3. ТЕХНОЛОГИЯ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ

3.1. Основные этапы решения задач на ЭВМ

Рассмотрим этапы решения задачи на ЭВМ в виде этапов *жизненного цикла* программы как промышленного изделия.

Период существования (разработки и эксплуатации) сравнительно небольшой программы можно разбить на следующие этапы (в скобках указано примерное распределение затрат на разработку между этапами).

1. *Проектирование программы* (17%).

1.1. *Постановка задачи* - точная формулировка решаемой программой задачи в виде *технического задания (ТЗ)* на разработку программы.

1.2. Выбор или разработка метода решения задачи.

1.3. *Алгоритмизация* - проектирование структуры данных и алгоритма программы.

Программа = Данные + Алгоритм

Разработка структуры данных проводится параллельно с разработкой алгоритма, несколько опережая ее: на каждом шаге сначала уточняются данные (операнды), а затем - операции над ними.

2. *Программирование* (8%) - перевод алгоритма на язык программирования – программная реализация алгоритма. Программирование можно рассматривать как последний шаг проектирования.

Иногда этот этап называют *кодированием*, а текст программы – (программным) *кодом*. Эти термины сохранились с тех времен, когда программы вручную кодировались в двоичной системе счисления.

3. *Отладка программы* (25%) - выявление, поиск и исправление ошибок. Отладка завершается официальным *испытанием*, *сдачей* и *опытной эксплуатацией* программы. Затем начинается ее *производство* (копирование программы вместе с документацией), постоянная эксплуатация и сопровождение.

4. *Сопровождение программы* (50%) - устранение ошибок и доработка программы (т.е. продолжение разработки) в течение всего периода ее эксплуатации.

На устранение ошибок при отладке и сопровождении уходит 75% затрат разработки. Образно говоря, один день программисты делают ошибки и три дня их исправляют. Борьба с ошибками - главная проблема программирования. Проще предотвращать ошибки, чем потом выискивать их. Особенно важны начальные этапы разработки: чем раньше сделана и позже обнаружена ошибка, тем больше вреда от нее.

Наиболее изучен самый простой этап разработки - программирование (всего 8% затрат). Ему посвящена почти вся литература (большинство книг называется "Программирование на языке ..."). По более сложным вопросам

алгоритмизации значительно меньше книг, по отладке очень мало, по сопровождению - почти нет, а именно на эти этапы уходит основная доля затрат труда, времени и средств. В исследовании и совершенствовании этих этапов разработки программ скрыты большие резервы экономии затрат.

Ближайшие разделы учебника посвящены методам и средствам представления и разработки алгоритмов [18, 20, 21, 37].

3.2. Структурное программирование

Алгоритмизация - это представление неформального, неточного и неполного описания известного метода решения задачи в виде четкого алгоритма.

Это - не простая проблема. Систематические методы алгоритмизации появились лишь в начале 70-х годов и связаны, прежде всего, с двумя независимыми друг от друга идеями: структурное программирование и разработка сверху вниз.

Эти идеи произвели настоящую революцию в программировании и способствовали его индустриализации. Они лежат в основе современной технологии программирования. В принципе, обе идеи достаточно просты и используются не только в программировании.

Структурное программирование - это метод программирования, в котором используются только алгоритмы, построенные из стандартного набора базовых структур (так называемые *структурные алгоритмы*).

Идея структурного программирования - стандартизация для борьбы с ошибками: ограничить возможную структуру алгоритмов, сделав их более простыми и наглядными (Э. Дейкстра, 1968).

При этом облегчается понимание, разработка, изменение, отладка и верификация алгоритма, уменьшается количество возможных ошибок, упрощается их поиск и, в конечном счете, увеличивается производительность труда программистов и повышается качество программ. В частности, повышаются их надежность и мобильность, упрощается модернизация. При этом, правда, алгоритм может стать более громоздким.

Как и любая стандартизация, структурное программирование рассчитано, прежде всего, на индустриальный подход - коллективную разработку больших и сложных программ в промышленных масштабах. Его роль повышается с ростом размеров и сложности разрабатываемых программ.

Алгоритм называется *структурным* (иногда говорят "структурированным"), если он имеет одну из базовых структур. Каждый блок этих структур сам может иметь внутри любую из этих допустимых структур и т.д. Таким образом из базовых структур можно построить структурный алгоритм любой сложности.

В качестве *базовых структур* обычно используют *последовательность, ветвление и цикл* (рис. 3.1).

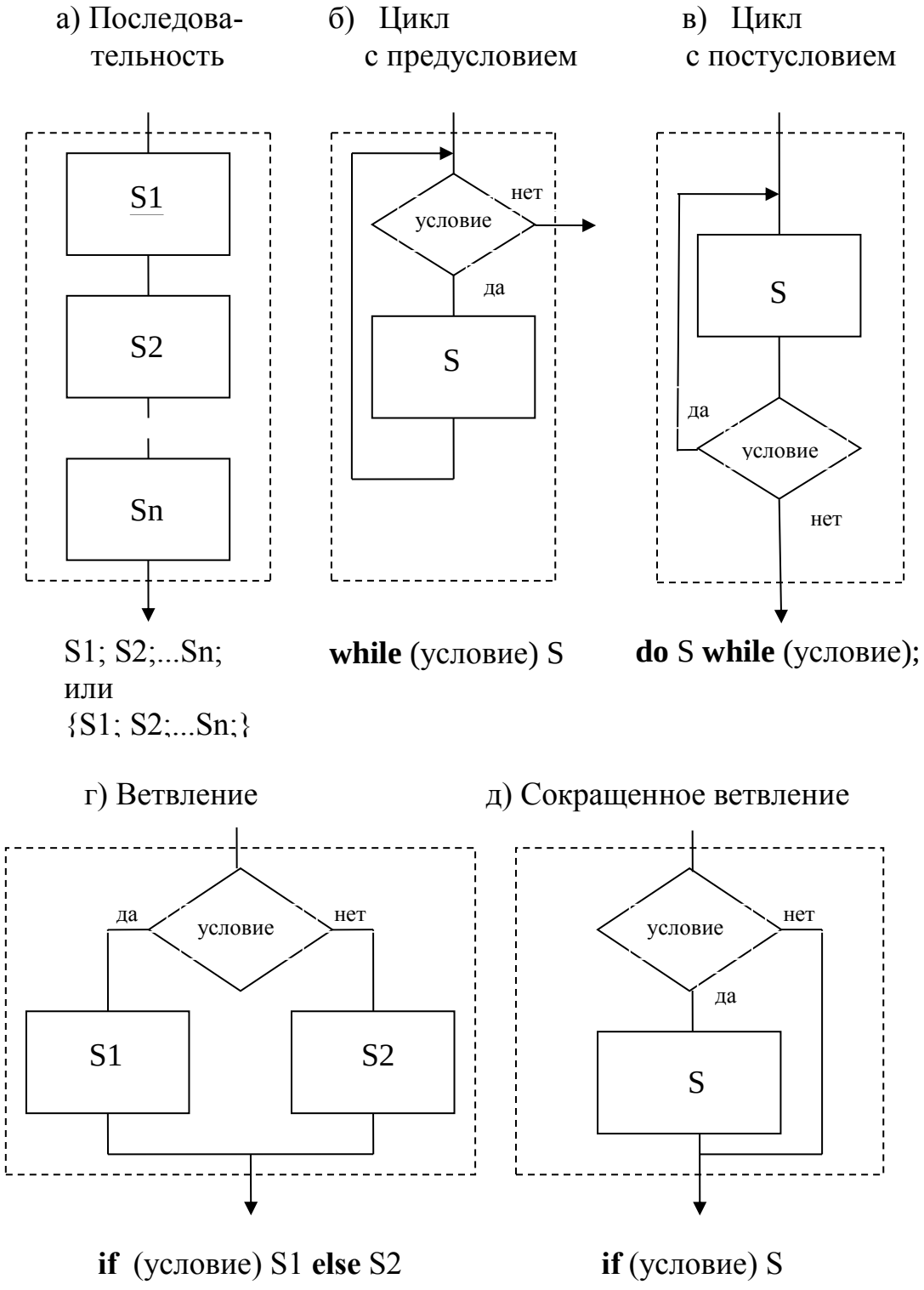


Рис. 3.1. Базовые структуры алгоритма и их запись на языке С
(S - оператор, S1 - оператор, S2 - оператор, Sn - оператор)

Каждая из базовых структур имеет один вход и один выход и может рассматриваться как один блок, показанный на рис. 3.1 пунктирной рамкой.

Для представления любого структурного алгоритма достаточно иметь в языке программирования операторы для записи базовых структур. На рис. 3.1 показаны такие операторы языка С.

Последовательная структура в языке С представляется просто последовательной записью операторов.

