Современное программирование с нуля!

В. Потопахин

Информатика-21

ДМК Пресс
издательство

Издание второе

Потопахин В.

Современное программирование с нуля!

ДМК Пресс
издательство
Москва, 2016

УДК 32.973.26-018.2
ББК 004.438
П64

Потопахин В.
П64 Современное программирование с нуля! – М.: ДМК Пресс, 2016. – 240 с.: ил.

ISBN 978-5-97060-405-2

Эта книга для тех, кто хочет получить хорошие навыки программирования с использованием языка Компонентный Паскаль -современной версии языка Паскаль. Изложение сопровождается большим количеством примеров, способствующих успешному усвоению материала людьми с различным уровнем подготовки – необходимо только желание и терпение.

Материал курса представлен в виде последовательности прикладных задач, нацеленных на формирование у обучаемого особой программисткой логики и дающих возможность изучить и отработать на практике все существенные особенности языка Компонентный Паскаль.

УДК 32.973.26-018.2
ББК 004.438

Потопахин Виталий Валерьевич

Современное программирование с нуля!

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Корректор *Синяева Г. И.*
Верстка *Старцевой Е. М.*
Дизайн обложки *Харевская И. В.*

Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 36. Тираж 100 экз.

Web-сайт издательства: www.dmkpress.com

ISBN 978-5-97060-405-2

Содержание

Предисловие

Книга, которую вы начинаете читать предназначена для начинающих изучать программирование, или имеющих небольшой программисткий опыт. В общем это книга для неискушенных, но желающих научится многому. Конечно, для профессионального познания любой области, одной книги всегда мало, но если у вас хватит терпения и упорства в проработке практического материала, а наша книга почти целиком посвящена практике, то можно быть уверенным, что ваш программисткий уровень станет достаточным для дальнейшего уже профессионального роста.

Книга состоит из трех глав и одного приложения. Первая глава «Неформальное введение», фактически самодостаточный логически завершенный самоучитель. Тщательное изучение неформального введения даст небольшой, но уверенный навык решения прикладных задач, и знание языка Компонентный Паскаль. Все неформальное введение от первой до последней страницы это решение задач. Каждая задача разбирается достаточно подробно, задачи используются и для рассказа о языке. Языковые конструкции вводятся по мере необходимости, тогда когда это нужно для решения очередной задачи. Это делает изучение языка хотя и несколько бессистемным (но в книге есть еще и систематическое введение), но прозрачным и понятным. Уровень сложности решаемых задач постепенное растет, но по настоящему сложных задач в неформальном введении все же нет, поэтому для его усвоения достаточно упорства и желания.

Вторая глава – систематическое введение в язык КП (Компонентный Паскаль). Здесь уже нет практики. Вся глава посвящена теории, а точнее изложению языка. Если в неформальном введении язык излагался «между делом», а главным было решение задач, то здесь главным становится язык, поэтому вторая глава отличается более строгой логикой и более трудна для понимания, но если неформальное введение вами пройдено успешно, то и вторая глава не должна создать серьезных затруднений. Еще одно важное отличие неформального введения от систематического. В первой главе используется не весь язык КП. Вторая глава излагает язык в полном объеме.

Третья глава – практикум, целиком посвящена задачам, но если в первой главе задачи использовались для объяснения, то в третьей главе задачи предлагаются для самостоятельного решения. Уровень сложности уже существенно выше. Но от вас не требуется решения с пустого места. Каждая предлагаемая в практикуме проблема снабжена пояснениями разного уровня. Где-то это описание алгоритма, где-то изложение идеи, иногда просто подсказка, иногда довольно детальная помощь. Конечно, несмотря на помощь, над каждой задачей придется основательно потрудится, но если первые две главы усвоены успешно, то и третья вполне преодолима.

Кроме трех основных глав есть еще небольшое приложение, посвященное теории графов. Оно невелико по размеру и не предназначено для хорошего изучения теории. Это приложение появилось лишь в силу того, что такая математическая конструкция как графы довольно часто используется в задачах практикума. Поэтому было сочтено полезным дать хотя бы небольшой справочный материал.

Глава 1

Неформальное введение

Кратко о главном

Программирование – наука являющаяся предметом данной книги стоит на трех слонах и одной черепахе. Слоны это: постановка задачи, алгоритм, программа. Каждое из перечисленных понятий имеет сложную историю образования, различные понимания, но для начала можно отвлечься от строгой науки и запомнить что:

- постановка задачи – это описание задачи в строгой математической терминологии;
- алгоритм – это описание действий, выполняя которые некий ИСПОЛНИТЕЛЬ ***обязательно*** получит требуемый результат. Обратите внимание на выделенное слово, именно так, не может получить, а обязательно получит;
- программа – это запись алгоритма на строгом, однозначно понимаемом языке.

А черепаха, на которой стоят наши слоны – это исполнитель, способный выполнить алгоритм. Кстати совершенно не обязательно компьютер. В самом общем случае необходимо говорить о устройстве способном выполнять определенный, жестко заданный набор команд, но наша цель – программирование компьютера, поэтому далее исполнитель это всегда компьютер.

Следовательно, научится программировать, это значит научится:

- формулировать задачу в строгих математических терминах;
- находить решение в виде последовательности действий понятных компьютеру;
- записывать эту последовательность на языке программирования.

Зачем нужен язык программирования

Язык, есть способ записи мысли. Это утверждение верно, как для естественного языка, так и для любого другого, в том числе и для языка программирования. Поэтому, главная проблема любого начинающего изучать науку программирования, это алгоритмический способ мышления, некоторые специальные методы и приемы рассуждений. Мыслить алгоритмически мы все более или менее умеем, практически любой человек в состоянии понять запись алгоритма на естественном языке, если предмет алгоритма ему известен, иначе говоря, если человек является исполнителем с достаточным для данного алгоритма набором команд.

Поэтому, по крайней мере на первых порах, проблема кажущаяся второстепенной выходит на первый план. Эта проблема названа в заголовке выше. Зачем нужен специальный язык и как им пользоваться? Попробуем сейчас решить небольшую конкретную задачу и убьем двух зайцев: ответим на поставленный вопрос и получим первую информацию о языке Компонентный Паскаль, который в дальнейшем станет основой нашего движения вглубь науки о программировании (или искусства).

Задача 1. Дано множество чисел. Найти сумму положительных.

Пока мы не знаем никакого языка кроме естественного, поэтому попробуем записать необходимую последовательность действий (алгоритм) на русском языке и посмотрим чем это будет хорошо или наоборот плохо.

Вариант 1:

Для всех чисел из данного множества:
если число положительное, то прибавляем его к сумме

Наверное, это описание будет понятно каждому кто немного знаком с математикой, хотя бы в пределах арифметики. Но к сожалению, все не так просто. В «алгоритме» (пока в кавычках, так как это описание еще очень далеко от алгоритма) сказано, что некоторую операцию необходимо выполнить для всех чисел из заданного множества, но не сказано, каким образом выбирать числа из этого множества. Следовательно, предполагается, что исполнитель алгоритма умеет это делать с произвольным множеством. То есть, он в состоянии перебрать произвольное множество и ни разу не ошибиться, не взять одно и то же число дважды и ни одно число не пропустить. Пожалуй, исполнитель с такой способностью должен обладать довольно высоким уровнем интеллекта, а следовательно быть довольно сложным и дорогим устройством. Это не может быть приемлемо. Решаемая задача проста и должна решаться простым устройством. А это означает, что придется процедуру перебора множества чисел как-то описать. И естественно как-то доопределить само понятие множества.

Пусть, все элементы множества пронумерованы и пусть известно, сколько во множестве элементов, например – N. Тогда проблема решается легко. Исполнитель начинает перебор с элемента, имеющего нулевой номер, а для получения следующего элемента увеличивает текущий номер на 1.

Вариант 2:

Для Номера изменяющегося от нуля до числа N-1 с шагом 1
выполнять действие
    Если очередное число больше нуля то прибавлять его к сумме положительных.

Это уже значительно лучше, но согласитесь, выглядит очень громоздко. А сейчас пойдем по пути упрощения записи. Рассмотрим первую фразу:

Для Номера изменяющегося от нуля до числа N-1 с шагом 1
выполнять действие

Здесь описан процесс изменения некоторой переменной величины, которую мы назвали номером. Существенно в этой записи только то что:

- переменная имеет имя, и это не обязательно слово Номер;
- исходное значение переменной равно нулю;
- конечное значение переменной равно N-1;

- переменная изменяется с шагом 1;
- на каждом шаге изменения переменной выполняется некоторое действие.

Попробуем переписать фразу, так чтобы эти существенные пункты не изменили своего смысла, но фраза стала короче. Следующий вариант:

Для k=1 до **k=N-1** с шагом 1 делать

Выделенный фрагмент содержит еще одну возможность для упрощения. Записав k=0 мы уже дали исполнителю информацию о том, что в дальнейшем речь пойдет о переменной по имени k, поэтому k=N-1 это пожалуй лишний повтор, и окончательная запись окажется очень короткой:

Для k=1 до N-1 с шагом 1 делать

Последнее, запись выполнена на русском языке. Конечно, подобную запись можно выполнить и на немецком и на хинди и на китайском и любом другом языке, но так уж получилось, что в качестве основы языков программирования взят английский. Поэтому перепишем запись следующим образом:

FOR k:=0 TO N-1 BY 1 DO

и мы получим запись так называемой конструкции цикла на языке компонентный Паскаль. Если для вас этого не сложно, то постарайтесь сразу отметить, что такая форма цикла называется циклом с параметром или циклом с шагом. Далее, будем пользоваться вторым его названием.

Примечания:

- в записи k:=0 двоеточие обязательно;
- шаг 1 считается наиболее часто встречающимся. Поэтому если шаг равен 1, то в КП (Компонентный Паскаль) запись BY 1 можно опустить.

Займемся второй фразой алгоритма

если очередное число больше нуля, то прибавлять его к сумме положительных

Очередное число это величина. Все величины должны иметь имена. Так как все перебираемые числа принадлежат одному и тому же множеству, то логично их всех назвать одним именем, а различать по номеру. То есть a[1] это элемент множества «a» с номером 1. заметьте не первый элемент, а элемент с номером 1. Это существенная разница. a[k] – это соответственно элемент множества «a» с номером k. Такое множество пронумерованных элементов имеющих одно имя называется массивом.

Еще одна используемая величина это «сумма положительных», дадим ему имя sum и перепишем фразу так:

Если a[k]>0 то sum:=sum+a[k]

И перейдя на английский, получим еще одну команду языка КП

```
IF a[k]>0 THEN sum:=sum+a[k];
END;
```

Ключевое слово END не предусмотрено алгоритмом, оно является требованием языка. Правила языка требуют завершать сложную конструкцию таким ключевым словом. А условная команда является сложной конструкцией.

А теперь полная запись

Листинг 1

```
sum:=0;
FOR k:=0 TO N-1 DO
    IF a[k]>0 THEN sum:=sum+a[k];
    END;
END;
```

Здесь две сложных конструкции, поэтому два ключевых слова END. Одно из них закрывает условную команду, второе завершает цикл.

