("%d",(m>>15-i)&a?1:0)` в `prn2()`.

Лабораторная работа № 7 «Оценки»

Задания:

1. Написать программу, которая с использованием оператора `switch` языка программирования Си преобразует вводимое с клавиатуры целое число от 2 до 5 в принятую в системе высшего образования России оценку («неудовлетворительно», «удовлетворительно», «хорошо», «отлично»), а в случае ввода другого числа выводит сообщение об отсутствии такой оценки.
2. Написать программу обратного преобразования с использованием массива строк.

Возможный вариант решения (первая часть):

```
#include <stdio.h>

int main()
{
  int Mark;

  // Ввод оценки в виде целого числа
  printf("Программа оценивания по шкале вуза\n Введите целое число-
>");
  scanf("%d",&Mark);

  // Вывод оценки в строковой форме
  switch (Mark)
  {
    case 2:printf("неуд\n");break;
    case 3:printf("удовл\n");break;
    case 4:printf("хорошо\n");break;
    case 5:printf("отлично\n");break;

    default:printf("Нет такой оценки!\n");
  }

  return 0;
}
```

Обратите внимание на то, что в конце каждой из рассматриваемых альтернатив оператора-переключателя `switch` обязательно должен стоять оператор `break`, поскольку иначе вследствие того факта, что конструкции «`case` *значение*» в программе рассматриваются как метки (`labels`), произойдет «проваливание» к следующей альтернативе и выполнение соответствующей строки. Строка `default` служит для задания действий в том случае, если ни одно из альтернативных значений, рассматриваемых в переключателе, не подошло к значению тестируемой переменной.

Возможный вариант решения (вторая часть):

```
#include <stdio.h>
#include <string.h>

int main()
{
  int i,pr=0; // pr – признак того, что оценка-строка найдена
  char str[12],Marx[4][12]={"неуд","удовл","хорошо","отлично"};

  // Ввод оценки в виде строки
  printf("Программа оценивания по шкале вуза\n Введите целое число-
>");
  scanf("%s",str);

  // Вывод оценки в строковой форме путем поиска в массиве строк
  for (i=0;i<4 && pr!=1;i++)
    if (!strcmp(Marx[i],str)) {printf("%d",i+2);pr=1;}

  if (pr!=1) printf("Нет такой оценки!"); // Проверяем признак

  return 0;
}
```

Обратите внимание на то, что при вводе текстовой строки в функции `scanf`, поскольку `str` и так представляет собой строку, то есть массив символов (другими словами, указатель), используется форматная строка `%s` и не используется значок амперсанда (&).

Признак `pr`, который изначально устанавливается прямо при объявлении в ноль, служит в качестве флага, который индицирует, нашли мы в массиве требуемое значение или нет. Если значение найдено, то не только печатается его номер в массиве (естественно, с прибавлением 2, чтобы соблюсти систему российских вузовских оценок), но и переменной-флагу `pr` присваивается значение 1. Подобный прием часто используется в программировании при решении самых разных задач. Альтернативный

вариант — прерывание выполнения цикла с помощью оператора `break`.

Обратите внимание на то, как изменилась программа — центр тяжести перенесен с исполняемой части (алгоритм) на данные, при этом функциональность сохранена. Подобные возможности часто существуют и в больших профессиональных программных комплексах. Подумайте над этим. Запомните, как задавать значения элементов массива при объявлении и то, что `Marx` получился двумерным массивом, поскольку он представляет собой одномерный массив строк, которые, в свою очередь, представляют собой массивы символов.

Лабораторная работа № 8. Система управления базой данных о студентах

Задание: написать программу управления базой данных о студентах с сохранением в текстовых файлах следующей информации: фамилия, имя, отчество, номер группы, год рождения, средний балл в последнюю сессию, размер стипендии (это минимальный набор, перечень хранимой информации можно расширять) — и возможностями поиска данных по фамилии, номеру группы, диапазону среднего балла (от минимума до максимума), диапазону годов рождения, пополнения базы новой записью и удаления записи по паре «ФИО + год рождения». Если условиям поиска удовлетворяют несколько студентов, вывести информацию обо всех.