Если требуется написать последовательность операторов там, где по правилам языка должен быть один оператор (например, внутри цикла или ветвления), эта последовательность заключается в фигурные *операторные скобки*, превращающие ее в *составной оператор.*

В языке Pascal и ряде других языков такими операторными скобками являются служебные слова **begin** и **end.**

Ветвление в языке С записывается с помощью *условного оператора* (оператора **if**), имеющего вид:

if (выражение) оператор **else** оператор

или *сокращенного (неполного) условного оператора:*

if (выражение) оператор

Сокращенное ветвление - это частный случай ветвления, когда в одном из возможных вариантов не требуется ничего делать - *пустой оператор.*

Для представления *цикла с предусловием* служит оператор **while**:

while (выражение) оператор

Для *цикла с постусловием* используется оператор **do-while**:

do оператор **while** (выражение);

Для построения программ достаточно одного из циклов: каждый из них можно выразить через другой цикл и ветвление.

В большинстве языков программирования используется только *цикл с предусловием*. Он более универсален, т. к. может иметь нулевое количество повторений.

Цикл с постусловием повторяется хотя бы один раз, и его можно использовать только в случаях, когда не бывает нулевого числа повторений, но в этих случаях он экономнее цикла с предусловием.

Начинающим программистам рекомендуется избегать использования цикла с постусловием, т. к. оно требует определенного опыта. В языках С и Pascal имеются оба эти цикла.

В циклах и ветвлениях языка С условие записывается в виде заключенного в скобки целочисленного выражения. Ненулевое значение выражения означает *истинность условия*, нулевое - *ложность*.

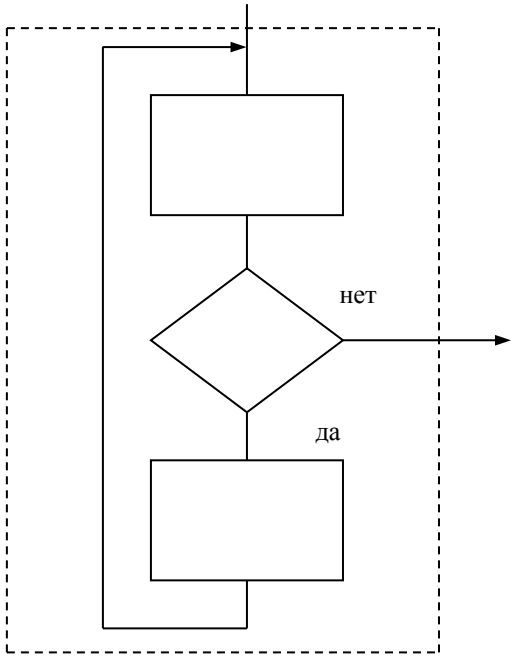
В рассмотренных операторах используются английские *служебные слова* языка С: **if** - если, **else** - иначе, **while** - пока, **do** - выполнять. Для наглядности они выделены полужирным шрифтом.

Подобные операторы для структурного программирования есть и в ряде других современных языков, например, Pascal.

Примеры структурных алгоритмов приведены на рис. 3.4, 3.5, 3.8, 3.9.

На рис. 3.2 показаны некоторые разновидности из бесконечного множества неструктурных циклов, на рис. 3.7 - неструктурное ветвление. Любой алгоритм, содержащий подобные циклы или ветвления, является неструктурным.

а) Цикл с выходом изнутри тела цикла



б) Цикл с двумя выходами

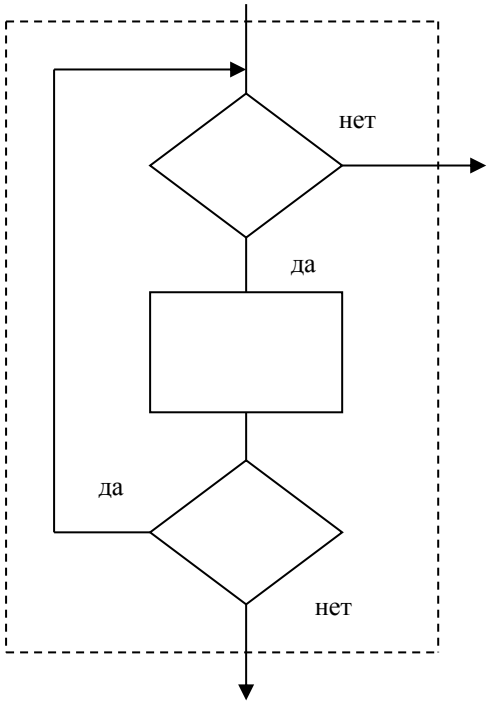


Рис. 3.2. Некоторые разновидности неструктурных циклов

Два алгоритма (программы) *функционально эквивалентны*, если они реализуют одну и ту же программную функцию - преобразование исходных данных в результат. Два алгоритма (программы) *эквивалентны по выполнению*, если они выполняют одинаковые операции в одинаковой последовательности.

Справедлива следующая ***теорема структуризации*** (К. Боэм и Г. Джакопини, 1966). Для произвольного алгоритма с одним входом и одним выходом существует функционально эквивалентный структурный алгоритм.

Другими словами, если для решения задачи существует алгоритм, то существует и структурный алгоритм. Таким образом, метод структурного программирования не ограничивает множество решаемых задач.

Доказательство. Рассмотрим произвольную программу (алгоритм), схема которой содержит произвольным образом связанные функциональные и предикатные блоки (прямоугольники и ромбы) с номерами 1, 2, ... m. Дадим первому исполняемому блоку номер 1, а выходному блоку - номер 0.

Добавим в исходную программу переменную n, обозначающую номер текущего выполняемого блока. Каждому блоку номер i исходной программы сопоставим блок Gi, который выполняет такие же операции, как и блок номер i, а кроме того вычисляет номер следующего блока n. Это можно сделать следующим образом.

Если исходный блок является функциональным блоком (оператором) Si, за которым выполняется блок номер j, то Gi имеет вид:

```
{ Si; n = j; }
```

Если исходный блок представляет собой предикатный блок (ромб) с условием Pi, за которым при соблюдении условия Pi выполняется блок номер j, а при его нарушении - блок номер k, то на языке C/C++ его можно записать в виде

```
if (Pi) goto блок j;  else  goto блок k;
```

В этом случае Gi имеет вид:

```
if (Pi) n=j; else n=k;
```

Тогда исходной программе будет функционально эквивалентна следующая структурная программа с одним циклом и m операторами **if-else**.

```
n=1;
while (n > 0)
  if (n == 1)       G1
  else if (n == 2)  G2
    . . .
  else if (n == m-1) Gm-1
  else if (n == m)   Gm
  else;
```

Эта программа выполняет блоки вида Si и Pi в том же порядке и поэтому получает такой же результат, как и исходная программа.

Интересным следствием этого доказательства является тот факт, что любой алгоритм можно представить в виде функционально эквивалентного алгоритма с одним циклом.

Кроме рассмотренного выше набора основных базовых структур, иногда используют дополнительные структуры. В языке PDL, например, допускается цикл из рис. 3.2 а. Возможны и другие наборы базовых структур.

Структурное программирование иногда не совсем правильно называют «программирование без **goto» -** без оператора перехода, который во многих языках служит для организации циклов и ветвлений (go to - перейти к).

Действительно, бессистемное применение оператора **goto** запутывает программу, а любой (структурный) алгоритм можно записать без использования **goto**, например, с помощью условного оператора и цикла с предусловием.

Однако идея структурного программирования - не сама по себе борьба против оператора **goto**, а стремление сделать программу проще, нагляднее и надежнее за счет стандартизации ее структуры.

Поэтому, с одной стороны, в структурном программировании допускается ограниченное применение **goto** в особых ситуациях, когда оно способствует простоте и наглядности программы: не для организации обычных циклов и ветвлений (пример 2.1).

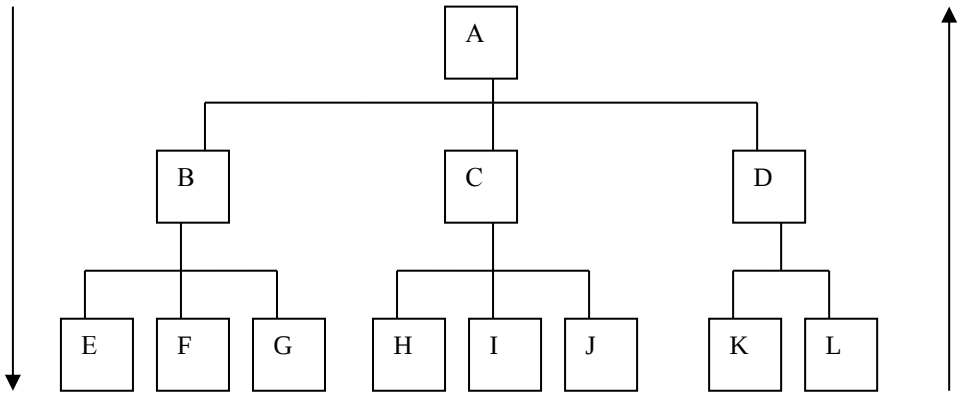
С другой стороны, несмотря на отсутствие условного оператора и цикла с предусловием, структурное программирование возможно и на таких языках, на которых ни одно ветвление или цикл не напишешь без **goto**, например, старых версиях языков Fortran и Basic или даже языках ассемблера.