Важное замечание. В окончательной записи добавилась команда sum:=0 которой не было в исходной записи алгоритма. Для понимания необходимости этой команды посмотрим внимательнее на запись sum:=sum+a[k]. Что здесь происходит? Команда берет уже посчитанное значение величины sum (оно справа от знака :=) увеличивает это значение на величину a[k] и полученный результат присваивает снова величине sum, то есть результат становится новым sum. Таким образом, на втором шаге sum образуется от sum полученного на первом шаге. На третьем шаге sum образуется от sum полученного на втором шаге и далее все ясно. А вот чему равно самое первое sum, от которого мы должны получить новое sum на первом шаге цикла? В команде цикла об этом ничего не говориться. А раз так, то первое значение может оказаться равным чему угодно, его значение вообще говоря это какой-то числовой мусор который к моменту работы нашего фрагмента оказался в ячейках памяти хранящих величину sum.

Поэтому, программист должен перед началом процесса вычисления величины позаботится о ее исходном значении. И команда sum:=0 именно такую работу и выполняет, а называется это инициализацией, то есть определением первого, исходного значения.

Сравните полученную запись с тем, что было изначально и вы согласитесь, что язык программирования сохраняя смысл записи, позволяет ее очень существенно укоротить. Это конечно не законченная программа. До полноценных программ еще довольно далеко. Сейчас мы обсуждаем только общие вопросы. Поэтому запишем еще один короткий алгоритм и ответим на главный вопрос, для чего нужен язык программирования.

Задача 2. Найти сумму квадратов натуральных чисел от 1 до N.

Не будем тратить времени на длинные рассуждения, запишем сразу алгоритм на КП.

Листинг 2

```
sum:=0;
FOR k:=1 TO N DO
    sum:=sum+k*k;
END;
```

А теперь главный вопрос

Наверное, вы уже согласны, что запись на строгом языке сокращает текст, делает его чтение общедоступным для большого количества людей, то есть может быть общепринятым стандартом общения для специалистов занимающихся разработкой алгоритмов. Это важно, но не это главное. Самое важное в языке программирования то, что он является связующим звеном между естественным языком и языком который в действительности понятен компьютеру. Это дает возможность писать специальные программы – трансляторы способные переводить программы, написанные на алгоритмических языках в тексты уже малопонятные для человека, но исполняемые компьютером.

Для компьютера не существует таких понятий, как множество, массив, переменная. Он оперирует регистрами, адресами ячеек памяти и т.д. и т.п. То есть чем то очень далеким даже для того, кто неплохо владеет математическим аппаратом. Поэтому до появления языков – посредников программирование было уделом немногих, ибо требовало слишком больших усилий даже для написания несложных программ.

Вспомним еще раз, что программирование это постановка задачи, алгоритмизация и кодирование (запись на языке). Все три «слона» одинаково нужны для решения любой задачи, поэтому будем просто решать задачи и одновременно учиться и алгоритмизации и кодированию, а постановка задачи пока означает запись условия задачи строгими математическими терминами. Для тех кто желает заниматься программированием глубоко еще будет возможность убедиться, что постановка задачи достаточно сложный и трудоемкий процесс.

А сейчас главная проблема это расширение языкового аппарата. В общем-то весь язык программирования сводится к набору понятий для перечисления которых хватит пальцев на одной руке. Это:

- величина (переменная или константа);
- команда присваивания;
- конструкция цикла;
- условная конструкция;
- процедура.

Но пусть вас не расслабляет столь малый набор. Каждое из этих понятий будучи развито, до необходимого функционального уровня становится довольно емким и сложным для хорошего понимания, такого понимания, какого необходимо добиться если ваша цель серьезное и систематическое освоение программирования.

С четырьмя из пяти понятий мы уже встречались. Присваивание это действие обозначаемое знаком :=, его результатом будет вычисление выражения справа от знака и присвоение полученного результата величине чье имя находится слева от знака присваивания. Цикл позволяет многократно выполнить последовательность действий записанную только один раз. Условная конструкция позволяет выполнять ту или иную последовательность команд в зависимости от результата проверки условия. С процедурой мы пока не встречались, поэтому заметим лишь, что процедура это фрагмент программы который будучи записан один раз, может выполняться в разных точках программы. Про величины уже известно, что у них есть имя и значение, еще они имеют тип – описание позволяющее определить размер памяти для их хранения.

Задача 3. Арифметическая прогрессия задана тремя величинами.

- N – количество элементов прогрессии;
- a1 – значение первого члена прогрессии;
- d – разность прогрессии.

Вычислить сумму ее членов.

Конечно, для прогрессии существует формула суммы, но такие формулы есть не для любого числового ряда, а задачи счета каких-либо рядов встречаются достаточно часто. Поэтому на примере арифметической прогрессии посмотрим, что можно сделать, если математика не дает конкретной формулы.

Итак, что необходимо сделать:

- N – раз выполнить операции:
  - Расчета очередного члена прогрессии.
  - Прибавления его к уже известной сумме.
  - Соответствующий фрагмент на КП (с грубой ошибкой).

Листинг 3

```
sum:=0;
FOR k:=1 TO N DO
    a1:=a1+d;
    sum:=sum+a1;
END;
```

Тело цикла в нашем фрагменте состоит из двух команд присваивания, первая из которых a1:=a1+d находит значение очередного члена арифметической прогрессии, и вторая sum:=sum+a1 увеличивает значение суммы на величину только

что посчитанного члена. Логика вполне понятная, но есть в ней один изъян. Самый первый член прогрессии из процесса суммирования выпадает, так как уже на первом шаге расчетов к первому прибавляется величина d и он превращается во второй. Исправить ситуацию можно так:

Листинг 4

```
sum:=a1;
FOR k:=1 TO N DO
    a1:=a1+d;
    sum:=sum+a1;
END;
```

В этом варианте первый член будет учтен в момент инициализации величины суммы, из чего следует, что идея инициализации не сводится к обнулению, хотя конечно присваивание инициализируемой величине нуля встречается наиболее часто.

Условные циклы

Уже рассмотренный нами цикл с шагом, не единственная форма цикла и даже не самая сильная. Легко придумать задачу для которой цикл с шагом не даст решения. Ясно, что цикл с шагом хорош только тогда, когда программист точно знает сколько раз необходимо выполнить тело цикла. А это вполне может оказаться и не известным. Для примера вот такая задача:

Задача 4. Арифметическая прогрессия задана начальным членом a1 и разностью d. Необходимо найти номер члена N такого, что сумма прогрессии включая N-ый превысит некое заданное число W.

Это именно та ситуация в которой известно что делать:

- вычислять очередной член прогрессии;
- находить очередную сумму.

И не известно сколько раз это делать. Следовательно, пришло время расширить набор языковых конструкций. Вернемся на время к записи на русском языке (такая запись кстати называется псевдокодом). Общая конструкция решения такова:

```
Пока Сумма<= Предельного значения делать
        Вычислить очередной член прогрессии
        Вычислить очередное значение суммы
```

Теперь, то же самое на КП

Листинг 5

```
sum:=a1;
WHILE sum<=W DO
    a1:=a1+d;
```

```
    sum:=sum+a1;
END;
```

Данная конструкция называется циклом с условием продолжения. Это означает, что тело цикла (команды записанные между заголовком и ключевым словом END) выполняется до тех пор пока истинно условие записанное после ключевого слова WHILE (Пока). Отметьте себе, что условие проверяется на каждом шаге цикла, причем сначала проверяется условие и лишь затем выполняются команды тела цикла. Это например, означает, что тело цикла может быть не выполнено ни разу, если в момент входа в цикл условие окажется ложным.

Данный фрагмент разъясняет работу новой конструкции, но не решает поставленную задачу. Задача же будет решена в том случае, если по завершению работы цикла какая-либо величина окажется равна номеру очередного члена прогрессии. Ниже уточненный вариант программы:

Листинг 6

```
sum:=a1;
k:=1;
WHILE sum<=W DO
    a1:=a1+d;
    sum:=sum+a1;
    k:=k+1;
END;
```

Величина k увеличивается на 1 на каждом проходе тела цикла и фактически равна номеру суммируемого члена прогрессии.

Цикл WHILE наиболее универсальная форма цикла, позволяющая смоделировать любой процесс, чего нельзя сказать о FOR (цикле с шагом). Но за эту универсальность надо платить тщательным построением условия продолжения цикла. Ошибка в построении условия может привести к так называемому зависанию, то есть бесконечному выполнению цикла. И вот тому простой пример:

Листинг 7

```
k:=1;
WHILE k<5 DO
    sum:=sum+k;
END;
```

В данном фрагменте некая величина k складывается с суммой на каждом шаге цикла, но цикл никогда не завершится, так как начальное значение величины k известно, но в цикле оно никак не изменяется и следовательно всегда будет меньше 5. Правильный фрагмент может выглядеть например так:

Листинг 8

```
k:=1;
WHILE k<5 DO
```

```
    sum:=sum+k;
    k:=k+1;
END;
```

Как изменяется k конечно определяется задачей, и необязательно оно изменяется с шагом 1. Но внесенное исправление по крайней мере решает проблему зависания. Цикл выполнит несколько шагов и при k=5 прекратит свою деятельность.

Цикл WHILE это цикл с условием продолжения. И в КП есть так называемый цикл с условием завершения. То есть конструкция, в которой сначала выполняется тело цикла и лишь затем проверяется условие. На псевдокоде такая конструкция будет выглядеть так:

```
Повторять
    Вычислить очередной член прогрессии
    Вычислить очередное значение суммы
Пока Сумма > Предельного значения (не станет больше)
```

А на КП это же запишется так:

Листинг 9

```
sum:=a1;
k:=1;
REPEAT
    a1:=a1+d;
    sum:=sum+a1;
    k:=k+1;
UNTIL sum>W;
```

Здесь просто переписаны уже известные вычисления в новой форме. Давайте проанализируем, как это работает и нет ли проблем. Работает цикл так: На каждом шаге выполняется тело цикла и лишь затем проверяется условие. Следовательно, тело цикла будет выполнено хотя бы один раз. Цикл завершает свою работу при истинном условии. Отметьте существенное различие от цикла с условием продолжения. WHILE выполняет свою работу пока условие истинно, а REPEAT UNTIL до тех пор пока условие не станет истинным. Поэтому и появилось различие в записи условия. Еще одно отличие формы записи в том, что цикл с условием завершения не нуждается в ключевом слове END. Его тело, это все команды находящиеся между ключевыми словами REPEAT и UNTIL.

Есть в записи цикла REPEAT и небольшая содержательная проблема. Заметим, что и как в случае условия продолжения сумма инициализируется первым членом прогрессии, плюс к тому цикл REPEAT гарантированно посчитает еще один член прогрессии, то есть второй. Следовательно, если уже первый член прогрессии окажется больше чем W программа ошибется и завершит работу при k=2, при правильном ответе k=1. Таким образом форма цикла REPEAT существенно меняет логику и правильный вариант программы будет таков:

Листинг 10

```
sum:=0;
k:=0;
REPEAT
    a1:=a1+d;
    sum:=sum+a1;
    k:=k+1;
UNTIL sum>W;
```

Циклы с условием завершения и условием продолжения взаимозаменяемы и оба они годны для замены цикла с шагом. Продемонстрируем эту взаимозаменяемость еще одним примером.