Один из возможных вариантов реализации:

```
/* Система управления базой данных о студентах */
#include <stdio.h>
#include <string.h>

// Описание структуры данных "Студент"
struct
{
  char Family[50]; // Фамилия
  char Imy[50];  // Имя
  char Otcestvo[50];  // Отчество
  char NGr[7]; // Номер группы
  int GodR; // Год рождения
  float SrBall; // Средний балл
  float Stip; // Размер стипендии
} Stud;

// Функция добавления в БД записи о студенте
```

```
int Dobavl()
{
  FILE *fp;

  // Открываем текстовый файл данных для добавления

if((fp=fopen("Students.txt","a"))==NULL){printf("Ошибка!\n\7");retur
n 1;}
  printf("Введите информацию о добавляемом в БД студенте\n");

  printf("ФИО:");scanf("%s %s
%s",Stud.Family,Stud.Imy,Stud.Otcestvo);
  fprintf(fp,"%s %s %s ",Stud.Family,Stud.Imy,Stud.Otcestvo);
  printf("Номер группы:");scanf("%s",Stud.NGr);fprintf(fp,"%s
",Stud.NGr);
  printf("Год рождения:");scanf("%d",&Stud.GodR);fprintf(fp,"%d
",Stud.GodR);
  printf("Средний балл:");scanf("%f",&Stud.SrBall);fprintf(fp,"%f
",Stud.SrBall);
  printf("Размер
стипендии:");scanf("%f",&Stud.Stip);fprintf(fp,"%f\n",Stud.Stip);

  fclose(fp);

  return 0;
}

// Функция чтения очередной записи о студенте из файла
int ReadNext(FILE *fp)
{
  fscanf(fp,"%s",Stud.Family); // прочитали фамилию
  fscanf(fp,"%s",Stud.Imy); // прочитали имя
  fscanf(fp,"%s",Stud.Otcestvo); // прочитали отчество
  fscanf(fp,"%s",Stud.NGr); // прочитали номер группы
  fscanf(fp,"%d",&Stud.GodR); // прочитали год рождения
  fscanf(fp,"%f",&Stud.SrBall); // прочитали размер стипендии
  fscanf(fp,"%f",&Stud.Stip); // прочитали размер стипендии

  return 0;
}

// Поиск по номеру группы
int PoiskNGr()
{
```

```
  FILE *fp;
  int pr=0;
  char Str[80];

  printf("Введите номер группы:");
  scanf("%s",Str);
  printf("\n");

  // начинаем читать файл с начала

if((fp=fopen("Students.txt","r"))==NULL){printf("Ошибка!\n\7");retur
n 2;}

  while(!feof(fp))
  {
    ReadNext(fp);

    if (!strcmp(Str,Stud.NGr))
    {
      pr=1;
      if(!feof(fp))
   printf("%s %s %s %s %d %5.2f
%7.2f\n",Stud.Family,Stud.Imy,Stud.Otcestvo,
         Stud.NGr,Stud.GodR,Stud.SrBall,Stud.Stip);
    }
  }

  if (pr!=1) printf("Не найдено подобной записи!\n");

  fclose(fp);

  return 0;
}

// Функция поиска по фамилии
int PoiskFamily()
{
  FILE *fp;
  int pr=0;
  char Str[80];

  printf("Введите фамилию:");
  scanf("%s",Str);
```

```
  printf("\n");

  // начинаем читать файл с начала

if((fp=fopen("Students.txt","r"))==NULL){printf("Ошибка!\n\7");retur
n 2;}

  while(!feof(fp))
  {
    ReadNext(fp);

    if (!strcmp(Str,Stud.Family))
    {
      pr=1;
      if(!feof(fp))
   printf("%s %s %s %s %d %5.2f
%7.2f\n",Stud.Family,Stud.Imy,Stud.Otcestvo,
         Stud.NGr,Stud.GodR,Stud.SrBall,Stud.Stip);
    }
  }

  if (pr!=1) printf("Не найдено подобной записи!\n");

  fclose(fp);

  return 0;
}

// Поиск по диапазону года рождения
int PoiskGodr()
{
  FILE *fp;
  int pr=0,GodMin,GodMax;

  printf("Год рождения от:");
  scanf("%d",&GodMin);
  printf("Год рождения по:");
  scanf("%d",&GodMax);
  printf("\n");

  // начинаем читать файл с начала
```

```
if((fp=fopen("Students.txt","r"))==NULL){printf("Ошибка!\n\7");retur
n 2;}

  while(!feof(fp))
  {
    ReadNext(fp);

    if (Stud.GodR>=GodMin && Stud.GodR<=GodMax)
    {
      pr=1;
      if(!feof(fp))
   printf("%s %s %s %s %d %5.2f