В этих случаях используют только структурные алгоритмы, а для каждой базовой структуры – какой-нибудь единообразный стандартный способ ее записи с помощью **goto**.

3.3. Разработка сверху вниз и снизу вверх

Любая система состоит из частей, каждую из которых также можно разложить на составные части и т. д. Например, университет делится на факультеты, факультеты включают кафедры и курсы, разделенные на лаборатории и группы, которые, в конечном счете, состоят из сотрудников и студентов.

Структуру самых разнообразных систем: технических, организационных, биологических, грамматических и т. п. часто изображают в виде дерева, обычно с корнем наверху (рис. 3.3).



Сверху вниз Снизу вверх

Рис. 3.3. Древовидная (иерархическая) структура алгоритма

Таким же образом можно представить любой алгоритм или программу, разбивая их на более мелкие алгоритмы, блоки и т. д. до команд.

Каждый узел дерева обозначает некоторый алгоритм. Отходящие от него вниз линии ведут к составным частям этого алгоритма. Так, на рис. 3.3

алгоритм A включает в себя алгоритмы B, D, C; алгоритм D, в свою очередь, состоит из K и L, и т. д.

Алгоритм можно разрабатывать, двигаясь по древовидной структуре *сверху вниз*: от целого к деталям, или *снизу вверх*: от частей к целому, т. е. в *нисходящем* или *восходящем* направлении.

Аналогичным образом можно выполнять построение, исследование, изучение, описание и другие действия с любой системой. Например, на лекциях, как в данной книге, материал может излагаться, в основном, сверху вниз: от более крупных понятий к деталям. Параллельно, на практических и лабораторных занятиях, программирование изучается снизу вверх: от примеров простых программ и отдельных деталей языка С к их обобщению и использованию в более крупных и сложных программах.

Разработка сверху вниз начинается от главной цели: на каждом этапе разработки решаемая задача (поставленная цель) разбивается на более простые подзадачи (подцели), с которыми затем поступают таким же образом.

Так, при написании книги или сочинения сначала составляется план: книга разбивается на части; а затем уже уточняются детали, т. е. пишутся планы и тексты этих частей.

На первом этапе проектируемый алгоритм представляется в виде одного блока. Затем определяется структура этого блока (например, выбирается одна из базовых структур структурного программирования). Таким образом, исходный алгоритм разбивается на части.

Далее разработка продолжается аналогично: каждый блок разбивается на более мелкие действия, пока весь алгоритм не будет разложен на достаточно простые операции, "понятные" процессору (имеющиеся в выбранном языке программирования).

Другие названия нисходящей разработки: *последовательное уточнение, пошаговая детализация* и т. п. Принцип такого подхода можно выразить словами "разделяй и властвуй".

В разделе 1.7 алгоритм разрабатывался снизу вверх: простые операции, из которых состоял алгоритмический процесс (1.2), объединялись в более крупные алгоритмы (блоки, шаги). Из этих блоков затем был составлен структурный алгоритм 1.2.

Восходящий метод менее удобен, чем нисходящий, т. к. трудно заранее предугадать, из каких мелких частей будет состоять искомый алгоритм. Это подобно тому, как, не имея общего плана сочинения, писать его отдельные страницы: многие из них потом могут оказаться ненужными, и большую часть придется переписывать.

Структурное программирование обычно сочетают с проектированием алгоритма сверху вниз. Иногда даже считают разработку сверху вниз частью структурного программирования. Это не совсем правильно: структурное программирование возможно и снизу вверх.

Действительно, базовые структуры можно рассматривать как допустимые в структурном программировании способы и для разделения

сложной операции на более простые, и, наоборот, для составления из мелких операций более крупной операции.

Нисходящая и восходящая разработка программы имеют свои достоинства и недостатки и их надо умело сочетать, но в целом предпочтительнее подход сверху вниз, как более целенаправленный.

3.4. Простой пример: максимум из трех чисел

Рассмотрим простой пример разработки алгоритма методом структурного программирования сверху вниз.

Пример 3.1. Присвоить X наибольшее из значений A, B или C.

На первом этапе весь алгоритм (назовем его MAX3A) можно представить, хотя бы мысленно, в виде одного блока с единственной операцией "Решить задачу...", в данном случае имеющей вид X=max(A, B, C), как показано на рис. 3.4 а.

Эта операция отсутствует в языке C. Для ее детализации анализируется процесс получения результата и определяется, какую из базовых структур имеет этот блок. Возможны разные варианты: MAX3A и MAX3B.

Решение А. Можно присвоить вспомогательной переменной D большее из чисел A и B, а затем найти максимум из C и D. Тогда алгоритм MAX3A примет последовательную структуру (рис. 3.4 б).

Исходная задача нахождения наибольшего из трех чисел сведена к двум более простым задачам: найти максимальное из двух чисел. Эта операция также отсутствует в языке C, т. е. требует дальнейшей детализации.

Наибольшим из двух чисел может быть либо первое, либо второе число. Поэтому данную операцию можно представить в виде ветвления. Алгоритм MAX3A приобрел структуру из двух последовательных ветвлений (рис. 3.4 в).

Вместо D можно использовать X: ведь значение X все равно изменится, а в задаче оно не используется. Заменив D на X, получим в одной из ветвей оператор X=X, который не изменяет значения переменных (как говорят, эквивалентен пустому оператору). Удалив его, получим алгоритм MAX3A, состоящий только из имеющихся в языке C присваиваний и сравнений, т. е. не требующий дальнейшей детализации (рис. 3.4 г).

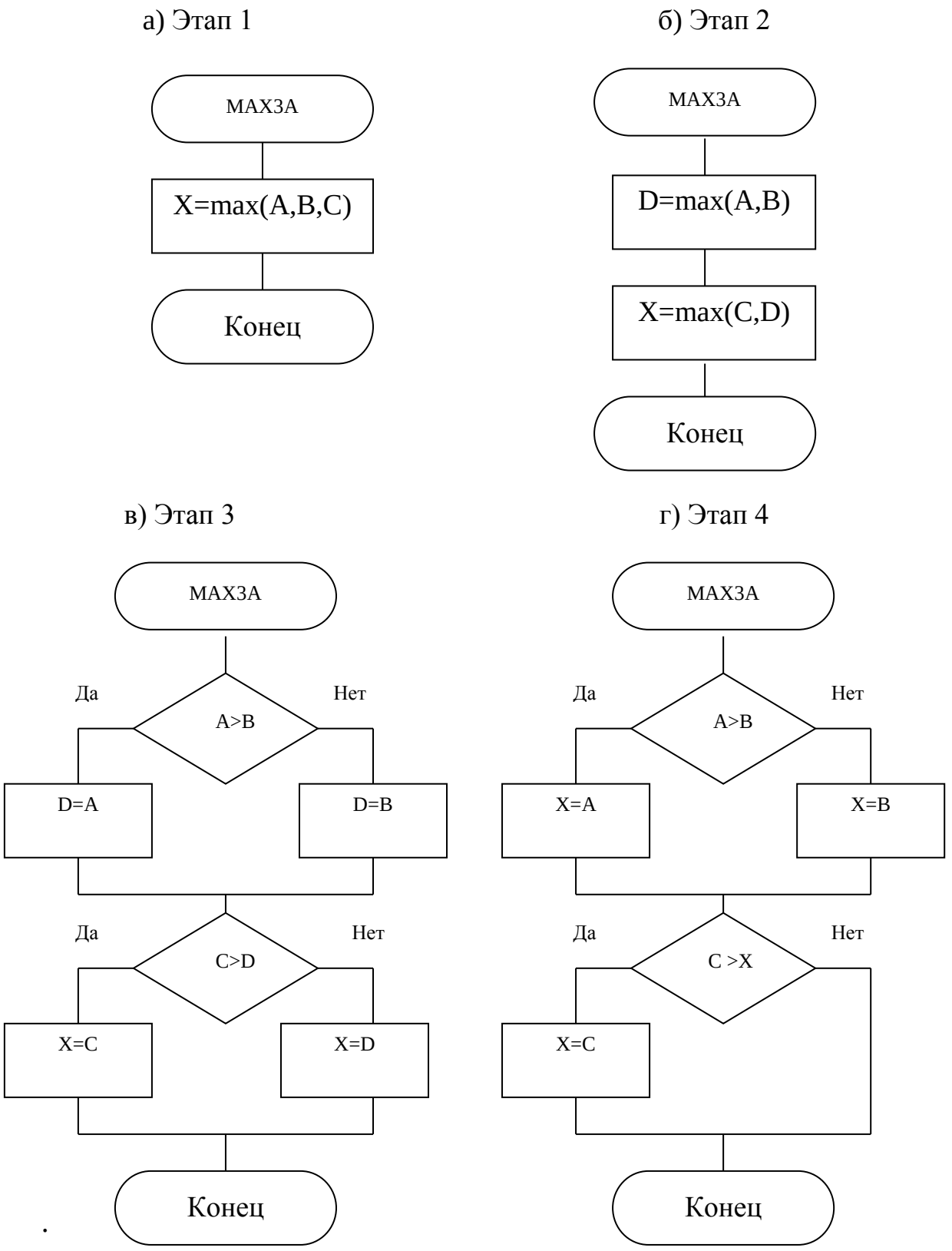


Рис. 3.4. Этапы разработки алгоритма MAX3A (решение А)
Этап 1. Исходный алгоритм; Этап 2. Определение структуры алгоритма; Этап 3. Определение структуры блоков; Этап 4. Устранение вспомогательной переменной