Задача 5. Вычислить факториал числа N.
Форма цикла с шагом:

Листинг 11

```
fact:=1;
FOR k:=2 TO N DO
    fact:=fact*k;
END;
```

Форма цикла с условием продолжения:

Листинг 12

```
fact:=1;
k:=2;
WHILE k<=N DO
    fact:=fact*k;
    k:=k+1;
END;
```

Форма цикла с условием завершения:

Листинг 13

```
fact:=1;
k:=2;
REPEAT
    fact:=fact*k;
    k:=k+1;
UNTIL k>N;
```

Итак, что уже известно.
К данному моменту мы довольно детально рассмотрели три вида циклов. В каждом программном фрагменте использованы команды присваивания, в первой за-

даче использована условная конструкция, но информации о ней пока конечно не достаточно. В каждом фрагменте использовалось понятие переменной, но и оно пока никак не раскрыто. На текущий момент переменная в нашем представлении это целое число. Поэтому следующая учебная задача – это условная конструкция, после чего уже будет совершенно необходимо расширить представление о типах переменных. Но пока закрепим полученную информацию небольшим самоконтролем.

Задачи для самоконтроля:

1. Напишите три варианта (для каждой из трех форм цикла) программного фрагмента суммирования N последовательных натуральных чисел: 1+2+3+…+N
2. Дано два целых числа a и b. Найти значение выражения a^b, форма цикла на ваше усмотрение.
3. Определить номер (в натуральном ряду) четного числа, такого, что сумма всех предыдущих четных включая данное больше заданного W. Будем считать, что 2 имеет номер 1, 4 номер 2 и т.д.
4. Найти сумму квадратов натуральных чисел от 1 не превышающую заданное число W. Задачу решить в двух вариантах: циклом с условием продолжения и циклом с условием завершения.
5. Не пользуясь формулой суммы найти сумму членов геометрической прогрессии заданной начальным членом, количеством членов прогрессии и ее знаменателем. Используйте для решения цикл с шагом.
6. Найти произведение двух чисел A и B не пользуясь операцией умножения. Выбор формы цикла на ваше усмотрение.
7. Вычислить N членов ряда Фиббоначи. Ряд Фиббоначи это ряд чисел определяемый следующими условиями: $a_1=1$; $a_2=1$; $a_i=a_{i-1}+a_{i-2}$. Выбор формы цикла на ваше усмотрение.
8. Найти остаток и частное от деления числа A на меньшее число B. Операциями нахождения остатка и деления пользоваться запрещается. Форма цикла на ваше усмотрение.
9. Найти сумму первых N – нечетных чисел. Выбор цикла на ваше усмотрение
10. Арифметическая прогрессия задана начальным членом и разностью. Геометрическая прогрессия задана начальным членом и знаменателем. Выяснить номер k при котором член геометрической прогрессии станет впервые больше члена арифметической прогрессии. Все величины – целые, положительные числа.

Общая структура программы на КП

Все написанные ранее примеры обладают одним существенным недостатком, они не являются полноценными программами, которые можно запустить и получить требуемый результат. К настоящей главе, у вас уже должно выработаться неплохое

представление о том, что есть такое небольшая программа на КП и должен возникнуть вопрос, а как довести разобранные задачи до полноценной, результативной программы. Рассмотрим проблему на следующем уже решенном примере – расчета факториала:

Листинг 14

```
fact:=1;
k:=2;
WHILE k<=N DO
    fact:=fact*k;
    k:=k+1;
END;
```

Итак, что очень важное здесь отсутствует. Во-первых, по завершению работы фрагмента величина fact содержит значение факториала, но мы его не увидим, так как нет операции вывода на экран посчитанного значения, во-вторых, для работы фрагмента необходимо как-то задать исходное значение величины N, которая можно сказать является аргументом для расчетного процесса, но это тоже не сделано. Дополним фрагмент необходимыми командами:

Листинг 15

```
In.Open;
In.Int(N);
fact:=1;
k:=2;
WHILE k<=N DO
    fact:=fact*k;
    k:=k+1;
END;
StdLog.Int(fact);
```

Команда In.Int(N) читается так: взять из входного потока одно целое значение и присвоить его переменной N. Команда StdLog.Int(fact) читается так: передать в выходной поток целое значение переменной fact. Команда In.Open открывает входной поток данных, действие без которого команда In.Int не будет иметь смысла. Точное определение понятий входного и выходного потока нам пока не нужно, достаточно знать, что входной поток позволяет вводить данные с клавиатуры, а отсылка данных в выходной поток, позволяет визуально увидеть величину.

Далее, для того, чтобы программу на КП можно было запустить на выполнение она должна иметь имя. Это вполне естественное требование, нельзя обратиться к тому, что не имеет имени. Завершенный программный фрагмент называется процедурой и выглядит следующим образом:

Листинг 16

```
PROCEDURE Calculation;
BEGIN
```

```
In.Open;
In.Int(N);
fact:=1;
k:=2;
WHILE k<=N DO
    fact:=fact*k;
    k:=k+1;
END;
StdLog.Int(fact);
END Calculation;
```

Слово PROCEDURE означает, что ниже записан логически завершенный фрагмент программы, который на КП называется процедурой. Далее, после ключевого слова записывается имя процедуры и наконец между ключевыми словами BEGIN и END записывается текст процедуры. Эту процедуру уже можно попытаться исполнить, но к сожалению безуспешно. В тексте не хватает еще несколько важных вещей.

Процедура действительно является логической единицей, но не вполне самодостаточной. Процедуры КП объединяются в модули. Это необходимо, даже в том случае, если процедура в модуле будет только лишь одна. Дополним наш текст:

Листинг 17

```
MODULE Example;
PROCEDURE Calculation;
BEGIN
In.Open;
In.Int(N);
fact:=1;
k:=2;
WHILE k<=N DO
    fact:=fact*k;
    k:=k+1;
END;
StdLog.Int(fact);
END Calculation;
END Example.
```

Ключевое слово MODULE означает начало описания тела модуля, которое состоит из процедур и различной другой вспомогательной информации. После MODULE записывается имя модуля. Обратите внимание, на повтор имен модуля и процедуры после соответствующих END. Это обязательно. Кроме того, END завершающий модуль записывается с точкой, после имени модуля, а после END завершающего процедуру записывается точка с запятой, после имени процедуры.

В любом языке программирования и КП в том не исключение есть одно важное правило: любое имя, используемое в программе необходимо описать, то есть должно быть известно, что это такое и как с ним работать. К примеру слова WHILE,

DO, END, BEGIN, PROCEDURE, MODULE и некоторые другие являются ключевыми словами КП и они известны компилятору изначально. Но ввести в память компилятора все возможные команды невозможно по двум причинам: во-первых, их слишком много, во-вторых, никто не знает, что может еще понадобиться программистам. Поэтому языки программирования создают расширяемыми. Часть команд объявляют стандартом языка и эти команды компилятору заранее известны, а часть команд находится в так называемых библиотеках (модулях), то есть дополнительных файлах содержащих имена команд и их исполняемый код. В нашем фрагменте такими, библиотечными командами являются команды ввода/вывода. In и StdLog имена модулей – библиотек. Int это собственно команда указывающая на выполняемое действие (ввод и вывод величины определенного типа, в нашем случае целой величины). Но для использования библиотечных команд компилятору необходимо сообщить, что та или иная библиотека (в дальнейшем будем использовать только термин модуль) будет использоваться. Перепишем модуль с необходимыми дополнениями:

Листинг 18

```
MODULE Example;
IMPORT In, StdLog;
PROCEDURE Calculation;
BEGIN
In.Open;
In.Int(N);
fact:=1;
k:=2;
WHILE k<=N DO
    fact:=fact*k;
    k:=k+1;
END;
StdLog.Int(fact);
END Calculation;
END Example.
```

Последний шаг. Мы сообщили компилятору, что команды ввода/вывода находятся в модулях In и StdLog , ключевые слова ему и так известны, сейчас осталось определить переменные: N, fact, k. Точнее, определить их тип. Необходимо это вот для чего. Компилятор до запуска программы распределяет оперативную память компьютера. Для чего это нужно вопрос достаточно обширный и детально мы его рассматривать не будем, пока ограничимся следующей версией – если этого не сделать, то в процессе работы программы может случиться конфликт между различными структурами данных претендующих на одну и ту же память. Причем компилятор помочь программисту в разрешении этого конфликта не сможет, так как в процессе работы программы компилятора уже нет.

Определение типа выполняется в определенном блоке, который можно создать в каждой отдельной процедуре, а можно и в модуле. Запишем окончательно работоспособный модуль:

Листинг 19

```
MODULE Example;
IMPORT In, StdLog;
PROCEDURE Calculation*;
VAR
    N,k,fact:INTEGER;
BEGIN
In.Open;
In.Int(N);
fact:=1;
k:=2;
WHILE k<=N DO
    fact:=fact*k;
    k:=k+1;
END;
StdLog.Int(fact);
END Calculation;
END Example.
```

Этот вариант уже можно запустить на выполнение и получить результат, но сначала три небольших, но очень важных замечания:

Замечание о регистре символов. Для КП маленькие и большие буквы, это разные буквы. Например, переменные fact, Fact, FACT это три различных переменных. Ключевые слова в КП обязательно пишутся заглавными буквами. Поэтому, запись ключевого слова while или While будет воспринята как ошибочная.

Замечание о символе «».* Заметьте, что в последней версии нашего модуля после имени процедуры появился символ звездочка. Это означает, что к данной процедуре можно получить доступ из среды BlackBox, то есть попросить среду выполнить данную процедуру и эту процедуру можно вызвать из других модулей.

Блок определения переменных. В нашем примере переменные величины объявлены в теле процедуры Calculation. Это означает, что переменные известны только процедуре Calculation. Если в данном модуле будут описаны другие процедуры, то для них эти переменные окажутся недоступными. Переменные можно описать перед всеми процедурами после имени модуля и объявления требующихся модулей. Вот так:

Листинг 20

```
MODULE Example;
IMPORT In, StdLog;
VAR
    N,k,fact:INTEGER;
PROCEDURE Calculation*;
BEGIN
In.Open;
In.Int(N);
fact:=1;
```

```
k:=2;
WHILE k<=N DO
    fact:=fact*k;
    k:=k+1;
END;
StdLog.Int(fact);
END Calculation;
END Example.
```

В данном варианте объявленные переменные уже смогут использоваться всеми процедурами модуля. А хорошо это или плохо, надо это или нет определяется только логикой задачи и замыслом программиста.

Задачи для самоконтроля:

Доведите решения 10 задач из первого задания до завершенных программ.

Условный оператор

Мы уже немного касались работы условного оператора. Сейчас займемся им более детально. Начнем с примера.