%7.2f\n",Stud.Family,Stud.Imy,Stud.Otcestvo,
         Stud.NGr,Stud.GodR,Stud.SrBall,Stud.Stip);
    }
  }

  if (pr!=1) printf("Не найдено подобной записи!\n");
  fclose(fp);

  return 0;
}

// Поиск по диапазону среднего балла
int PoiskSrBall()
{
  FILE *fp;
  int pr=0;
  float BalMin,BalMax;

  printf("Средний балл от:");
  scanf("%f",&BalMin);
  printf("Средний балл по:");
  scanf("%f",&BalMax);
  printf("\n");

  // начинаем читать файл с начала

if((fp=fopen("Students.txt","r"))==NULL){printf("Ошибка!\n\7");retur
n 2;}

  while(!feof(fp))
```

```
  {
    ReadNext(fp);

    if (Stud.SrBall>=BalMin && Stud.SrBall<=BalMax)
    {
      pr=1;
      if(!feof(fp))
   printf("%s %s %s %s %d %5.2f
%7.2f\n",Stud.Family,Stud.Imy,Stud.Otcestvo,
         Stud.NGr,Stud.GodR,Stud.SrBall,Stud.Stip);
    }
  }

  if (pr!=1) printf("Не найдено подобной записи!\n");
  fclose(fp);

  return 0;
}

// Удаление записи
int Udal()
{
  FILE *fp,*fp1;
  int pr=0;
  char Fam[50],Imy[50],Otc[50];
  int God;

  printf("Введите ФИО для удаления: ");
  scanf("%s %s %s",Fam,Imy,Otc);
  printf("Введите год рождения: ");
  scanf("%d",&God);
  printf("\n");

  // начинаем читать файл с начала

if((fp=fopen("Students.txt","r"))==NULL){printf("Ошибка!\n\7");retur
n 2;}
  // промежуточный файл

if((fp1=fopen("Students.bak","w"))==NULL){printf("Ошибка!\n\7");retu
rn 3;}

  while(!feof(fp))
```

```
  {
    ReadNext(fp);

    // Переписываем в промежуточный файл по очереди все записи,
кроме удаляемой
    if (strcmp(Fam,Stud.Family) || strcmp(Imy,Stud.Imy) ||
   strcmp(Otc,Stud.Otcestvo) || God!=Stud.GodR)
    {
      if(!feof(fp))
   fprintf(fp1,"%s %s %s %s %d %5.2f
%7.2f\n",Stud.Family,Stud.Imy,Stud.Otcestvo,
         Stud.NGr,Stud.GodR,Stud.SrBall,Stud.Stip);
    }
    else
      pr=1;
  }

  if (pr==1)
    printf("Запись успешно удалена!");
  else
    printf("Не найдено подобной записи!\n");

  fclose(fp);
  fclose(fp1);

  // Для успеха копирования промежуточного файла сначала стираем
основной
  if (remove("Students.txt")==-1) {printf("Ошибка!\n\7");return 4;}
  // Теперь переименовываем промежуточный файл в основной
  if (rename("Students.bak","Students.txt")==-1)
  {
    printf("Ошибка!\n\7");
    return 5;
  }

  return 0;
}

// Меню
int Menu()
{
  int alt;
```

```
  printf ("\nВведите номер требуемого режим работы:\n\n");
  printf ("1. Добавление данных\n");
  printf ("2. Поиск данных по фамилии\n");
  printf ("3. Поиск данных по номеру группы\n");
  printf ("4. Поиск данных по диапазону годов рождения\n");
  printf ("5. Поиск данных по диапазону среднего балла\n");
  printf ("6. Удаление данных\n");
  printf ("7. Выход\n");

  scanf("%d",&alt);

  return alt;
}

// Главная функция программы
int main()
{
  int Reg=0;

  printf("\nСистема управления базой данных о студентах\n");

  while (Reg!=7) // Цикл, пока не будет введен вариант выхода
  {
    Reg=Menu();

    switch (Reg)
    {
      case 1:Dobavl();break; // Вызов функции добавления данных
      case 2:PoiskFamily();break; // Вызов функции поиска по фамилии
      case 3:PoiskNGr();break; // Вызов функции поиска по номеру
группы
      case 4:PoiskGodr();break; // Вызов функции поиска по году
рождения
      case 5:PoiskSrBall();break; // Вызов функции поиска по
среднему баллу
      case 6:Udal();break; // Вызов функции удаления
    }
  }

  return 0;
}
```