Решение В. Если, например, A > B, то наибольшим из трех чисел может быть A или C. В противном случае (A ≤ B) наибольшее число необходимо выбрать из B и C. Наличие двух вариантов действий позволяет представить проектируемый алгоритм MAX3B в виде структуры ветвления (рис 3.5 б).

Исходная задача сведена к более простым задачам нахождения максимального из двух чисел. Представив их в виде ветвлений, получим окончательный вариант алгоритма MAX3B (рис. 3.5 в).

Как видно из этого примера, структура блока может определяться неоднозначно, и структурное программирование облегчает поиск ее возможных вариантов, ограничивая его базовыми структурами.

Вопрос. Подумайте, например, нельзя ли было использовать другую структуру для нахождения максимального из двух чисел?

Если задача не вызывает проблем, можно сразу составить алгоритм и не требуются какие-либо особые методы. Возникновение трудностей говорит о преждевременном углублении в детали - детализируйте алгоритм постепенно: сначала через крупные операции, а затем уже через более мелкие.

Выше рассмотрена самая медленная детализация, на каждом шаге которой уточняется лишь один блок в виде одной базовой структуры. Каждый этап реального проектирования объединяет столько подобных шагов, сколько удобно разработчику.

Возникает вопрос, какое из полученных решений лучше, MAX3A или MAX3B?

Оценим их, прежде всего, по времени и памяти (рис. 3.6).

Из схемы (рис. 3.5 в) видно, что при любом пути выполнения алгоритма MAX3B выполняются два сравнения и одно присваивание. В алгоритме MAX3A (рис. 3.4 г) выполняется сравнение, присваивание, сравнение и, возможно, еще одно присваивание в зависимости от условия C > X. На рис. 3.6 это возможное присваивание показано пунктирным прямоугольником.

Таким образом, при одних исходных данных оба алгоритма выполняют одинаковые по времени операции, при других - MAX3A выполняет дополнительное присваивание. В среднем MAX3A медленнее, чем MAX3B.

MAX3A содержит два сравнения и три присваивания. MAX3B более громоздкий: три сравнения и четыре присваивания, и для его хранения нужно больше памяти. Оба варианта алгоритма не используют вспомогательные переменные и приблизительно одинаковы по трудоемкости разработки.

Таким образом, MAX3B быстрее, но требует больше памяти; MAX3A, наоборот, эффективней по памяти, но менее эффективный по времени. Поэтому нельзя однозначно ответить на вопрос, какой из двух алгоритмов лучше! Наилучшего алгоритма не существует!

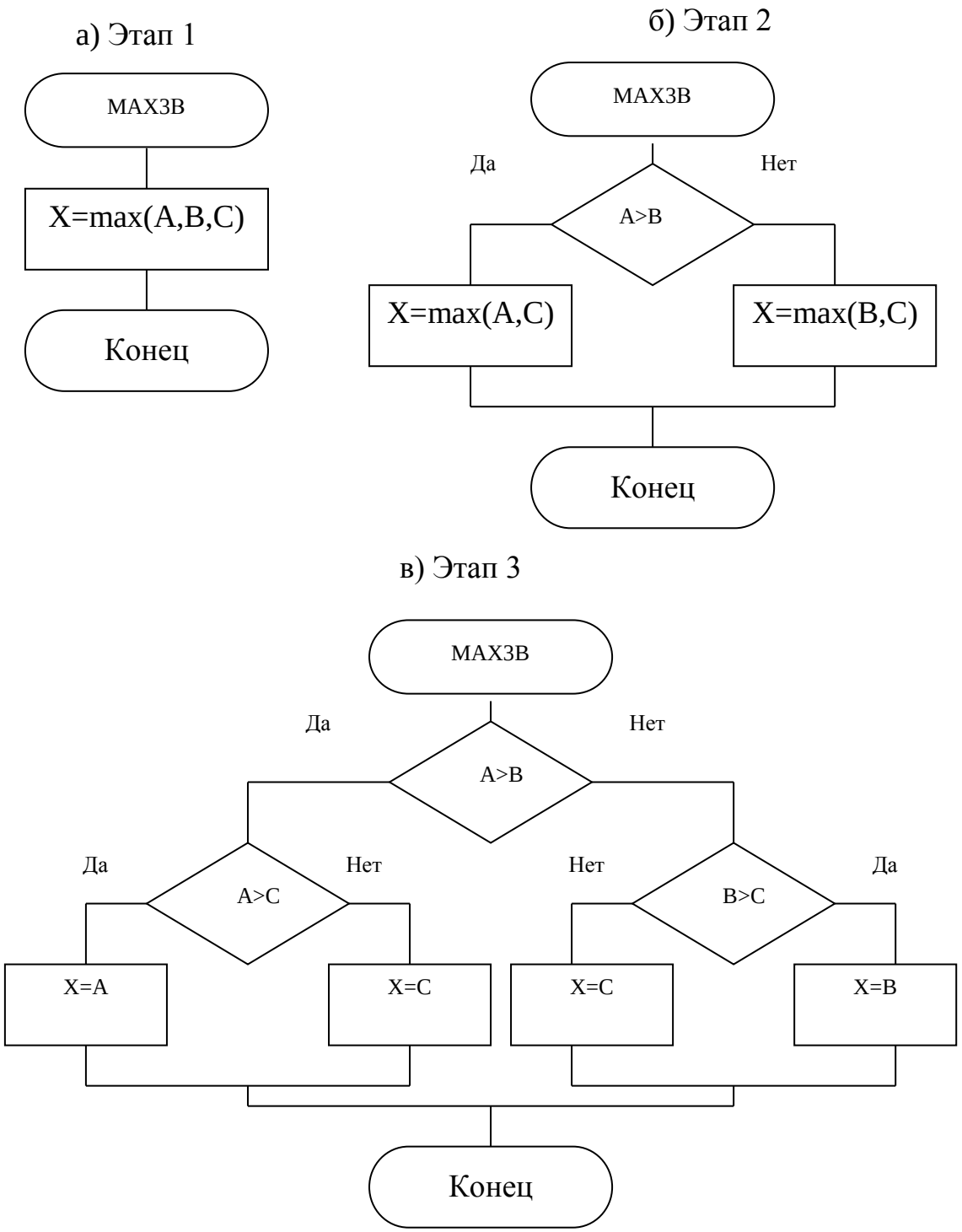


Рис. 3.5. Этапы разработки алгоритма MAX3B (решение B)
Этап 1. Исходный алгоритм; Этап 2. Определение структуры алгоритма;
Этап 3. Определение структуры ветвей

Можно было бы объединить в один блок две одинаковые ветви X = C алгоритма MAX3B (рис. 3.5 в), сократив его без изменения процесса выполнения. Однако полученный таким образом алгоритм MAX3C (рис. 3.7) является неструктурным и поэтому более запутанным и менее надежным,

чем MAX3B. Две его ветви “срослись”, и его труднее понять; при изменениях одной ветви может нарушиться работа другой.