Задача 6. Дано три целых числа: a, b, c. Выяснить могут ли они быть сторонами треугольника.

Геометрия утверждает, что в любом треугольнике сумма двух любых его сторон больше третьей. Это означает, что необходимо и достаточно проверить три неравенства:

a + b > c и a + c >b и b + c >a.

Если мы проверим первое неравенство и оно окажется ложным, то следующее неравенство уже можно и не проверять, но если оно окажется истинным, то ситуация останется неопределенной и потребуется проверка второго условия, если же и второе окажется истинным, то потребуется проверка третьего. Это можно записать так:

```
Если a + b > c то
    Если a + c >b то
        Если b + c >a то "Это стороны треугольника"
```

А сейчас то же самое на КП

Листинг 21

```
IF a + b > c THEN
    IF a + c >b THEN
        IF b + c >a THEN StdLog.String('Это стороны треугольника');
        END;
    END;
END;
```

Примечание: StdLog.String команда вывода в выходной поток строки

Такая конструкция называется вложенным условием. Каждое последующее условие проверяется только в том случае, если истинно условие верхнего уровня.

Что здесь плохо. Если три наших числа удовлетворяют всем трем условиям, то мы получим содержательное сообщение, а вот если хотя бы одно из неравенств окажется ложным, то никакого сообщения программа не выдаст. Видимо в программе при проверке условия должно быть описано две последовательности действий, одна из которых выполняется если условие истинно и второе должно выполняться, если условие ложно. Для этих целей в условном операторе КП есть ключевое слово ELSE. С дополнением программный фрагмент приобретет следующий вид:

Листинг 22

```
IF a + b > c THEN
    IF a + c >b THEN
        IF b + c >a THEN StdLog.String('Это стороны треугольника');
        ELSE StdLog.String('Это не стороны треугольника');
        END;
    ELSE StdLog.String('Это не стороны треугольника');
    END;
ELSE StdLog.String('Это не стороны треугольника');
END;
```

Новый вариант выполняет свою работу корректно, но несколько громоздко. Заметим, что для положительного вывода необходима истинность всех трех условий. Ложность хотя бы одного из них приводит к отрицательному выводу. Следовательно, возможно сформировать одно сложное условие, которое должно быть истинным только в случае истинности всех трех элементарных и ложным если ложно хотя бы одно из них. Запишем следующий вариант сразу на КП

Листинг 23

```
IF (a + b > c) & (a + c > b) & (b + c > a)
THEN StdLog.String('Это стороны треугольника');
ELSE StdLog.String('Это не стороны треугольника');
END;
```

Новый текст существенно короче. Операция & называется логическим умножением (И). Если два или более условий связываются логическим И, то сложное условие истинно тогда и только тогда когда истинны все элементарные условия в него входящие и ложно если ложно хотя бы одно из элементарных. Еще одна полезная логическая связка это связка OR (ИЛИ). Если два или более логических выражений связываются логическим ИЛИ, то сложное условие истинно если истинно хотя бы одно из элементарных условий, ложным же оно будет в том случае если все элементарные условия окажутся ложными. И если речь зашла о логических операциях то упомянем еще логическое отрицание «~». Если логическое условие

истинно, то его отрицание ложно и наоборот если условие ложно, то его отрицание истинно. А сейчас рассмотрим более сложную задачу:

Задача 7. Дано три целых числа: a, b, c выяснить могут ли они являться сторонами треугольника и если да, то является ли этот треугольник: равносторонним, равнобедренным, разносторонним.

Ясно, что фрагмент программы определяющий являются ли три числа сторонами треугольника, необходимо дополнить какими-то командами после слова THEN. Сообщение о том, что это треугольник оставим на месте, но после сообщения необходим анализ условий на равнобедренность, равносторонность, разносторонность.

Для того, чтобы объявить треугольник равносторонним достаточно проверить два равенства a=b и b=c. Так как обязательно выполнение обоих условий, то можно воспользоваться логическим условием & (И). Запишем требуемое условие:

```
IF (a=b) & (b=c) THEN StdLog.String('Треугольник равносторонний');
END
```

А весь фрагмент запишется так:

Листинг 24

```
IF (a + b > c) & (a + c > b) & (b + c > a)
THEN
    StdLog.String('Это стороны треугольника');
    IF (a=b) & (b=c) THEN StdLog.String('Треугольник равносторонний');
    END;
ELSE StdLog.String('Это не стороны треугольника');
END;
```

Важное замечание. Обратите внимание, мы существенно изменили свойства готового фрагмента просто механически вставив новый код, в правильности которого нет сомнений, в точку в которой по смыслу это необходимо сделать. Это идеальная ситуация когда получается именно так. Если для изменения программы надо переделывать, весь ранее написанный код, то скорее всего это означает, что программа была очень неудачно спроектирована.

Еще одно небольшое замечание. Мы уже довольно часто использовали команды присваивания и знаем, что эта команда строится из двух знаков :=. Здесь же в операторе IF есть выражения вида a=b, содержащие один знак. Не путайте, это не присвоение, это сравнение. Результатом сравнения является логическое значение Истина или Ложь, но ни в коем случае не изменение значений переменных.

Вернемся к задаче. Еще необходимо проверить условия равнобедренности. Треугольник является равнобедренным, если есть пара равных сторон. Таких пар может быть три: a=b, b=c, c=a. Причем достаточно одной из них, или a=b или b=c или c=a. Запишем сказанное на КП с использованием логических связок:

```
IF (a=b) OR (b=c) OR (c=a) THEN
```

```
    StdLog.String('Треугольник равнобедренный');
END;
```

И весь фрагмент:

Листинг 25

```
IF (a + b > c) & (a + c > b) & (b + c > a)
THEN
    StdLog.String('Это стороны треугольника');
    IF (a=b) & (b=c)
    THEN StdLog.String('Треугольник равносторонний');
    END;
    IF (a=b) OR (b=c) OR (c=a)
    THEN StdLog.String('Треугольник равнобедренный');
    END;
ELSE StdLog.String('Это не стороны треугольника');
END;
```

Полученный фрагмент можно немного усовершенствовать. Заметим, что проверка на равнобедренность будет проводится независимо от результатов проверки на равносторонность, в любом случае программа отработает оба условных оператора, но ведь ясно, что если установлена равносторонность треугольника, то проверять его на равнобедренность уже не нужно. Изменим текст с учетом сказанного и запишем полноценную программу.

Листинг 26

```
MODULE Example;
IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    a,b,c:INTEGER;
BEGIN
In.Open;
In.Int(a);In.Int(b);In.Int(c);
IF (a + b > c) & (a + c > b) & (b + c > a)
THEN
    StdLog.String('Это стороны треугольника ');
    IF (a=b) & (b=c)
    THEN StdLog.String('Треугольник равносторонний');
    ELSE
        IF (a=b) OR (b=c) OR (c=a)
            THEN StdLog.String('Треугольник равнобедренный');
        ELSE StdLog.String('Треугольник разносторонний');
        END;
    END;
ELSE StdLog.String('Это не стороны треугольника');
END;
```

```
END Calc;
END Example.
```

Мы рассмотрели хороший пример сложного условия использующего и логические связки и вложения условия в условие, а сейчас несколько несложных задач на совместное использование условий и циклов. Одну из них доведем до законченной программы, в остальных обойдемся фрагментами.

Задача 8. Дано число. Выяснить, является ли оно простым.

Для тех кто подзабыл математику, напомним, что простые числа, – это числа делящиеся (нацело) только на единицу и самое себя. Примеры простых: 2, 3, 5, 7, 11, 13, 17, 19, 23 и т.д. Определение простого числа дает и метод решения. Пусть анализируемое число, это число N тогда достаточно определить числовой отрезок в котором могут находиться возможные делители и проверить их все. Если обнаружится хотя бы один, то число составное, если же ни одного делителя не найдется, то число простое. Естественно в этом гипотетическом отрезке не должно быть самого числа N и 1. Отсюда следует, что наиболее очевидный отрезок таков [2, N-1], но этот отрезок слишком велик. Предположим, что N=10, ясно, что анализировать делимость 10 на 6 нет смысла, значит можно отрезок сократить до [2, N / 2].

Но и это еще не все. Заметим еще более интересную вещь. Любой делитель имеет себе пару. Пусть например, проверяется на простоту число 100. Его делители составляют пары (2; 50), (4; 25), (5; 20), (10, 10). Первый элемент пары, начинаясь с наименьшего увеличивается, второй начиная с наибольшего уменьшается и оба они стремятся к 10, корню квадратному из 100. Следовательно, минимальный отрезок таков [2, Sqrt(N)], Sqrt – математическая функция в КП вычисляющая корень квадратный из числа.

Алгоритмически проблема решается следующим образом: некоторая переменная, назовем ее флагом, перед началом перебора делителей равна нулю флаг=0. Договоримся, что если она останется равна нулю, то это будет означать, что анализируемое число простое. А если найдется хотя бы один делитель, пусть флаг=1. По завершению цикла перебора возможных делителей посмотрим на значение флага. Если флаг равен нулю то ни одного делителя найдено не было и число N простое, а если он единица, то по крайней мере один делитель был найден и следовательно число составное. Ниже полноценное решение проблемы на КП:

Листинг 27

```
MODULE Example;
IMPORT In, StdLog, Math;
PROCEDURE Calc*;
VAR
    flag:BOOLEAN;
    N,k:INTEGER;
BEGIN
In.Open;
```

```
In.Int(N);
flag:=FALSE;
FOR k:=2 TO SHORT(ENTIER(Math.Sqrt(N))) DO
    IF N MOD 2=0 THEN flag:=TRUE;
    END;
END;
IF flag THEN
    StdLog.String('Число составное');
ELSE
    StdLog.String('Число простое');
END;
END Calc;
END Example.
```

Программа решает поставленную перед ней задачу, но возможно более эффективное решение, мы его приведем чуть позже, а сейчас разберем то новое, что здесь появилось.

Новый тип данных. Переменная flag объявлена как BOOLEAN. Это логический тип. Переменные данного типа имеют два значения TRUE (истина) и FALSE (ложь). В нашем случае двух значений для переменной flag достаточно, поэтому мы и выбрали данный тип. Кроме того, тип BOOLEAN дает возможность небольшого упрощения в операторе IF. Можно записать и так:

```
IF flag=TRUE THEN
```

Но так как flag и само по себе есть логическое выражение, то сравнение с TRUE не обязательно.

Преобразование типов. SHORT(ENTIER(Math.Sqrt(N))) До сих пор мы обходились одним целым типом – INTEGER. Но этого типа недостаточно для ряда математических операций. Например, операция деления не определена для целых чисел. Точно также не определена для целого и операция извлечения квадратного корня. Операция извлечения корня определена для вещественных чисел. Поэтому, если даже N и целая величина, то корень квадратный из нее окажется вещественным числом. Это плохо. КП запрещает использовать в заголовке цикла с шагом вещественные числа. Поэтому применяется функция ENTIER. Она преобразует вещественный корень в целое число, естественно его округляя.