Обратите внимание на то, что в программе используется *структура*

данных языка программирования Си. Структура данных `struct` объявлена в начале вне функций, поэтому переменная данного типа `Stud` доступна из любой функции программы как глобальная.

Важный момент, заслуживающий внимания, — повторяющиеся в различных местах программы одинаковые действия (например, чтение очередной записи из текстового файла) вынесены в отдельные функции.

В программе при удалении записи используется тот же подход с промежуточным файлом, что и в предыдущей лабораторной работе.

Обратите также внимание на то, что программа работает с записями на диске и не хранит постоянно в оперативной памяти компьютера полную базу данных. Это позволяет сэкономить расходуемый объем ОЗУ, но отрицательно сказывается на производительности (что несущественно для учебного примера, но может стать значимым при больших объемах данных).

Приведенная программа не является единственным вариантом реализации программы, решающей поставленную задачу.

При выполнении данной лабораторной работы всячески поощряются творческий подход, использование различных вариантов решения и самостоятельная работа. В частности, можно было бы построить более удобную систему меню с применением функций консольного ввода-вывода (из библиотеки с заголовочным файлом `conio.h`). Можно также применить для сохранения информации не текстовый режим, а режим сохранения целых записей, поддерживаемый языком Си. Напротив, можно рассматривать все данные (включая год рождения, средний балл и стипендию) как текстовые. Или по-другому организовать удаление информации. Также можно применить, например, функцию поиска в текстовом файле `fseek` и другие файловые возможности. Тем не менее приведенный вариант позволяет решить поставленную задачу, является довольно простым и обозримым.

Лабораторная работа № 9. ООП на примере классов геометических фигур

Задание: написать и отладить объектно-ориентированную программу на языке C++, реализующую абстрактный класс геометрических фигур и наследующие его классы конкретных фигур (по вариантам):

Вариант 1. Окружность и прямоугольник;

Вариант 2. Плюсик и окружность;

Вариант 3. Отрезок линии и окружность;

Вариант 4. Треугольник и окружность.

Должны быть реализованы для фигур методы «показать», «покрасить», «скрыть» (минимальный набор фигур и методов, приветствуется расширение).

Возможный вариант программы:

```
// Геометрические фигуры
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include <math.h>

#define Pi 3.1415926

// Объявление класса геометрических фигур на плоскости
class Fig
{
  public:
   // Переменные-члены класса
   int x,y; // Декартовы координаты
   int color; // Цвет

   // Функции-члены класса
   virtual void Pokaz()=0; // «Чистая» виртуальная функция
   void Skryt()
   {
     color=BLACK;
     Pokaz();
   }
   void Pokras(int cwet)
   {
     color=cwet;
   }
};

// Класс прямоугольника Pram, наследуемый от Fig
class Pram : public Fig
{
  public:
   int Shir,Vys; // Добавляются ширина и высота
   void Pokaz()
   {
```

```
    setcolor(color);
    gotoxy(x,y);
    rectangle(x,y,x+Shir,y+Vys);
  }
  // Конструктор
  Pram (int ix=20,int iy=20,int Sh=20,int V=10,int cvet=WHITE)
  {
    x=ix;y=iy;Shir=Sh;Vys=V;color=cvet;
  }
};

// Класс круга Krug, наследуемый от Fig
class Krug : public Fig
{
  public:
   int Radius; // Добавляется радиус
   void Pokaz()
   {
     setcolor(color);
     gotoxy(x,y);
     circle(x,y,Radius);
   }
   // Конструктор
   Krug (int ix=50,int iy=50,int r=10,int cvet=WHITE)
   {
     x=ix;y=iy;Radius=r;color=cvet;
   }
};

// Главная функция программы
int main()
{
  int gdriver = DETECT, gmode, errorcode;
  int x,y;
  Pram P1,P2(100,50,30,20,GREEN);
  Krug K(150,100,15,YELLOW);

  // Инициализация графического режима
  initgraph(&gdriver, &gmode, "");
  errorcode = graphresult();
  if (errorcode != grOk)
  {
```

```
    cerr << "Ошибка инициализации графического режима!\7\n";
    getch();
    return errorcode;
  }

  // Начало работы с фигурами
  P1.Pokaz();
  getch();
  P1.Pokras(MAGENTA);
  P1.Pokaz();
  getch();
  P2.Pokaz();
  getch();
  K.Pokaz();
  getch();
  K.Skryt();
  getch();
  P2.Skryt();

  getch();

  // Движение по синусоиде, переливаясь
  int c=WHITE;
  for (float ug=-2*Pi;ug<2*Pi;ug+=0.1,c++)
  {
   P2.x=320+40*ug;
   P2.y=220-sin(ug)*240;

   P2.Pokras(c);
   P2.Pokaz();
   for(float c=0;c<5000000;c++) ; // задержка
   //getch();
   //P2.Skryt();
  }

  // Отключение графического режима
  closegraph();
  return 0;
}
```

Более сложный пример, разработанный студентом СГАУ А. Авдеевым, с движущимся паровозиком, пляшущими человечками и прыгающим мячиком (рис. А.2 и А.3).