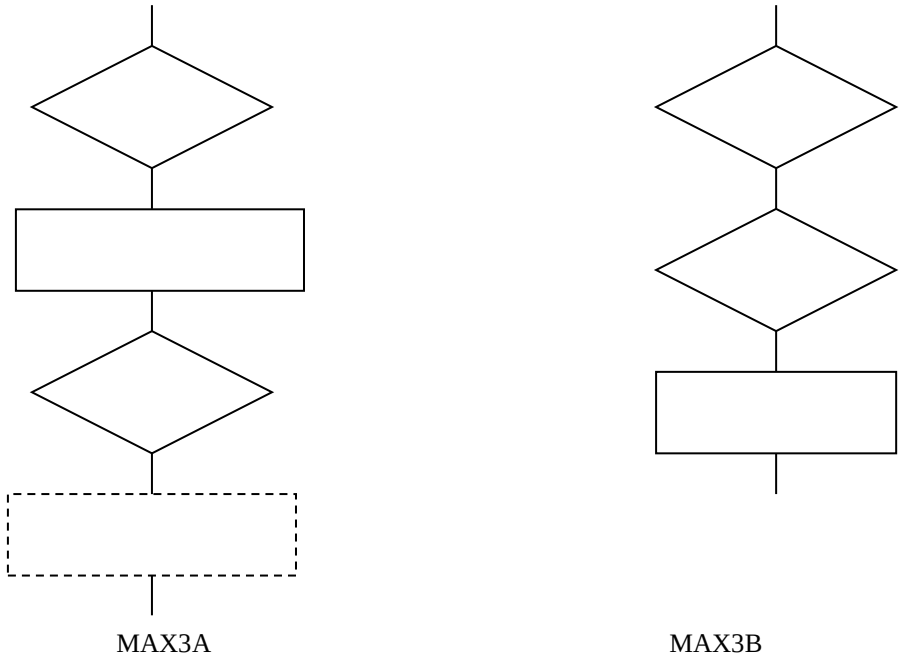


Рис. 3.6. Сравнение алгоритмов MAXA (рис. 3.4 г) и MAXB (рис. 3.5 в) по времени выполнения

В отказе от подобной экономии, наносящей ущерб простоте и надежности, и заключена идея структурного программирования, хотя здесь проявляется и определенный недостаток этого метода.

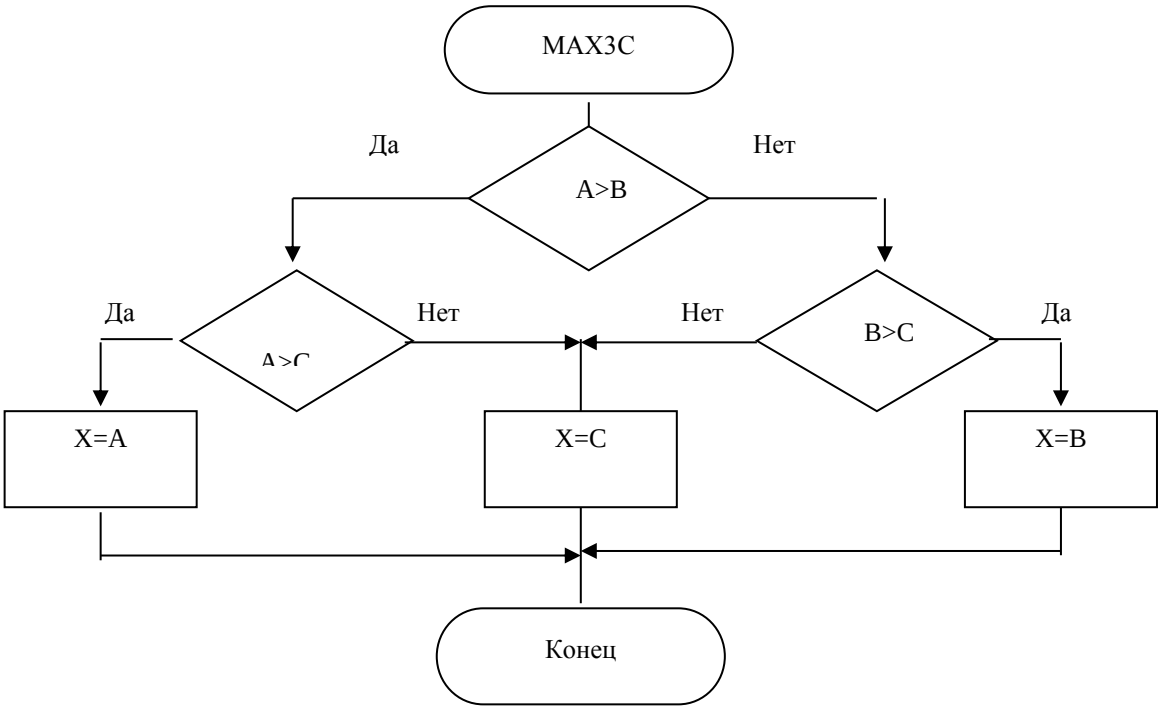


Рис. 3.7. Неструктурный алгоритм MAX3C

В этом простом примере, как в капле воды, отразились весьма характерные и важные *закономерности программирования*.

1. Существует много решений даже очень простой задачи.

2. Структурное программирование сверху вниз облегчает поиск вариантов алгоритма (ограничивая их базовыми структурами).

3. Наилучшая программа решения даже простой задачи не существует! По разным критериям лучшими оказываются разные программы, необходим поиск компромиссных вариантов.

4. Выиграешь время - проиграешь память и наоборот: экономя память, увеличишь время решения задачи.

5. Неструктурный алгоритм может оказаться компактнее структурного, но он обычно бывает более запутанным и менее надежным.

3.5. Пример разработки программы

Пример 3.2. Составить программу решения уравнения

$$\cos x = x \qquad (3.1)$$

На этом примере подробно рассмотрим основные этапы разработки программы методом структурного программирования сверху вниз.

1. Постановка задачи.

Уравнение (3.1) не имеет аналитического решения (в виде общей формулы для x). Поэтому требуется находить приближенное значение x с погрешностью не более заданной величины e.

Обозначив $F(x) = \cos x - x$, из (3.1) получим уравнение

$$F(x) = 0 \qquad (3.2)$$

Если функция F(x) непрерывна на отрезке [a; b] и в его концах имеет разные знаки: $F(a)*F(b) < 0$, то она имеет на этом отрезке хотя бы один корень (в общем случае - любое нечетное количество корней). Для уравнения (3.1) можно взять $a = 0$, $b = \pi/2$ или $b = 1$, т. к. $\cos(0) - 0 = 1 > 0$, а $\cos(1) - 1 < 0$.

2. Выбор метода решения задачи

Корень можно уточнить с любой заданной погрешностью, например, следующим методом *дихотомии* - деления пополам (не будем сейчас заниматься математической проблемой поиска лучшего метода).

Находится середина исходного отрезка, и в качестве нового отрезка выбирается та его половина, где находится корень (на ее концах функция F(x) имеет разные знаки).

Описанное разбиение отрезка пополам повторяется, пока не будет достигнута требуемая точность. Это обязательно произойдет, поскольку при каждом разбиении длина содержащего корень отрезка уменьшается в два раза. Рано или поздно она станет меньше любой заранее заданной допустимой погрешности.

Если корней несколько, будет найден один из них.

3. Алгоритмизация

Алгоритмический процесс часто содержит повторяющуюся последовательность операций - циклическую часть. Обычно перед циклом выполняются некоторые подготовительные действия, а после него - завершающие действия. Такой алгоритм в общем случае имеет последовательную структуру из трех частей: Подготовка, Цикл и Завершение. Назовем ее *обобщенным циклом* (см., например, рис. 3.8 б).

Аналогично, если процесс содержит разные варианты действий, то его всегда можно представить в виде *обобщенного ветвления* из трех последовательных операций: Подготовка, Ветвление и Завершение. Такой пример показан на рис. 3.8 в.

Таким образом, при разработке алгоритма сверху вниз удобно представлять циклические и разветвляющиеся процессы в виде обобщенного цикла или ветвления, заранее предусматривая операции подготовки и завершения, о которых программисты часто забывают. При последующем уточнении подготовка и/или завершение могут оказаться и пустыми.

Определив структуру процесса, удобно прежде всего изобразить всю эту структуру с пустыми блоками, а затем уже вписывать в них операции.

В обобщенном цикле и ветвлении удобнее сначала уточнять смысл основного действия (условия и тела цикла или ветвления), а затем детализировать подготовительные и завершающие действия, зависящие от основной части, которая может определяться по-разному.

Например, процесс выполнения обобщенного цикла с предусловием

```
Подготовка
while (Условие)
    Тело цикла
Завершение
```

имеет вид:

```
                Подготовка
              ⎧ Условие? - да
              ⎩ Тело цикла
Циклическая   ⎧ Условие? - да
  часть       ⎩ Тело цикла
(повторяется        . . .
  ≥ 0 раз)    ⎧ Условие? - да
              ⎩ Тело цикла
                Условие? - нет
                Завершение
```

Здесь видны составные части, которые нужно выделить в проектируемом процессе, чтобы правильно сформулировать алгоритм. Проверку условия

можно располагать в разных точках процесса, а с нее начинается циклическая часть, которая поэтому выделяется не однозначно.

Сначала программа вводит и проверяет исходные данные. Метод деления пополам не гарантирует нахождение корня, когда F(x) имеет одинаковые знаки на концах исходного отрезка, т. к. в таком случае он может содержать любое четное, в том числе нулевое, количество корней.

В этом случае программа выдает сообщение об ошибке. При правильных исходных данных происходит уточнение корня методом деления пополам (рис. 3.8 а).