Но функция ENTIER не решает всех проблем. Дело в том , что в КП есть несколько видов целых чисел отличающихся друг от друга объемом требуемой памяти. ENTIER преобразует вещественное число в целое более длинное чем INTEGER. Заголовок же цикла управляется переменной k объявленной как INTEGER. Выходит так, что параметр k не соответствует своей верхней границе по типу, и это с точки зрения компилятора ошибка. Операция SHORT устраняет данную ошибку обрезая лишние биты у числа ENTIER(Math.Sqrt(N)). Что впрочем не приводит к потере информации. Обрезаемые биты все равно не содержат значащих цифр.

Операция нахождения остатка. Операция MOD находит остаток от целочисленного деления числа (выражения) записанного справа от MOD на число (выражение)

записанного слева от MOD. Ясно, что эти числа (выражения) должны быть целого типа.

Необходимые замечания сделаны, попробуем улучшить алгоритм. Главный недостаток имеющегося решения в том, что цикл FOR добросовестно проверяет все потенциальные делители, в то время, как процесс можно завершить уже после обнаружения первого. Переход к циклу WHILE позволит сформировать сложное условие, одновременно учитывающее и достижение верхней границы числового отрезка и обнаружение первого делителя. Полностью программу записывать не будем, ограничимся фрагментом:

Листинг 28

```
k:=2;
flag:=TRUE;
n:=Math.Sqrt(N);
WHILE (k<=n) & flag DO
    IF N MOD k=0 THEN
        flag:=FALSE;
    END;
k:=k+1;
END;
IF flag THEN
    StdLog.String('Число простое');
ELSE
    StdLog.String('Число составное');
END;
```

Цикл WHILE завершает работу в двух случаях:

- переменная k вышла за границу отрезка содержащего делители. В этом случае флаг остается истинным и это означает, что делителей нет, а число N соответственно простое;
- флаг приобретает значение ложь. Это означает, что был найден делитель, и следовательно число составное. Цикл скорее всего заканчивает свою работу досрочно.

Еще несколько примечаний:

Обратите внимание, что в заголовке WHILE сравниваются переменные разного числового типа (целое и вещественное). В операции присваивания это вызвало бы сообщение об ошибке, сейчас различие типов проблем не вызовет, сравнивать величины разных числовых типов допустимо.

Заголовок возможно переписать так WHILE (k<= Math.Sqrt(N)) & flag DO такая запись позволила бы сэкономить одну переменную n и текст сократится на одну команду присваивания, но в скорости работы мы бы проиграли. Вспомним, что условие WHILE проверяется на каждом шаге цикла и выражения в условии вычисляются на каждом шаге, а следовательно на каждом шаге вычислялся бы квадратный корень, а это достаточно трудоемкая операция. В исходном варианте, корень считается только один раз. Еще две задачи:

Задача 9. Найти максимальное целое число во входном потоке.

Листинг 29

```
In.Open;
In.Int(max);
WHILE In.Done DO
    In.Int(N);
    IF (N>max) & (In.Done) THEN
        max:=N;
    END;
END;
StdLog.Int(max);
```

В решении используется полезная переменная Done модуля In. Эта переменная логического типа принимает истинное значение при успешном открытии входного потока и ложное при первой неудачной попытке получения значения из входного потока. Переменная Done позволяет получить все данные из входного потока, даже если неизвестно, сколько их там.

Задача 10. Дано три различных числа a, b, c определить большее.

Листинг 30

```
IF (a>b) & (a>c) THEN
    StdLog.String('Наибольшее число - a');
ELSE
    IF (b>a) & (b>c) THEN
        StdLog.String('Наибольшее число - b');
    ELSE
        StdLog.String('Наибольшее число - c');
    END;
END;
```

И последний штрих в изучении условного оператора – конструкция ELSIF позволяющая немного упростить построение сложного условия

Листинг 31

```
IF (a>b) & (a>c) THEN
    StdLog.String('max - a');
    ELSIF (b>a) & (b>c) THEN
        StdLog.String('max - b');
ELSE
    StdLog.String('max - c');
END;
```

Смысл, нового ключевого слова наверное понятен из примера.

Задачи для самоконтроля

1. Определить все делители числа N исключая N и 1
2. Определить, является ли число N совершенным. Совершенные числа, это числа равные сумме своих делителей. Например: 6 = 1 + 2 + 3; или 28 = 1 + 2 + 4 + 7 + 14.
3. Вычислить сумму всех положительных чисел из входного потока данных.
4. Вычислить сумму всех четных чисел из входного потока данных.
5. Дано три числа, определить, являются ли они сторонами прямоугольного треугольника.
6. Дано 4 числа a<b<c<d и число l определить его положение в этом ряду.
7. Дано число N. Определить является ли оно степенью двойки.
8. Дано три целых числа. Проверить являются ли они членами арифметической прогрессии.
9. Дано три целых числа. Проверить являются ли они членами геометрической прогрессии.
10. Система из двух линейных уравнений задана своими коэффициентами. Определить имеет ли она решение, если нет или бесконечно много, то сообщить об этом, если одно, то найти это решение.

Какие еще есть типы данных в КП

КП предоставляет программисту неплохой набор типов данных, достаточный для решения любой разумно поставленной задачи. Это: целый – INTEGER, длинный целый – LONGINT, действительный – REAL, логический – BOOLEAN. Эти типы называются основными. есть еще сложные типы или составные, но сейчас ограничимся только расширением основного набора и к уже перечисленным добавим: SHORTCHAR – литеры набора Latin-1; CHAR – литеры набора UNICODE; BYTE – маленькие целые, из названия ясно, что это числа помещаемые в один байт памяти; SHORTINT – короткое целое; SHORTREAL – короткие вещественные; SET – множества целых чисел.

Основной используемый тип в задачах неформального введения все же останется тип INTEGER. А следующий пример показывает использование вещественного типа:

Задача 11. Решить квадратное уравнение.
Решение. Напомним формулы. Квадратное уравнение это уравнение вида:

$ax^2 + bx + c = 0$

для вычисления его корней рассчитывается величина называемая дискриминантом, равная

$D = b^2 - 4ac$

Затем рассматриваются три случая:

- **D<0** уравнение корней не имеет
- **D=0** уравнение имеет только один корень x_1 =-b/(2a)
- **D>0** уравнение имеет два корня x_1 =(-b+Sqrt(D))/(2a); x_2=(-b-Sqrt(D))/(2a);

Листинг 32

```
PROCEDURE Calc*;
VAR
    a,b,c,x,D:REAL;
BEGIN
In.Open;
In.Real(a);In.Real(b);In.Real(c);
D:=b*b-4*a*c;
IF D<0 THEN
    StdLog.String('Уравнение корней не имеет');
ELSIF D=0 THEN
    x:=-b/2/a;
    StdLog.Real(x);
ELSE
    x:=(-b+Math.Sqrt(D))/2/a;
    StdLog.Real(x);
    x:=(-b-Math.Sqrt(D))/2/a;
    StdLog.Real(x);
END;
END Calc;
```

Это был пример на использование вещественных чисел. А без чисел типа LONGINT, например сложно обойтись при вычислении значений числовых функций. Уже вычисляемый выше факториал – функция настолько быстро растущая, что в типе INTEGER правильно будет посчитан только 12!.

Короткие числовые типы полезны, если стоит задача хранения большого количества числовых данных, значения которых изменяются в небольшой интервале. Это например могут быть физические параметры людей: рост, вес, возраст. Каждый из них не превышает 200, но для большого коллектива людей, например крупное предприятие такого рода данные могут потребовать значительной памяти, поэтому есть смысл выбрать более короткий тип.

Массивы

На самых первых страницах неформального изложения, речь о массивах уже шла, но совсем немного и вскользь. Однако массивы исключительно важный тип составных данных, поэтому их назначение и технику обработки необходимо освоить даже на начальном этапе изучения программирования.

Итак, массив это упорядоченное множество данных какого-либо основного типа. Объявляется массив следующим образом:

```
Имя: ARRAY Количество OF имя типа
```

Например правильными объявлениями будут:

```
a: ARRAY 100 OF INTEGER
mass: ARRAY 2 OF CHAR
flag: ARRAY 1000 OF BOOLEAN
```

Если в объявлении массива указано 1000 элементов, это означает, что нумерация элементов выполняется от 0 до 999. Указать произвольную нумерацию элементов массива нельзя. Количество это обязательно константа. Например, следующее объявление будет ошибочным:

```
VAR
   N:INTEGER;
   mass: ARRAY N OF BYTE;
```

Как уже говорилось объявления переменных, любых, в том числе и массивов, необходимы для того, чтобы дать компилятору возможность распределить память до начала работы программы. Но N переменная и ее значение будет определено в процессе работы программы, следовательно ее значение на этапе компиляции неизвестно, и следовательно неизвестно сколько памяти требуется под массив, посему компилятор данную запись не пропустит.

Но вполне допустима следующая запись:

```
CONST n=100;
VAR
   mas: ARRAY n OF INTEGER;
```

Здесь n – вроде бы буквенный идентификатор, но мы чуть выше объявили его значение (CONST – раздел объявления констант) и это для компилятора уже вполне допустимо. Данная запись для компилятора идентична следующей:

```
mas: ARRAY 100 OF INTEGER;
```

О технике работы с константами еще будет сказано в систематическом введении, здесь же заметим только, что значение константы нельзя изменять. Если вы полагаете, что длину массива можно изменить записав в теле программы присвоение n:=n+Число, то компилятор расценит это как ошибку. А сейчас несколько примеров.

Задача 12. Дан массив найти наибольшее значение.

Решение:

Идея такова: предположим, что максимальный элемент это первый:

```
max:=mas[1];
```

Затем организуем цикл перебора от 2 до последнего элемента и на каждом шагу цикла, если очередной элемент массива больше уже найденного максимального, то пусть максимальному присваивается значение очередного элемента. Запишем программу полностью:

Листинг 33

```
MODULE Example;
IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    mas:ARRAY 100 OF INTEGER;
    k,N,max:INTEGER;
BEGIN
In.Open;
N:=-1;
WHILE In.Done DO
    N:=N+1;
    In.Int(mas[N]);
END;
N:=N-1;
k:=1;
max:=mas[0];
WHILE k<=N DO
    IF mas[k]>max THEN
        max:=mas[k];
    END;
    k:=k+1;
END;
StdLog.Int(max);
END Calc;
END Example.
```

Примечания. Оператор N:=N-1 необходим для учета последней неудачной операции ввода. Перед тем, как цикл ввода WHILE завершит свою работу, выполнится попытка чтения из уже пустого потока, что создаст состояние ошибки, переменная Done получит значение FALSE и цикл завершит свою работу, но тело цикла уже будет выполнено и N вырастет на 1, эту 1 и надо вернуть назад.