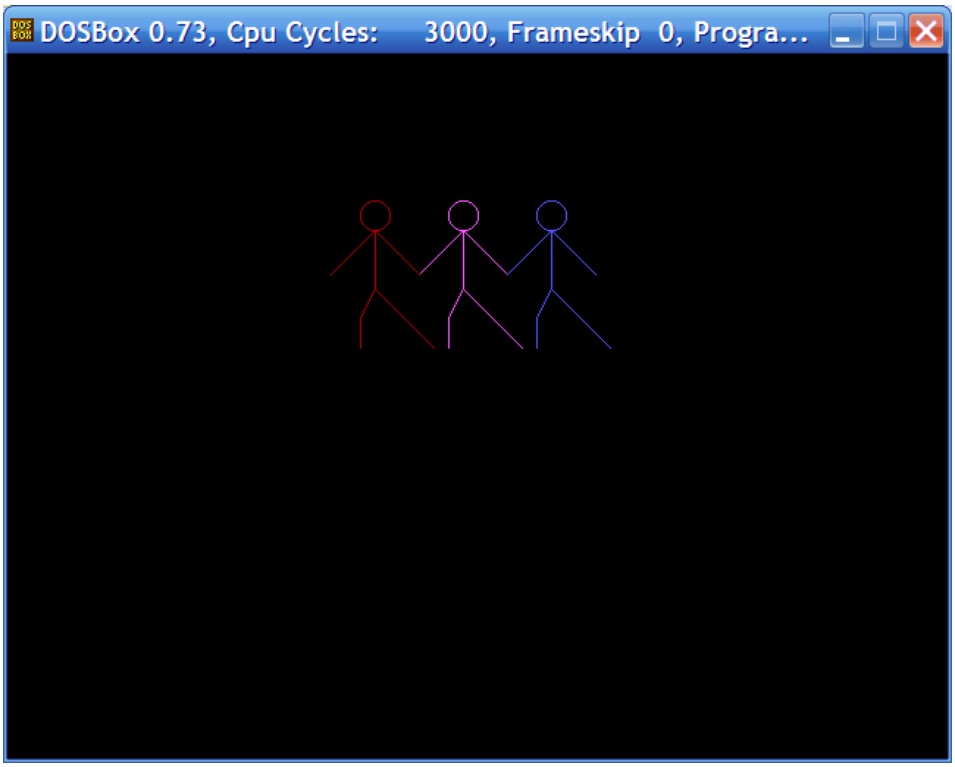


Рис. А.2

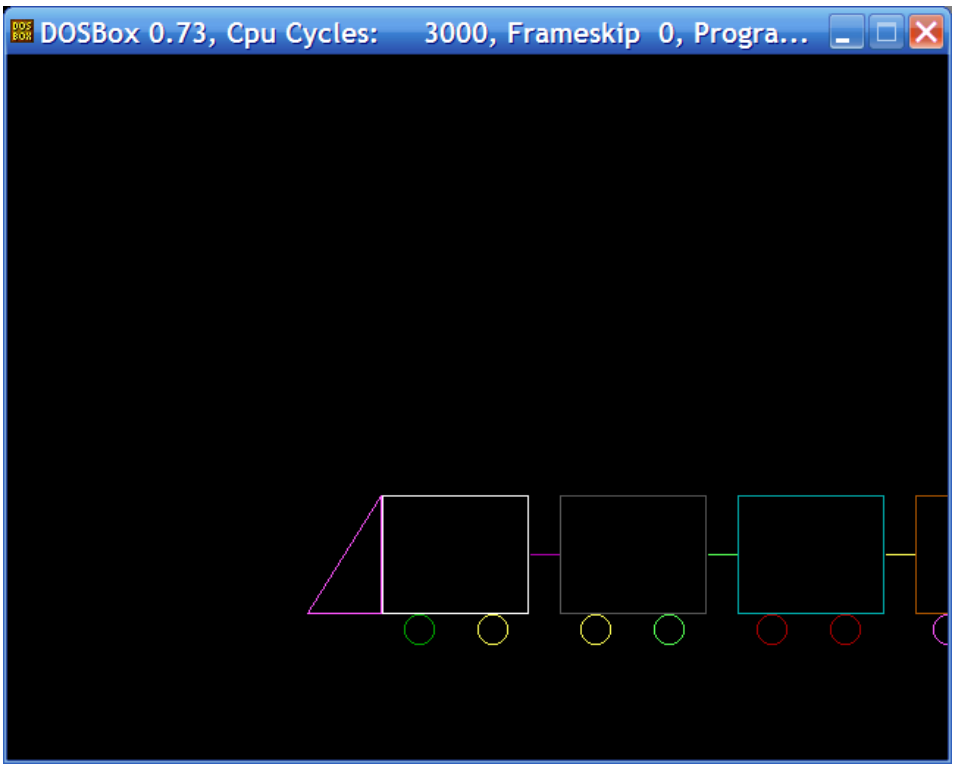


Рис. А3

```
//Программа графика-C++, выполнил Авдеев Андрей, гр. 6111
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
```

```
#include <dos.h>

//Класс геометрической фигуры
class figura
{
  protected:
    int cvet,x,y;
  public:
    virtual void Pokaz()=0;
    void Skryt()//Функция скрытия фигуры
    {
      cvet=BLACK;
      Pokaz();
    }
    void Pokras(int color)//Функция изменения цвета фигуры
    {
      cvet=color;
    }
    void setxy(int X,int Y) // Установка новых координат
    {
      x=X;y=Y;
    }
};

//Класс треугольник, наследующий функции класса figura
class treug:public figura
{
  public:
    //Конструктор для создания переменной типа treug
    treug(int x1, int y1)
    {
      x=x1; y=y1;
    }
    void Pokaz()//Функция показа фигуры
    {
      setcolor(cvet);
      line(x,y,x,y+80);
      line(x,y+80,x-50,y+80);
      line(x-50,y+80,x,y);
    }
};
```

```
//Класс прямоугольник – наследник класса figura
class pryam:public figura
{

  public:
    //Конструктор для создания переменной типа pryam
    pryam(int x1,int y1)
    {
      x=x1;y1=y1;
    }
    void Pokaz()//Функция показа фигуры
    {
      setcolor(cvet);
      rectangle(x+1,y,x+100,y+80);
    }
};

//Класс линия – наследник класса figura
class liniya:public figura
{
  public:
    //Конструктор для создания переменной типа liniya
    liniya(int x1, int y1)
    {
      x=x1;y=y1;
    }
    void Pokaz()//Функция показа фигуры
    {
      setcolor(cvet);
      line(x+102,y+40,x+122,y+40);
    }
};

//Класс круг - наследник класса figura
class krug:public figura
{
  private:
    int r;
  public:
    //Констуктор для создания переменной типа krug
    krug(int x1, int y1, int rad)
    {
```

```
      x=x1, y=y1, r=rad;
    }
    void Pokaz()//Функция показа фигуры
    {
      setcolor(cvet);
      circle(x+26,y+91,r);
    }
};

//Класс человек, наследующий функции класа figura
class chelovek:public figura
{
  private:
    int x,y,r,pol;
  public:
    //Конструктор для создания переменной типа chelovek
    chelovek(int x1, int y1,int rad)
    {
      x=x1; y=y1;r=rad;
    }
    void pol0()//Положение номер 0
    {
      pol=0;
    }
    void pol1()//Положение номер 1
    {
      pol=1;
    }
    void pol2()//Положение номер 2
    {
      pol=2;
    }
    void pol3()//Положение номер 3
    {
      pol=3;
    }
    void Pokaz();//Функция показа фигуры
};

void chelovek::Pokaz()//Функция показа фигуры
{
      setcolor(cvet);
```

```
switch(pol)//Выбор положения фигуры
{
 case 0:
 line(x,y,x,y-20);
 line(x,y-20,x+10,y-40);
 line(x+10,y-40,x+20,y-20);
 line(x+20,y-20,x+20,y);
 line(x+10,y-40,x+10,y-80);
 line(x+10,y-80,x-10,y-60);
 line(x-10,y-60,x,y-40);