В укрупненном алгоритме трудно и нецелесообразно сразу детализировать процесс уточнения корня. Соответствующий блок получает название УТОЧНЕНИЕ из заглавных букв, которое по ГОСТу должно подчеркиваться, а подробности этой операции раскрываются отдельной схемой, в начальном овале которой указывается название детализируемого блока (рис. 3.8 б).

Процесс решения уравнения методом деления пополам содержит повторяющиеся действия. Поэтому блок УТОЧНЕНИЕ имеет структуру обобщенного цикла. Тело цикла - блок РАЗБИЕНИЕ (разбиение отрезка пополам) детализируется отдельной схемой (рис. 3.8 в).

Разбиение пополам продолжается, пока погрешность превышает допустимое значение e. Для детализации этого условия, а также подготовительных и завершающих действий блока УТОЧНЕНИЕ, требуется разработать структуру данных - определить величины, участвующие в процессе уточнения корня.

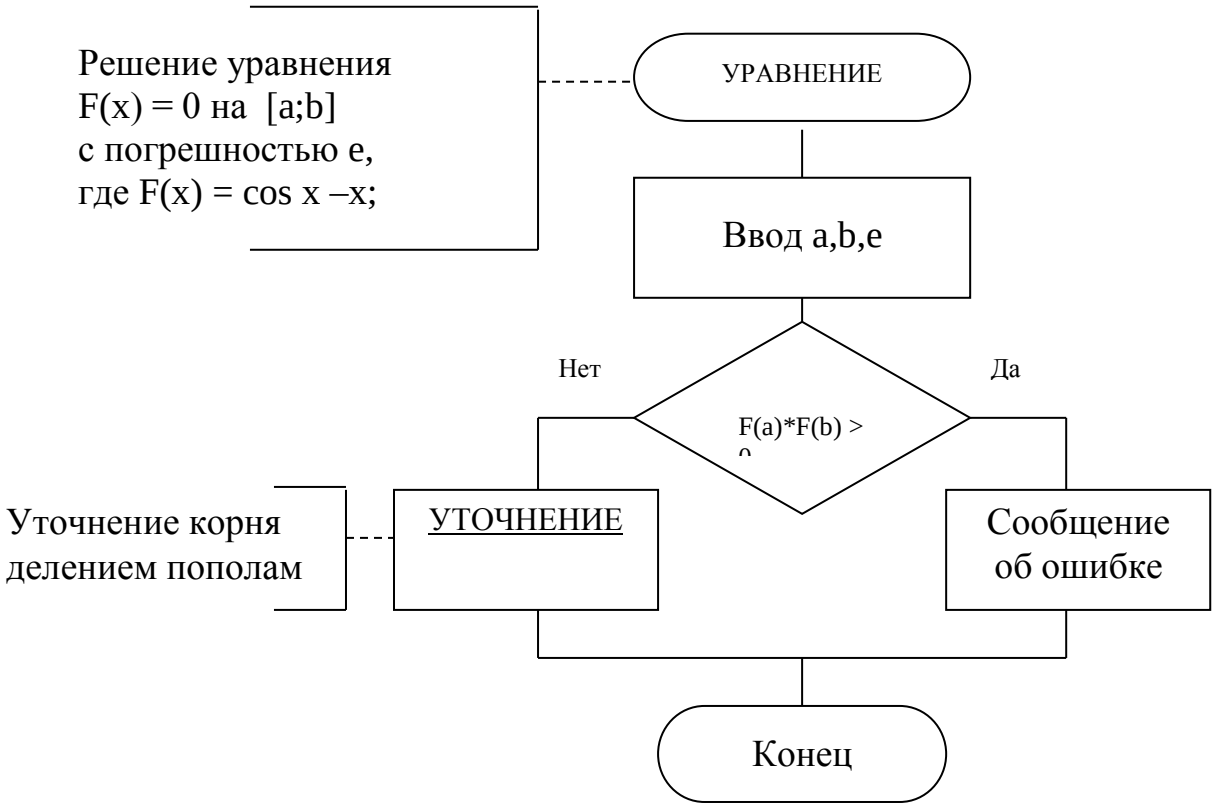
Обозначим:

l - координата (абсцисса) левого конца текущего отрезка,
p – координата (абсцисса) правого конца текущего отрезка,
s - координата (абсцисса) середины текущего отрезка.

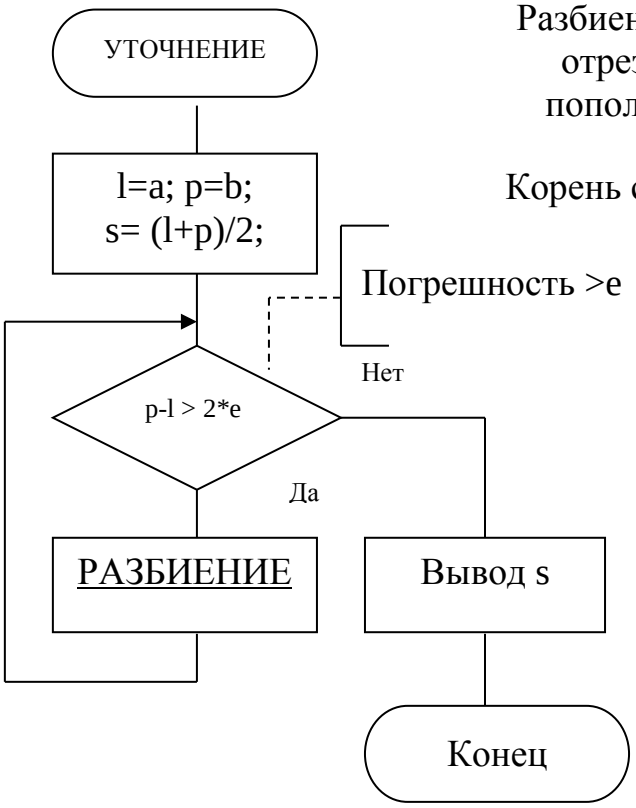
Приближенным значением корня естественно считать середину s отрезка, где он находится. Тогда погрешность не превысит половины длины отрезка (p - l) / 2, и условие повторения цикла можно записать в виде $p - l > 2*e$.

При достижении заданной точности завершается цикл и выводится приближенное значение корня s.

а) Укрупненный алгоритм



б) Схема блока УТОЧНЕНИЕ



в) Схема блока

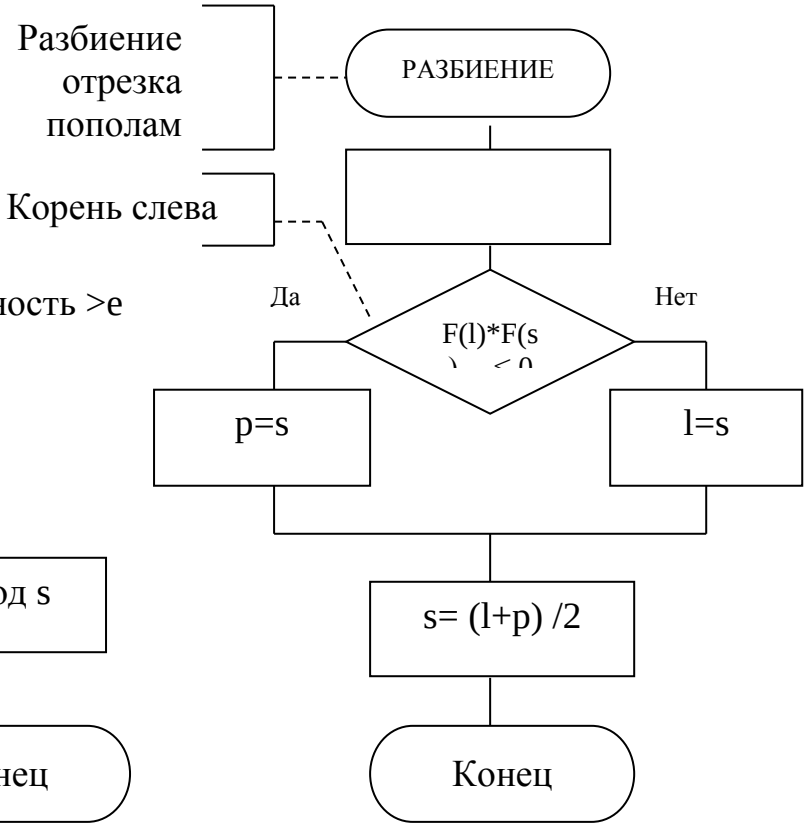


Рис 3.8. Поэтапная разработка алгоритма решения уравнения F(x) = 0

Подготовка - это присваивание переменным l, p и s начальных значений - концов и середины исходного отрезка [a; b].

Блок РАЗБИЕНИЕ включает выбор той половины текущего отрезка, где находится корень, и поэтому имеет структуру обобщенного ветвления (рис. 3.8 в).