В нашем примере в объявлении массива зарезервирована память на 100 элементов, это не означает, что вы обязаны вводить их именно 100. Можно меньше, больше нельзя, объявленной памяти не хватит. Недостаток такого объявления в том, что необходимо резервировать память с запасом, и если часть памяти окажется не нужна, ее нельзя вернуть в свободную область, более того, если массив больше не понадобится, память отведенную под него все равно уже нельзя использовать. Недостаток впрочем не слишком серьезный, вы просто должны тщательно проектировать программу и определять именно те массивы и прочие структуры данных, которые реально нужны и выделять под них столько памяти сколько в действительности понадобится. На более сложные случаи в КП есть динамические структуры данных, с которыми можно обращаться более свободно. Сейчас, еще несколько примеров, для закрепления понятия массива. Договоримся, для экономии места, опускать ввод и вывод данных, так как это достаточно стандартная операция.

Задача 13. Дан массив, N – ненулевых элементов. Посчитать количество положительных и отрицательных чисел.

Листинг 34

```
sum1:=0; sum2:=0;
k:=0;
WHILE k<N DO
    IF mas[k]>0 THEN
        sum1:=sum1+1;
    ELSE
        sum2:=sum2+1;
    END
    k:=k+1;
END;
```

Примечание. Обратите внимание, в предыдущей задаче N имело смысл верхней границы индекса элементов массива, сейчас и далее, N будет иметь смысл количества элементов. Поэтому, с учетом начала нумерации элементов массива с нуля в заголовке цикла условие записано как k<N, а не k<=N;

Задача 14. Найти сумму четных элементов массива

Листинг 35

```
sum:=0; k:=0;
WHILE k<N DO
    IF mas[k] MOD 2=0 THEN
        sum:=sum+mas[k];
    END;
    k:=k+1;
END;
```

Задача 15. Дан символьный массив. Подсчитать количество символов «b».

Решение: В этой задаче мы впервые встречаемся с массивом не числового типа, поэтому ниже полное решение.

Листинг 36

```
MODULE Example;
    IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    mas:ARRAY 100 OF CHAR;
    N,k,sum:INTEGER;
BEGIN
In.Open;
N:=-1;
WHILE In.Done DO
```

```
    N:=N+1;
    In.Char(mas[N]);
END;
k:=0;sum:=0;
WHILE k<N DO
    IF mas[k]='b' THEN
        sum:=sum+1;
    END;
    k:=k+1;
END;
StdLog.Int(sum);
END Calc;
END Example.
```

Задача 16. В символьном массиве состоящем из четного количества элементов, поменять местами четные и нечетные символы.

Пример:

- исходный массив: a, d, s, h, g, r, p, 1;
- массив результат: d, a, h, s, r, g, 1, p.

Листинг 37

```
k:=0;
WHILE k<N DO
    c:=mas[k];
    mas[k]:=mas[k+1];
    mas[k+1]:=c;
    k:=k+2;
END;
```

Задача 17. Дан массив целых чисел. Распределить его положительные и отрицательные элементы по двум разным массивам.

Решение: Задача несколько сложнее предыдущих, поэтому обсудим алгоритм. Естественно, необходимо пройти весь массив записав, для этого ту или иную форму цикла. Команды тела цикла должны проверять положителен очередной элемент массива или отрицателен и в зависимости от результата проверки заносить очередной элемент либо в массив положительных, либо в массив отрицательных.

Заметим также, что первое положительное, должно попасть в первый элемент массива положительных, второе соответственно, во второй, независимо от того, где эти первый и второй элемент были найдены. Если первый положительный, в исходном массиве будет сотым, то в массиве положительных, он все равно должен быть первым. Аналогично и для массива отрицательных. Программу запишем полностью:

Листинг 38

```
MODULE Example;
    IMPORT In, StdLog;
PROCEDURE Calc*;
```

```
VAR
    mas,masO,masP:ARRAY 100 OF INTEGER;
    k,kp,ko,N:INTEGER;
BEGIN
In.Open;
N:=-1;
WHILE In.Done DO
    N:=N+1;
    In.Int(mas[N]);
END;
k:=0;ko:=0;kp:=0;
WHILE k<N DO
    IF mas[k]>0 THEN
        masP[kp]:=mas[k];
        kp:=kp+1;
    ELSE
        masO[ko]:=mas[k];
        ko:=ko+1;
    END;
    k:=k+1;
END;
StdLog.Ln;
FOR k:=0 TO ko-1 DO
    StdLog.Int(masO[k]);
END;
StdLog.Ln;
FOR k:=0 TO kp-1 DO
    StdLog.Int(masP[k]);
END;
END Calc;
END Example.
```

Примечания:

Величины ko и kp отсчитывают количество уже найденных отрицательных и соответственно положительных чисел и одновременно играют роль номера очередного элемента соответствующих массивов masO (массив отрицательных) и masP (массив положительных).

Команда StdLog.Ln не выполняет никакой содержательной работы, она просто переводит печать результатов в журнале на следующую строку.

Задачи для самоконтроля

1. Найти сумму элементов массива имеющих четное значение индекса. Будьте внимательны, эта задача похожа на задачу из решенных выше, но там учитывались элементы немного иной природы.
2. В символьном массиве найти все символы «а» и заменить их на «1».
3. Найти в массиве два наибольших значения. Пример: для массива: 1, 1, 2, 2, 3, 1; ответом будет 2 и 3.

4. Вычислить N – членов последовательности Фиббоначи с использованием массивов.
5. Перевернуть массив, состоящий из N чисел, так чтобы первый элемент встал на место последнего, второй на место предпоследнего и т.д. Пример: для массива: 1, 2, 3, 4, 5; ответом будет 5, 4, 3, 2, 1. Четность/Нечетность количества элементов массива значения не имеет.
6. Распечатать в разных строках журнала участки символьного массива состоящие из одинаковых символов. Пример, для массива: a, a, v, a, v, s, s, s, ответом будет:
   - a a;
   - v;
   - a;
   - v;
   - s s s.
7. Переместить элементы числового массива по кругу. Пример, для массива: 1, 2, 3, 4, 5, ответом будет: 5, 1, 2, 3, 4;
8. Дан числовой массив, состоящий из ненулевых элементов, и число L. Вставить ноль в позицию L массива. Пример, дан массив: 1, 2, 3, 4, 5; L=4. Тогда результатом работы программы будет 1, 2, 3, 0, 4, 5
9. Дан числовой массив, состоящий из ненулевых элементов, и число L. Удалить из массива число, стоящее в позиции L. Пример, дан массив: 1, 2, 3, 4, 5; L=4. Тогда результатом работы программы будет 1, 2, 3, 5
10. Дано два символьных массива длины N. Выяснить, совпадают ли они с точностью до символа.

Вложенные циклы

Любой язык, в том числе и КП, допускают использование вложенных конструкций, выше мы уже работали с вложенными условиями, поэтому совершенно новой ситуация не является, но вложенные циклы иногда существенно усложняют логику программы, поэтому тема и выделена специально. Следовательно, важная дополнительная учебная задача главы, – анализ более сложных алгоритмов. Как обычно, рассмотрим несколько примеров, но для начала немного общих рассуждений.

Конструкция из двух вложенных циклов может выглядеть например так:

Листинг 39

```
FOR k:=1 TO 10 DO
    FOR j:=1 TO 10 DO
    END;
END;
```

Фрагмент состоит из двух вложенных циклов с пустым телом. Цикл по параметру k называется внешним, цикл по параметру j внутренним. Внутренний вложен во внешний. Это означает, что внутренний цикл выполняется для каждого

шага внешнего цикла. В данном фрагменте в заголовке внешнего цикла указано 10 шагов, столько же для внутреннего, это означает, что пустое тело внутреннего цикла будет выполнено 100 раз. Ниже аналог такой же структуры для цикла с условием продолжения и цикла с условием завершения:

Цикл с условием продолжения:

Листинг 40

```
k:=1;
WHILE k<=10 DO
    j:=1;
    WHILE j<=10 DO
        j:=j+1;
    END;
    k:=k+1;
END;
```

Цикл с условием завершения:

Листинг 41

```
k:=1;
REPEAT
    j:=1;
    REPEAT
        j:=j+1;
    UNTIL j>10;
    k:=k+1;
UNTIL k>10;
```

Обратите внимание, поведение внешнего и внутреннего циклов описывается различными переменными. С точки зрения правил построения сложных конструкций, это не является обязательным, но тем не менее, вложенные циклы, поведение которых описывается специальной переменной должны этой самой переменной отличаться. Простой пример:

Листинг 42

```
k:=1;
WHILE k<=10 DO
    k:=1;
    WHILE k<=10 DO
        k:=k+1;
    END;
    k:=k+1;
END;
```

На первом шаге внешнего цикла внутренний отработает 10 раз, после чего переменная k получит значение 10, и внутренний цикл завершит свою работу, но внешний также завершается при k=10, следовательно и внешний цикл на первом

же шаге закончит свое функционирование, что делает оформление цикла бессмысленным.

Еще один вариант ошибки (с использованием FOR):

Листинг 43

```
FOR k:=1 TO 10 DO
    FOR k:=1 TO 5 DO
    END;
END;
```

Эта конструкция зависнет. Внутренний цикл не даст величине k достичь значения 10.

Мы рассмотрели, только два варианта ошибки. К сожалению, возможностей ошибиться в построении сложных циклических конструкций достаточно много, две упомянутые наиболее грубые и простые, но о них очень часто спотыкаются начинающие программисты. А сейчас несколько содержательных примеров.

Задача 18. Выяснить, есть ли в числовом массиве различные числа, и если да, то сколько их.

Решение:

Для каждого элемента массива, необходимо принять решение о его уникальности. Для этого можно просмотреть весь массив и выяснить, есть ли хотя бы один такой же. Но, можно ограничится только элементами правее данного или элементами левее данного. Подумайте самостоятельно, почему нет необходимости проверять весь массив.

Выберем для определенности просмотр всех, которые левее. Следовательно, нам потребуется внешний цикл для прохода всего массива и внутренний для прохода элементов левее данного (выяснение уникальности данного). Дело внутреннего цикла запомнить, случилось ли хотя бы одно совпадение. Для запоминания этого факта используем понятия флага. Вернитесь выше, в раздел «Условная конструкция», задача 8, посмотрите что это такое. Программу запишем полностью:

Листинг 44

```
MODULE Example;
    IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    mas:ARRAY 100 OF INTEGER;
    k,j,N,sum:INTEGER;
    flag:BOOLEAN;
BEGIN
In.Open;
N:=-1;
WHILE In.Done DO
    N:=N+1;
    In.Int(mas[N]);
```

```
END;
sum:=0; k:=0;
WHILE k<N DO
    j:=0; flag:=TRUE;
    WHILE j<k DO
        IF mas[k]=mas[j] THEN
            flag:=FALSE;
        END;
        j:=j+1;
    END;
    IF flag THEN
        sum:=sum+1;
    END;
    k:=k+1;
END;
StdLog.Int(sum);
END Calc;
END Example.
```

Задача 19. Дан символьный массив и число L. Выполнить циклический сдвиг массива на L шагов.