 line(x+10,y-80,x+30,y-60);
 line(x+30,y-60,x+20,y-40);
 circle(x+10,y-90,r);break;
 case 1:
 line(x+10,y,x-10,y-10);
 line(x-10,y-10,x+10,y-20);
 line(x+10,y-20,x+30,y-10);
 line(x+30,y-10,x+10,y);
 line(x+10,y-20,x+10,y-60);
 line(x+10,y-60,x-10,y-40);
 line(x-10,y-40,x+10,y-30);
 line(x+10,y-60,x+30,y-40);
 line(x+30,y-40,x+10,y-30);
 circle(x+10,y-70,r);break;
 case 2:
 line(x,y,x,y-20);
 line(x,y-20,x+10,y-40);
 line(x+10,y-40,x+30,y-20);
 line(x+30,y-20,x+50,y);
 line(x+10,y-40,x+10,y-80);
 line(x+10,y-80,x-10,y-60);
 line(x-10,y-60,x-20,y-50);
 line(x+10,y-80,x+30,y-60);
 line(x+30,y-60,x+40,y-50);
 circle(x+10,y-90,r);break;
 case 3:
 line(x-30,y,x-10,y-20);
 line(x-10,y-20,x+10,y-40);
 line(x+10,y-40,x+20,y-20);
 line(x+20,y-20,x+20,y);
 line(x+10,y-40,x+10,y-80);
 line(x+10,y-80,x-10,y-60);
```

```
       line(x-10,y-60,x-20,y-50);
       line(x+10,y-80,x+30,y-60);
       line(x+30,y-60,x+40,y-50);
       circle(x+10,y-90,r);break;
     }
}

int main()
{
  int x1,y1,x9,y9,x16,y16,k,r,r1,i,r2,l,d;
  int s1,s2,s3,v1,v2,v3,v4,n,k1,k2,k3,k4,k5,k6,k7,k8,vv;

  //Подключение графического режима
  int gd=DETECT, gm;
  initgraph(&gd, &gm, "C:\\BGI\\");

  //Создание фигур, составляющих паровоз
  x1=689;y1=300;r=10;
  treug nos(x1,y1);
  pryam vag1(x1,y1);
  liniya sv1(x1,y1);
  pryam vag2(x1+121,y1);
  liniya sv2(x1+121,y1);
  krug kol1(x1,y1,r);
  krug kol2(x1+50,y1,r);
  krug kol3(x1+120,y1,r);
  krug kol4(x1+170,y1,r);
  krug kol5(x1+240,y1,r);
  krug kol6(x1+290,y1,r);
  krug kol7(x1+360,y1,r);
  krug kol8(x1+410,y1,r);
  pryam vag3(x1+242,y1);
  liniya sv3(x1+242,y1);
  pryam vag4(x1+363,y1);

  //Цикл движения паровоза
  for(n=5,s1=13,s2=2,s3=54,v1=7,v2=32,v3=11,v4=14,k1=10,k2=6,k3=22,
   k4=18,k5=76,k6=44,k7=5,k8=65,x1=689;x1+363>0;n++,s1++,s2++,s3++,
   v1++,v2++,v3++,v4++,k1++,k2++,k3++,k4++,k5++,k6++,k7++,k8++,x1-
=5)
  {
    nos.Pokras(n);
```

```
nos.Pokaz();
vag1.Pokras(v1);
vag1.Pokaz();
sv1.Pokras(s1);
sv1.Pokaz();
vag2.Pokras(v2);
vag2.Pokaz();
sv2.Pokras(s2);
sv2.Pokaz();
kol1.Pokras(k1);
kol1.Pokaz();
kol2.Pokras(k2);
kol2.Pokaz();
kol3.Pokras(k3);
kol3.Pokaz();
kol4.Pokras(k4);
kol4.Pokaz();
kol5.Pokras(k5);
kol5.Pokaz();
kol6.Pokras(k6);
kol6.Pokaz();
kol7.Pokras(k7);
kol7.Pokaz();
kol8.Pokras(k8);
kol8.Pokaz();
vag3.Pokras(v3);
vag3.Pokaz();
sv3.Pokras(s3);
sv3.Pokaz();
vag4.Pokras(v4);
vag4.Pokaz();
delay(30);
//Сокрытие этих фигур
nos.Skryt();
vag1.Skryt();
sv1.Skryt();
vag2.Skryt();
sv2.Skryt();
vag3.Skryt();
kol1.Skryt();
kol2.Skryt();
sv3.Skryt();
```

```
  vag4.Skryt();
  kol4.Skryt();
  kol3.Skryt();
  kol5.Skryt();
  kol6.Skryt();
  kol7.Skryt();
  kol8.Skryt();
  //Изменение координат этих фигур
  nos.setxy(x1,y1);
  vag1.setxy(x1,y1);
  sv1.setxy(x1,y1);
  vag2.setxy(x1+121,y1);