Левая половина отрезка, например, будет выбрана, если функция F(x) имеет разные знаки в ее концах (F(l)*F(s)<0) или равна нулю в одном из них (F(l)*F(s)=0). Отсюда - условие выбора левой половины: $F(l)*F(s) \le 0$.

При выборе левой половины левый конец текущего отрезка не изменяется, а правый конец перемещается в середину: p = s. При выборе правой половины перемещается левый конец: l = s.

Подготовка или завершение блока РАЗБИЕНИЕ должны содержать вычисление середины s нового отрезка: s = (l + p) / 2;. При первом входе в блок РАЗБИЕНИЕ s уже вычислена в подготовке блока УТОЧНЕНИЕ, и поэтому ее новое значение можно вычислять в завершении блока РАЗБИЕНИЕ. Подготовка блока РАЗБИЕНИЕ остается пустой (рис. 3.8 в).

Возможен и другой вариант алгоритма. В условии цикла блока УТОЧНЕНИЕ s не используется. Поэтому вычисление s можно перенести из подготовки блока УТОЧНЕНИЕ в подготовку блока РАЗБИЕНИЕ, а завершение блока РАЗБИЕНИЕ сделать пустым.

Тогда при выходе из цикла s не будет соответствовать последнему отрезку, и придется также вставить вычисление (l + p) / 2 в завершение блока УТОЧНЕНИЕ. Таким образом, этот вариант алгоритма не дает выигрыша по сравнению с приведенным на рис. 3.8.

Полученный алгоритм можно собрать в одну схему (рис. 3.9), а можно оставить и в первоначальном виде (рис. 3.8).

Вообще, желательно размещать схему программы на одной странице. Если она не помещается на странице, ее следует укрупнить, объединяя по несколько блоков в более крупный блок. На других страницах таким же образом размещаются схемы полученных блоков. В результате алгоритм любой программы будет состоять из отдельных схем, не превышающих страницы.

На практике для разработки алгоритма вместо схем удобнее использовать *псевдокод* – неформальный вариант языка программирования, представляющий собой сочетание языка программирования с естественным языком и любыми удобными обозначениями. Примеры использования псевдокода приведены ниже и в последующих разделах.

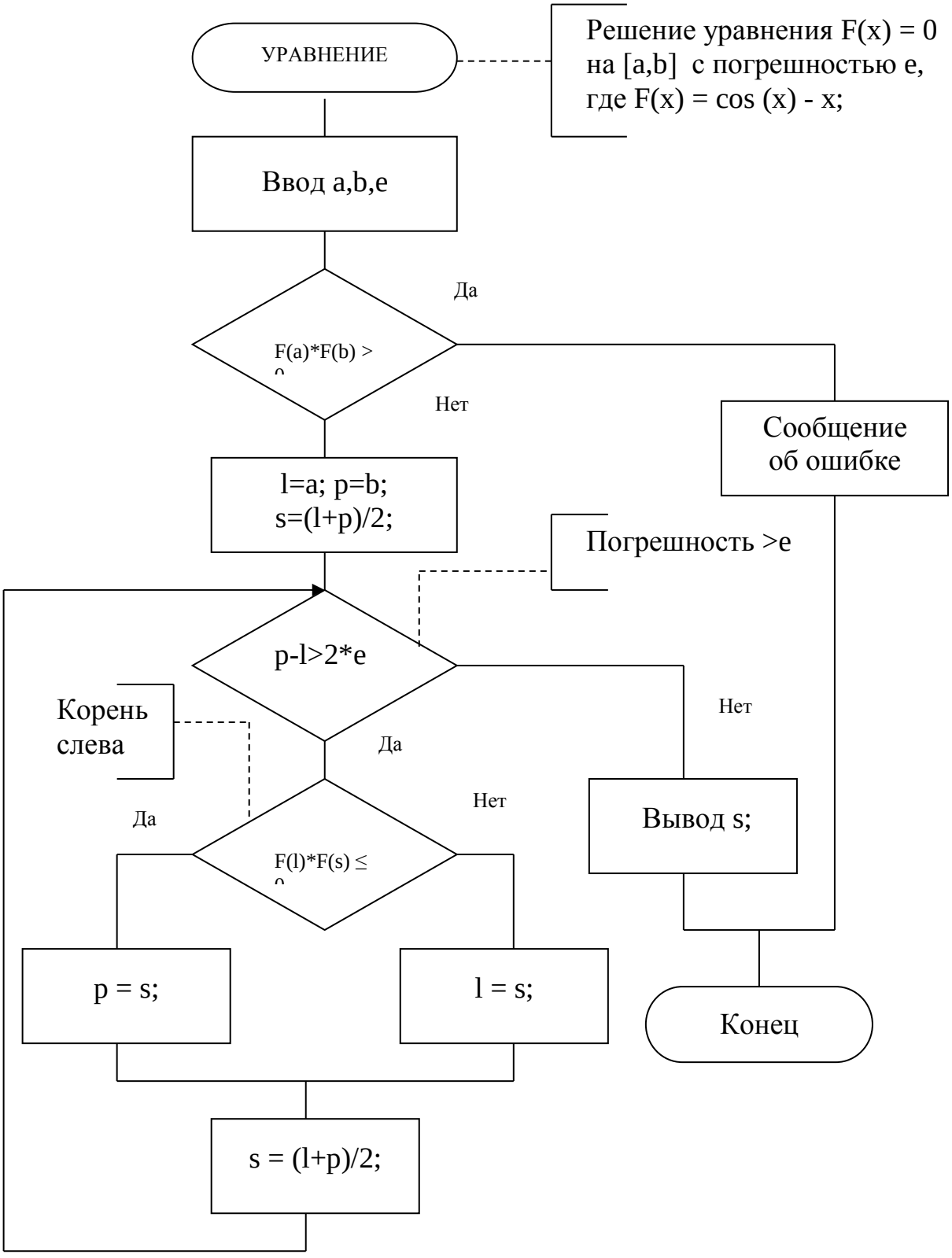


Рис.3.9. Схема программы решения уравнения F(x) = 0

Например, укрупненный алгоритм, соответствующий схеме, показанной на рис. 3.8 а, можно записать на псевдокоде, основанном на языке С, вместе с определением переменных следующим образом.

```
/*  Решение уравнения F(x)=0 на [a; b] с погрешностью e   */
/*  методом деления пополам   (где F(x) = cos x  - x )    */

float a, b;          /* Концы исходного отрезка            */
float e;             /* Допустимая погрешность             */
float l, p, s;       /* Концы и середина текущего отрезка  */

Ввод a,b,e;
if (F(a) * F(b) > 0)          /* Корень может отсутствовать  */
   Вывод сообщения об ошибке;
else
   Уточнение корня делением пополам;
```

Правильное использование комментариев и других средств языка для повышения наглядности программ составляет понятие хорошего *стиля программирования*, существенно влияющего на качество программ, в том числе их надежность. Необходимо, как минимум, комментировать назначение программы и ее основных частей, смысл всех использованных в ней величин, а также инварианты циклов и другие ключевые идеи алгоритма. Важно также помнить, что комментарии могут быть излишними, если среди них трудно увидеть операторы программы!

Для наглядности программы на языке С (и других подобных языках) рекомендуется каждый оператор начинать с новой строки. Строка может также содержать несколько коротких операторов.

Начало внутренних операторов в цикле, условном операторе и составном операторе рекомендуется сдвигать вправо на ступеньку из 2 - 3 символов по отношению к охватывающему оператору.

Для более четкого выделения соответствующих пар операторных скобок закрывающая скобка обычно пишется либо в одной строке с открывающей, либо в той же позиции одной из последующих строк.

Такая *ступенчатая запись* подчеркивает структуру вложенности операторов, облегчает чтение программы и является важным элементом хорошего стиля программирования. Детали такой записи могут быть различными.

Вывод сообщения легко записывается на языке С и поэтому не требует детализации. Операцию уточнения корня детализируем на псевдокоде отдельным алгоритмом (заменяющим схему блока УТОЧНЕНИЕ на рис. 3.8 б).