Решение:

Заметим, что подобную задачу мы уже решали, выполняя смещение на 1 шаг. Это из задач самоконтроля. Надеемся, что вы ее решили, если же нет, то поработаем над ней вместе. Дело в том, что решение этой простой задачи дает ключ к решению сложной. Пусть есть некоторый программный текст выполняющий сдвиг массива на одну позицию. Назовем его ТЕКСТ. Тогда выполнение ТЕКСТ-а L – раз и приведет к сдвигу на L позиций. Фрагмент программы запишется так:

```
FOR k:=1 TO L DO
    ТЕКСТ
END;
```

Общая структура ясна, осталось выяснить, что кроется за словом ТЕКСТ. Для смещения на одну позицию, запомним в дополнительной переменной последний элемент массива, затем все остальные сместим от начала к концу на одну позицию, после чего запомненный элемент поместим в первую позицию.

Листинг 45

```
c:=a[N];
FOR k:=N TO 2 BY -1 DO
    a[k]:=a[k-1];
END;
a[1]:=c;
```

Это и есть искомый ТЕКСТ. Далее, продолжим решать задачи со вложенными циклами, но от массивов немного отойдем.

Задача 20. Найти все простые числа не превосходящие заданное N. Более простая задача уже решена (задача 8). Осталось ее немного модифицировать. Посмотрим, что нам не хватает.

Листинг 46

```
k:=2;
flag:=TRUE;
n:=Math.Sqrt(N);
WHILE (k<=n) & flag DO
    IF N MOD k=0 THEN
        flag:=FALSE;
    END;
    k:=k+1;
END;
IF flag THEN
    StdLog.String('Число простое');
ELSE
    StdLog.String('Число составное');
END;
```

Результат работы фрагмента – выдача сообщения о факте, простое число или составное. Для новой задачи необходимо не текстовое сообщение, а печать самого числа, если будет выяснено, что оно простое. Сообщение о том, что число составное лишнее. Внесем необходимые изменения:

Листинг 47

```
k:=2;
flag:=TRUE;
n:=Math.Sqrt(L);
WHILE (k<=n) & flag DO
    IF N MOD k=0 THEN
        flag:=FALSE;
    END;
    k:=k+1;
END;
IF flag THEN
    StdLog.Int(L);
END;
```

Кроме того, мы изменили имя переменной проверяемой на простоту с N на L. Принципиального значения данная перемена не имеет, просто имя N в условии зарезервировано под верхнюю границу проверяемых чисел. А сейчас, можно воспользоваться испытанным приемом, заключить готовый фрагмент в новый цикл. Для разнообразия воспользуемся циклом FOR.

Листинг 48

```
FOR L:=2 TO N DO
    k:=2;
    flag:=TRUE;
    n:=Math.Sqrt(L);
    WHILE (k<=n) & flag DO
        IF N MOD k=0 THEN
            flag:=FALSE;
        END;
        k:=k+1;
    END;
IF flag THEN
    StdLog.Int(L);
END;
END;
```

Обратите еще раз внимание на то, как была решена задача. Такой подход называется декомпозицией. Существо подхода заключается в простой и естественной идее. Не бросаться на задачу в лобовую атаку, а попробовать разбить ее на несколько подзадач, каждая из которых несколько проще исходной. Опыт говорит, что решить несколько простых задач легче чем одну сложную и по времени и по затраченным интеллектуальным усилиям. Конечно, рассматриваемые задачи сами по себе не сложны, но и на таких задачах, выгода видна, что же касается сложных и очень трудоемких проблем, то можно с уверенностью сказать, что без разбиения большого проекта на подзадачи серьезное программирование в принципе невозможно.

Задача 21. Вычислить сумму ряда. 1 + (1 + 2) + (1 + 2+ 3) + ...+ (1+....+N).
Решение:

Эту задачу решим дважды. Во-первых, используем идею декомпозиции. Во-вторых, попробуем провести лобовую атаку и решить проблему одним ударом. Скажем сразу, что второе решение окажется короче. Но это ни в коем случае не надо рассматривать, как камень в огород хорошей идее. Действительно иногда полезно подумать подольше и придумать красивое, яркое и короткое решение. Декомпозиция гарантирует решение, но не гарантирует, что оно будет наилучшим. Но гарантии наилучшего решения в общем-то никогда нет, а получить решение необходимо всегда, причем зачастую в ограниченное время. В общем все не так однозначно.

Вернемся к задаче. В первую очередь обратим внимание, что имеет место сумма сумм. Каждая скобочка это сумма, отличающаяся от предыдущей количеством элементов. Если написать фрагмент считающий суммы в скобочках, то заключив его в цикл отсчитывающий скобочки получим требуемое решение. Следовательно, первая подзадача это подсчет суммы натуральных чисел от 1 до L.

Листинг 49

```
sum:=0;k:=1;
WHILE k<=L DO
    sum:=sum+k;
    k:=k+1;
END;
```

Подзадача решена. Далее, в каждой очередной скобочке количество суммируемых чисел на единицу больше. Следовательно, N изменяется с шагом 1. Это все что необходимо для записи внешнего цикла:

Листинг 50

```
L:=1;
SUM:=0;
WHILE L<=N DO
    sum:=0;k:=1;
    WHILE k<=L DO
        sum:=sum+k;
        k:=k+1;
    END;
    SUM:=SUM+sum;
    L:=L+1;
END;
```

Декомпозиция дала неплохое решение, требующее двух вложенных циклов. А сейчас попробуем найти математическое решение. Исходное положение то же самое – задача заключается в суммировании сумм, назовем эти суммы промежуточными и введем обозначение S_k. Заметим, что $S_{k+1} = S_k + (k+1)$. То есть если известна некоторая промежуточная сумма, то для подсчета следующей суммы цикл НЕ НУЖЕН. Достаточно одной команды присваивания. А самая первая промежуточная сумма известна, она равна единице.

Листинг 51

```
SUM:=0; sum:=0; k:=1;
WHILE k<=N DO
    sum:=sum+k;
    SUM:=SUM+sum;
    k:=k+1;
END;
```

Команда, выделенный жирным шрифтом, заменила целый цикл.

Задача 22. Найти сумму степеней вида $a_k x^k$. a_k – элементы массива, x некоторое вещественное число. Пользоваться готовыми функциями вычисления степени не разрешается. k=0, 1, ... ,N

Решение:

Ситуация уже знакома. Требуется выполнить суммирование величин, каждую из которых еще необходимо вычислить. В нашем случае такая величина – $a_k x^k$. Впрочем, умножение степени на коэффициент проблемы не составляет, поэтому думать необходимо только о вычислении степени x^k. Ниже фрагмент выполняющий эту работу:

Листинг 52

```
st:=1;
i:=1;
REPEAT
    st:=st*x;
    i:=i+1;
UNTIL i>k;
st:=a[k]*st;
```

Фрагмент содержит немного более обещанного, по завершении цикла вычисленная степень умножается на коэффициент представленный соответствующим элементом массива. Вычисление степени завершено, второй шаг – это запись цикла суммирующего степени:

Листинг 53

```
k:=0;
sum:=0;
WHILE k<=N DO
    st:=1;
    i:=1;
    REPEAT
        st:=st*x;
        i:=i+1;
    UNTIL i>k;
    st:=a[k]*st;
    sum:=sum+st;
    k:=k+1;
END;
```

В завершении напомним, что различные типы циклов используются только для практики в их использовании.

Задача 23. Упорядочить массив, состоящий из целых чисел в порядке возрастания. Пример, для массива: 1, 4, 9, 2, 1; результат: 1, 1, 2, 4, 9

Решение:

Упорядочивание множеств различной природы – это большой и очень важный раздел программирования. Проблема заключается в том, что операция упорядочивания довольно трудоемка, поэтому для многих задач, в которых тре-

буется упорядочить много элементов или упорядочить их очень быстро, проблема сортировки (далее будем пользоваться этим термином) становится очень и очень трудной. Алгоритмов сортировки с различными свойствами и достоинствами придумано не мало. Мы рассмотрим наименее эффективный, но пожалуй самый простой способ сортировки – сортировка пузырьком. И прежде чем перейти к написанию программы рассмотрим суть самого метода.

Сортировка пузырьком

Для пузырьковой сортировки введем понятие неправильной пары. Пара рядом стоящих элементов массива a_k и a_{k+1} ,будет считаться неправильной парой если $a_k > a_{k+1}$. Если, просматривая массив, программа обнаружит такую пару, элементы пары должны обменяться своими значениями, вот так:

c:= a_k; a_k:= a_{k+1}; a_{k+1}:=c;

Естественно необходимо организовать полный проход всех возможных пар массива и выполнять обмен элементов пары только в том случае если пара неправильная.

Листинг 54

```
FOR k:=0 TO N-2 DO
    IF a[k]>a[k+1] THEN
        c:=a[k];a[k]:=a[k+1]; a[k+1]:=c;
    END;
END;
```

За один проход такого цикла некоторое количество неправильных пар обменяются своими значениями. Будет ли этого достаточно? Возьмем в качестве числового примера массив упорядоченный в порядке убывания: 5, 4, 3, 2, 1. Отработаем процесс по шагам. N здесь равно 5, поэтому шагов 4.

1. a[0]>a[1]. Выполняется обмен. Состояние массива: 4, 5, 3, 2, 1
2. a[1]>a[2] (так как сейчас 5 стоит во второй позиции). Выполняется обмен. Состояние массива: 4, 3, 5, 2, 1
3. a[2]>a[3]. Выполняется обмен. Состояние массива: 4, 3, 2, 5, 1
4. a[3]>a[4]. Выполняется обмен. Состояние массива: 4, 3, 2, 1, 5

Обратите внимание, что за один проход наибольшее число уже встало на свое место, а все остальные сместились на одну позицию к началу массива. Если положить начало массива верхом, а конец низом, то можно сказать, что легкие числа (как пузырьки) поднимаются вверх, а тяжелые тонут. Отсюда и название метода – «Пузырек». Итак, самое большое встало на место за один проход, следовательно следующее самое большое из оставшихся , встанет на свое место за следующий проход. Выходит так – за один проход как минимум одно число встает на свое законное место, но может быть и больше, это зависит от состояния массива.

Исключение составляет последний шаг, на последнем шаге сразу два числа встанут на свои места. Отсюда следует, что для массива состоящего из N чисел необходим N-1 проход. И получаем следующий программный фрагмент:

Листинг 55

```
FOR i:=1 TO N-1 DO
    FOR k:=0 TO N-2 DO
        IF a[k]>a[k+1] THEN
            c:=a[k];a[k]:=a[k+1]; a[k+1]:=c;
        END;
    END;
END;
```

Здесь нет ошибок, фрагмент вполне работоспособен, но его эффективность можно значительно повысить. Заметим, что после первого прохода, последнее число можно исключить из анализа. После второго прохода можно исключить из анализа два последних числа и наконец общее правило: на i – проходе из анализа можно исключить i элементов массива. Окончательный вариант запишем в виде полноценной программы:

Листинг 56

```
MODULE Example;
    IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    a:ARRAY 100 OF INTEGER;
    k,i,N,c:INTEGER;
BEGIN
In.Open;
N:=-1;
WHILE In.Done DO
    N:=N+1;
    In.Int(a[N]);
END;
FOR i:=1 TO N-1 DO
    FOR k:=0 TO N-i-1 DO
        IF a[k]>a[k+1] THEN
            c:=a[k];a[k]:=a[k+1]; a[k+1]:=c;
        END;
    END;
END;
FOR k:=0 TO N-1 DO
    StdLog.Int(a[k]);
END;
END Calc;
END Example.
```

Все рассмотренные выше примеры на вложенные циклы имеют один существенный недостаток. Они создают неправильное впечатление, что при организации циклического процесса, заведомо известно, сколько должно быть выполнено операций. Чтобы это впечатление не укрепилось рассмотрим два примера, в которых программист не знает сколько раз должно выполняться тело цикла.