  sv2.setxy(x1+121,y1);
  vag3.setxy(x1+242,y1);
  kol1.setxy(x1,y1);
  kol2.setxy(x1+50,y1);
  sv3.setxy(x1+242,y1);
  vag4.setxy(x1+363,y1);
  kol4.setxy(x1+170,y1);
  kol3.setxy(x1+120,y1);
  kol5.setxy(x1+240,y1);
  kol6.setxy(x1+290,y1);
  kol7.setxy(x1+360,y1);
  kol8.setxy(x1+410,y1);
}

//Задание координат для объектов типа chelovek
x9=300; y9=200;r1=10;

//Создание фигур типа человек в позе № 0
chelovek ch1(x9,y9,r1);
chelovek ch2(x9+60,y9,r1);
chelovek ch3(x9-60,y9,r1);
ch1.pol0();
ch2.pol0();
ch3.pol0();
ch1.Pokras(29);
ch2.Pokras(9);
ch3.Pokras(4);
ch1.Pokaz();
ch2.Pokaz();
ch3.Pokaz();
```

```
delay(1000);
ch1.Skryt();
ch2.Skryt();
ch3.Skryt();

//Цикл танцующих человечков
for(vv=0; vv<10; vv++)
{
  //Смена координат для позы № 1
  ch1.pol1();
  ch2.pol1();
  ch3.pol1();
  ch1.Pokras(29);
  ch2.Pokras(9);
  ch3.Pokras(4);
  ch1.Pokaz();
  ch2.Pokaz();
  ch3.Pokaz();
  delay(200);
  ch1.Skryt();
  ch2.Skryt();
  ch3.Skryt();

  //Смена координат для позы № 2
  ch1.pol2();
  ch2.pol2();
  ch3.pol2();
  ch1.Pokras(29);
  ch2.Pokras(9);
  ch3.Pokras(4);
  ch1.Pokaz();
  ch2.Pokaz();
  ch3.Pokaz();
  delay(200);
  ch1.Skryt();
  ch2.Skryt();
  ch3.Skryt();

  //Смена координат для позы № 1
  ch1.pol1();
  ch2.pol1();
  ch3.pol1();
```

```
  ch1.Pokras(29);
  ch2.Pokras(9);
  ch3.Pokras(4);
  ch1.Pokaz();
  ch2.Pokaz();
  ch3.Pokaz();
  delay(200);
  ch1.Skryt();
  ch2.Skryt();
  ch3.Skryt();

  //Смена координат для позы № 3
  ch1.pol3();
  ch2.pol3();
  ch3.pol3();
  ch1.Pokras(29);
  ch2.Pokras(9);
  ch3.Pokras(4);
  ch1.Pokaz();
  ch2.Pokaz();
  ch3.Pokaz();
  delay(200);
  ch1.Skryt();
  ch2.Skryt();
  ch3.Skryt();
}
//Создание объекта типа krug
x16=294; y16=149; r2=20;
krug kr(x16,y16,r2);

//Цикл для движения круга, отталкивающегося от стен
for(d=0; d<5 ; d++)
{
  //Цикл для движения вправо вверх
  for(x16=-6, y16=149,l=0; x16<294; x16+=15,  y16-=11,l++)
  {
    kr.Pokras(l);
    kr.Pokaz();
    delay(30);
    kr.Skryt();
    kr.setxy(x16,y16);
  }
```

```
    //Цикл для движения вправо вниз
    for(x16=294, y16=-71,l=0; x16<594; x16+=15, y16+=11 ,l++)
    {

      kr.Pokras(l);
      kr.Pokaz();
      delay(30);
      kr.Skryt();
      kr.setxy(x16,y16);
    }

    //Цикл для движения влево вниз
    for(x16=594, y16=149,l=0; x16>294; x16-=15, y16+=11 ,l++)
    {
      krug kr(x16,y16,r2);
      kr.Pokras(l);
      kr.Pokaz();
      delay(30);
      kr.Skryt();
      kr.setxy(x16,y16);
    }

    //Цикл для движения влево вверх
    for(x16=294, y16=369, r2=20,l=0; x16>-6; x16-=15, y16-=11 ,l++)
    {
      krug kr(x16,y16,r2);
      kr.Pokras(l);
      kr.Pokaz();
      delay(30);
      kr.Skryt();
      kr.setxy(x16,y16);
    }
  }

  //Закрытие графики
  closegraph();

  return 0;
}
```