Для наглядности поместим перед ним в виде заголовочного комментария текст соответствующего неформального оператора:

```
/* Уточнение корня делением пополам          */
l=a;  p=b;  s=(l+p)/2;
while (p-l > 2*e)       /* Погрешность >e    */
   Разбиение отрезка пополам;
Вывод корня s;
```

Аналогичным образом, вместо схемы (рис. 3.8 в), раскроем на псевдокоде алгоритм операции разбиения отрезка пополам:

```
/* Разбиение отрезка пополам                  */
if (F(l)*F(s) <= 0)     /* В левой части меняется знак */
   p = s;               /* Выбор левой половины       */
else
   l = s;               /* Выбор правой половины      */
s = (l + p) / 2;
```

Записанные на псевдокоде алгоритмы легче преобразовать в программу на языке С, чем схемы. При необходимости можно предварительно собрать их в единый алгоритм, подобный схеме из рис. 3.9:

```
/*  Решение уравнения F(x) = 0 на отрезке [a; b] с погрешностью  e */
/*  методом деления пополам   (где F(x) = cos x  - x )              */

float a, b;          /* Концы исходного отрезка          */
float e;             /* Допустимая погрешность           */
float l, p, s;       /* Концы и середина текущего отрезка */

Ввод   a, b, e;
if (F(a) * F(b) > 0)        /* Корень может отсутствовать */
   Вывод сообщения об ошибке;
else
{  /* Уточнение корня делением пополам          */
   l = a;  p = b;   s = (l + p) / 2;
   while (p – l  >  2 * e)      /* Погрешность >e      */
   {   /* Разбиение отрезка пополам                    */
      if (F(l)*F(s) <= 0)       /* Корень в левой половине */
         p = s;                 /* Выбор левой половины    */
      else
         l = s;                 /* Выбор правой половины   */
      s=(l+p)/2;
   }
   Вывод   корня  s;
}
```

4. Ручное тестирование алгоритма

Разработанный алгоритм желательно проверить методом верификации или ручного тестирования. Рассмотрим ручное тестирование.

Данному алгоритму не требуются значения функции F(x), достаточно знать их знаки. Возьмем, например, в качестве теста функцию, корень которой равен 1.56.

Тест:

F(x) < 0 при x < 1.56 a = 1, b = 1.8, e = 0.1
F(x) = 0 при x = 1.56
F(x) > 0 при x > 1.56 Ожидаемый корень: от 1.46 до 1.66

Трассировочная таблица:

l	= 1	1.4	
p	= 1.8		1.6
s	= 1.4	1.6	1.5
p-l>2*e	= да	да	нет
F(l)*F(s)<=0	= нет	да	

Результат: 1.5 (правильный)

Тест подобран так, чтобы проверить обе ветви блока РАЗБИЕНИЕ: один раз выбиралась правая половина отрезка и один раз - левая половина.

5. Программирование

По алгоритму рис. 3.9 (или 3.8) написана программа 3.1 на языке С. Она мало отличается от алгоритма на псевдокоде.

Изучение языка программирования удобно начинать с разбора примера программы, который дает общее представление о языке, а затем уже продолжить знакомство с деталями языка (сверху вниз!).

Примечания к программе 3.1

1. *Оператор препроцессора* - команда

#include <stdio.h>

пишется с новой строки и размещается в начале программы, использующей стандартные функции ввода или вывода данных: scanf(), printf() и др.

Аналогичным образом команда

#include <math.h>

необходима для программ, использующих математические функции: cos(), sin(), log() и др. (см. раздел 2.8.8).

Программа 3.1

```
/* Решение уравнения F(x)=0 на [a; b] с погрешностью e                    */
/*   методом деления пополам    (где F(x) = cos x  - x )                  */

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

/*        Функция F(x) = cos x - x        (левая часть уравнения)          */
float F (float x)
{  return cos(x)-x;
}
int main (void)
{   float a, b;            /* Концы исходного отрезка                     */
    float e;               /*  Допустимая погрешность                     */
    float l, p, s;         /*  Концы и середина текущего отрезка          */
    printf("\nВведите  границы исходного отрезка и погрешность\n");
    scanf("%f %f %f", &a, &b, &e);
    if (F(a) * F(b) > 0)                    /* Корень может отсутствовать  */
       printf  ("\nОшибка: на концах отрезка функция одного знака");
    else                /* Уточнение корня делением пополам               */
     {  l = a;   p = b;   s = (l + p) / 2;
       while (p - l > 2 * e)
       {   /* Разбиение отрезка пополам                                   */
          if (F(l)*F(s) <= 0)      /* в левой части меняется знак
         */
              p = s;               /* Выбор левой половины                */
          else
              l = s;               /* Выбор правой половины               */
          s = (l + p) / 2;
       }
       printf ("\nКорень = %f", s);
       return 0;
    }
}
```

Результаты работы программы 3.1 (на экране):

```
Введите границы исходного отрезка и погрешность
0  1  1E-6
Корень = 0.739085
```

2. Разработанная программа может решать не только заданное уравнение cos(x) = x, но и другие уравнения вида F(x) = 0.

Строки

```
float F (float x)
{  return cos(x)-x;
}
```

представляют собой определение функции: F(x) = cos(x) - x.

3. Строки вида

```
int main (void)
{
    . . .                    /*  Тело функции main */
}
```

составляют определение главной функции программы, решающей нужную задачу (main - главный).

4. В начале функции размещены определения (описания) типа использованных в ней переменных, например, строка

```
float a, b;
```

определяет, что a и b являются переменными вещественного типа.

5. В теле функции main записан алгоритм программы с помощью присваиваний, условных операторов и цикла с предусловием (см. разделы 2.6 и 3.2). Символы <= обозначают операцию "меньше или равно". Операция >= обозначает "больше или равно", != - "не равно".

6. Оператор вида printf ("текст");

выводит на экран заданный в нем текст с помощью стандартной функции printf(). Символы \n обозначают переход на новую строку.

Содержащиеся в выводимом тексте форматы, например %f, заменяются при выводе значениями, указанными через запятую после текста. Формат %f предназначен для вывода вещественного значения.

Например, оператор

```
printf ("\nКорень = %f", s);
```

выводит на экран с новой строки текст "Корень = ", а затем значение переменной s.

7. Стандартная функция scanf() служит для ввода данных, например, оператор

```
scanf("%f %f %f", &a, &b, &e);
```

вводит с клавиатуры три вещественных числа по форматам %f и присваивает их переменным a, b и e. Для ввода целого числа служит формат %d.

Для ввода данных в режиме диалога предварительно выводится приглашение "Введите ...".

Перед отладкой на ЭВМ программу или отдельные ее части, как и алгоритм, полезно проверить ручным тестированием, хотя бы на простых тестах, особенно, если она значительно детальнее алгоритма.

Упражнения и задачи

3.1. Записать на языке C/C++ алгоритмы MAXA (рис. 3.4 г) и MAXB (рис. 3.5 в).

3.2. Составить программу вычисления среднего арифметического значения последовательности из n действительных чисел, перед которой задано n:

n X_1 X_2 **...** X_n.

3.3. Составить программу (фрагмент программы) вычисления $y = x^n$, где x - вещественное число, n - неотрицательное целое число. Желательно использовать минимальное число умножений.

3.4. Последовательность чисел Фибоначчи определяется так: $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$. Составить программу (фрагмент программы) вычисления F_n для заданного n.

3.5. Составить программу вычисления НОД (x, y) - наибольшего общего делителя заданных натуральных чисел x и y.

3.6. Квадратное уравнение задано коэффициентами A, B, C. Составить программу решения квадратного уравнения вида $A x^2 + B x + C = 0$. Вывести одно из следующих сообщений о корнях уравнения: "КОРНИ ДЕЙСТВИТЕЛЬНЫЕ", "КОРНИ ДЕЙСТВИТЕЛЬНЫЕ РАВНЫЕ", "КОРНИ КОМПЛЕКСНЫЕ" и значения корней.

3.7. Составить фрагмент программы для подсчета количества единичных битов в двоичной записи заданного числа X типа **unsigned long**. Для этого выполнять операцию X & (X-1) до тех пор, пока результат не станет равным нулю. Количество повторений этого цикла будет равно искомому числу единиц, поскольку данная операция обнуляет правую единицу числа X.

3.8. Составить фрагменты программы решения следующих задач.

а) *День недели.* Определить день недели (по новому стилю) заданной даты D, M, G, где D – день (0<D<32), M – месяц (0 < M < 13), G – год (0<G<3000). Пример. Для даты 1 1 1 и 1 1 2001 день недели – понедельник.

Указание. Год високосный (содержит 366 дней вместо 365), если он делится на 400, либо делится на 4, но не делится на 100.

б) *Период дроби.* Получить последовательность цифр, образующую период десятичной дроби M / N для данных натуральных чисел M и N (M, N < 10000). Если дробь конечная, то ее период состоит из одной цифры 0. Например, для M=1, N=7 период 142857, для M=7, N=2 период 0.

3.9. Составить тесты для программ решения следующих задач (учтите возможные ошибки в данных).

а) Даны три целых числа, интерпретируемые как длины сторон треугольника. Программа сообщает вид треугольника: разносторонний, равнобедренный или равносторонний.

б) Даны три целых числа, интерпретируемые как длины сторон треугольника. Программа сообщает, можно ли построить такой треугольник.

в) Даны три целых числа, интерпретируемые как длины сторон треугольника. Программа сообщает, является ли этот треугольник остроугольным, прямоугольным или тупоугольным.

г) Даны коэффициенты A, B, C. Программа находит корни квадратного уравнения $A \cdot x^2 + B \cdot x + C = 0$ (задача 3.6).