УНИВЕРСИТЕТ ИТМО

Миссия университета – генерация передовых знаний, внедрение инновационных разработок и подготовка элитных кадров, способных действовать в условиях быстро меняющегося мира и обеспечивать опережающее развитие науки, технологий и других областей для содействия решению актуальных задач.

КАФЕДРА КОМПЬЮТЕРНЫХ ОБРАЗОВАТЕЛЬНЫХ ТЕХНОЛОГИЙ

Кафедра «Компьютерные образовательные технологии» (КОТ) создана в 2001 году на факультете информационных технологий и программирования (ИТП) Санкт-Петербургского государственного университета информационных технологий, механики и оптики (НИУ ИТМО) для реализации учебного процесса по ряду профильных дисциплин направления подготовки специалистов 230200 - «Информационные системы». С 2003 года кафедра КОТ стала выпускающей по специальности 230202 - «Информационные технологии в образовании», а с 2009 года по направлению подготовки магистров 230200 «Информационные системы». В апреле 2011 г. кафедра КОТ перешла в состав факультета компьютерных технологий и управления (КТиУ). В настоящее время в рамках реализации ФГОС кафедра КОТ перешла на двухуровневую подготовку выпускников (бакалавр, магистр) направления 09.03.02 – «Информационные системы и технологии». Мы предлагаем для этого образовательную программу «Автоматизация и управление в образовательных системах». Ведется подготовка аспирантов по специальности 05.13.06 - «Автоматизация и управление технологическими процессами и производствами (образование)» по техническим наукам. С 2015 года открыто новое направление подготовки магистров – «Программная инженерия систем реального времени»

Тюгашев Андрей Александрович

Основы программирования. Часть II.

Учебное пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49