Задача 24. Двоичное число задано целочисленным массивом длины N. Выяснить, сколько можно выполнить операций сложения этого числа с единицей до состояния переполнения. Состояние переполнения возникает, если для хранения значений необходимо более чем N элементов массива.

Решение:

Уже из условия видно, что количество операций не может быть известно, именно оно и является предметом вычислений. Решением будет цикл, в теле которого выполняется суммирование числа с единицей и выясняется возможность переполнения на последующем шаге.

Возможность переполнения выясним проверкой всех элементов массива. Если хотя бы один из них равен нулю, то следующая операция прибавления единицы возможна, если же все единицы то далее произойдет переполнение. Фрагмент проверки запишется так:

Листинг 57

```
flag=FALSE;
k:=0;
WHILE k<N DO
    IF a[k]=0 THEN
        flag:=TRUE;
    END;
    k:=k+1;
END;
```

Как прибавить единицу. Для операции сложения с единицей достаточно в числе (массиве) найти первый младший разряд равный нулю, присвоить ему единицу, а затем обнулить все разряды младшие его. Вот как это запишется:

Листинг 58

```
k:=N-1;
WHILE a[k]=1 DO
    k:=k-1;
END;
a[k]:=1;
i:=N-1;
WHILE i>k DO
    a[i]:=0;
    i:=i-1;
END;
```

Цикл ищущий первый ноль исходит из того, что этот ноль существует. Если же его нет, то цикл будет ошибочен, но до ошибки не дойдет, работа прекратится при обнаружении угрозы переполнения. И вся проблема только в том, сможем ли мы грамотно скомпоновать эти два фрагмента в единое целое.

Единым целым будет цикл выполняющий свою работу до тех пор, пока flag принимает значение истинно. Скомпонуем:

Листинг 59

```
sum:=0;
flag:=TRUE;
WHILE flag DO
    sum:=sum+1;
    k:=N-1;
    WHILE a[k]=1 DO
        k:=k-1;
    END;
    a[k]:=1;
    i:=N-1;
    WHILE i>k DO
        a[i]:=0;
        i:=i-1;
    END;
    flag=FALSE;
    k:=0;
    WHILE k<N DO
        IF a[k]=0 THEN
            flag:=TRUE;
        END;
        k:=k+1;
    END;
END;
```

Величина sum считает количество операций сложения до состояния переполнения. Полученная программа имеет один небольшой недостаток. Она предполагает, что хотя бы один раз единицу прибавить можно. Тест, состоящий из одних единиц, приведет к ошибке. А сейчас еще один пример:

Задача 25. В сортировке пузырьком, количество проходов массива жестко определяется количеством элементов, в результате в зависимости от исходного состояния массива, часть проходов может оказаться лишней. Например для массива 1, 2, 3, 5, 4 достаточно только одного прохода.

Доработаем программу так, чтобы проходы завершались по реальному завершению сортировки.

Листинг 60

```
flag:=TRUE;
WHILE flag DO
```

```
    flag:=FALSE;
    FOR k:=0 TO N-2 DO
        IF a[k]>a[k+1] THEN
            c:=a[k];a[k]:=a[k+1]; a[k+1]:=c;
            flag:=TRUE;
        END;
    END;
END;
```

Если флаг истинен, то массив неупорядочен. Если флаг ложен то массив упорядочен. Следовательно вход во внешний цикл WHILE будет выполнен в любом случае, перед началом работы предполагается, что массив все же не упорядочен. Но сразу же за заголовком флаг меняет свое значение на ложь и истиной становится только в том случае, если будет найдена хотя бы одна неправильная пара. Следовательно, в данном варианте, необходимым условием продолжения работы внешнего цикла будет обнаружение хотя бы одной неправильной пары. Следовательно, лишний проход будет только один и только в том случае, если на входе массив полностью упорядоченный по возрастанию.

Итак, мы существенно повысили сложность рассматриваемых примеров и изучили возможность построения вложенных конструкций, кстати состоящих не только лишь из одних циклов. На этом главу закончим и перейдем к самоконтролю.

Задачи для самоконтроля

1. Вычислить все совершенные числа не превосходящие заданное целое N.
2. Удалить из символьного массива все символы «а».
3. Дано два символьных массива. Выяснить, являются ли они равносоставленными, то есть существует ли такая перестановка элементов одного из них, после которой массивы станут полностью идентичными.
4. Дано два символьных массива. Построить третий массив являющийся их пересечением.
5. Дано два символьных массива, построить третий массив являющийся их разностью.
6. Реализовать сортировку выбором числового массива. Сортировка выбором, заключается в следующем: в исходном массиве обязательно существует наименьший элемент (то, что он может повторятся роли не играет). Найдем этот наименьший и поставим его в первую позицию массива. Среди оставшихся обязательно существует новый наименьший. Найдем его и поставим во вторую позицию. И так далее. На k+1-ом шаге сортировки, k – элементов оказываются отсортированными, наименьший среди оставшихся необходимо найти и поставить его в позицию k+1. Пример:
   - исходный массив: 1, 2, 8, 1, 3;
   - первый шаг: 1, 2, 8, 1, 3;
   - второй шаг: 1, 1, 2, 8, 3;
   - третий шаг: 1, 1, 2, 8, 3;

- четвертый шаг: 1, 1, 2, 3, 8.
- последний элемент не участвует в анализе, он и так наименьший для себя самого.

7. Реализовать сортировку вставками числового массива. Сортировка вставками заключается в поиске для каждого элемента массива начиная с первого, его «правильного места». Правильное место, это такое, что все элементы стоящие левее не больше данного. Предположим, что k – элементов уже отсортировано и необходимо принять решение о правильном месте для k+1 элемента. Для этого выполняется проход по массиву от k+1 позиции в сторону начала массива и ищется первый элемент уже больший k+1-го. Пусть найденный элемент имеет номер j. Тогда массив смещается на одну позицию начиная с позиции j вправо и k+1 –ый элемент ставится в позицию j. Пример:
   - исходный массив: 1, 2, 8, 1, 3;
   - первый шаг: 1, 2, 8, 1, 3;
   - второй шаг: 1, 2, 3, 8, 1;
   - третий шаг: 1, 2, 3, 8, 1;
   - четвертый шаг: 1, 1, 2, 3, 8;
   - первый элемент не участвует в анализе, так как левее его нет ни одного элемента массива.
8. Дано два массива A и B. Обнулить в массиве A все элементы имеющиеся и в массиве B.
9. Свернуть одномерный массив по следующему правилу: на каждом шаге очередной элемент нового массива получается суммированием двух симметричных (относительно середины) элементов массива, полученного на предыдущем шаге. Процесс заканчивается, когда остается одно число.
10. В массиве найти палиндром заданной длины, если такового нет, то выдать соответствующее сообщение.

Многомерные массивы

В двух предыдущих главах достаточно детально рассмотрено понятие массива, но только частного случая – одномерного. Конечно, одномерный массив – это наиболее часто встречающаяся структура данных, но размерность массивов не ограничивается единицей. Например, возможно следующее определение:

```
mas: ARRAY 100, 100 OF INTEGER;
```

Здесь объявлен, так называемый двумерный массив, то есть массив в котором каждый элемент имеет два индекса. Первый индекс изменяется от 0 до 99, второй от 0 до 99. Итого $100 \times 100 - 10000$ целых чисел. Двумерный массив можно наглядно представить в виде матрицы (таблицы).

Правда хотя такое представление полезно для наглядности, с ним надо быть осторожным, так как наглядность часто ведет к неточности понимания. Например, в таблице есть столбцы и строки и если двумерный массив отождествить

с таблицей, то возникнет соблазн один из индексов связать с понятием строки, а другой с понятием столбца, ведь для таблицы эти понятия реальны. Но для массива, что считать столбцом, а что строкой лишь некая условность. Все определяется личным желанием программиста.

Кроме того, массивы ведь могут быть и трех и четырех – мерными и даже более. И если трехмерный еще можно наглядно представить в виде параллелепипеда, то с 4-х и более размерностями, попытка дать геометрическое представление вообще не сработает.

Для первого примера, рассмотрим уже решенную ранее задачу поиска наибольшего, но решена она была для одномерного массива, мы рассмотрим ситуацию с двумерным. Ниже завершенная программа:

Листинг 61

```
MODULE Example;
    IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    a:ARRAY 10, 10 OF INTEGER;
    k,i,N,M,max:INTEGER;
BEGIN
In.Open;
In.Int(N);In.Int(M);
FOR k:=0 TO N-1 DO
    FOR i:=0 TO M-1 DO
        In.Int(a[k,i]);
    END;
END;
max:=a[0,0];
FOR k:=0 TO N-1 DO
    FOR i:=0 TO M-1 DO
        IF max<a[k,i] THEN
            max:=a[k,i];
        END;
    END;
END;
StdLog.Int(max);
END Calc;
END Example.
```

Одно отличие от одномерного массива очевидно. Полный обход двумерного массива требует два цикла (для каждого индекса). Два цикла для ввода, два цикла для обработки и если бы потребовалось массив распечатать, то и два цикла для вывода.

Второе отличие более тонкое. Если нулевой элемент до начала процесса объявляется максимальным, то его из анализа можно исключить. В случае с одномерным массивом так и сделано:

Листинг 62

```
k:=1;
max:=mas[0];
WHILE k<=N DO
    IF mas[k]>max THEN
        max:=mas[k];
    END;
    k:=k+1;
END;
```

С двумерным массивом несколько сложнее. Попробуем так:

Листинг 63

```
max:=a[0,0];
FOR k:=1 TO N-1 DO
    FOR i:=0 TO M-1 DO
        IF max<a[k,i] THEN
            max:=a[k,i];
        END;
    END;
END
```

В этом случае пропущен будет не один элемент a[0,0] а целых N при k=0. Поэтому оставим так как было, при этом элемент a[0,0] поучаствует в анализе дважды, но это беда не большая. Следующая задача:

Задача 26. Найти сумму произведений строк в двумерной, квадратной матрице.
Решение:
Задача сформулирована в терминах матриц, поэтому термин «строка» в условии вполне уместен. Мы помним, что сопоставление индекса строкам и столбцам это условность. Важно лишь то, что по некоторому индексу выполняется умножение, а по другому сложение. Запишем сказанное на КП:

Листинг 64

```
sum:=0;k:=0
WHILE k<N DO
    p:=1; i:=0;
    WHILE i<N DO
        p:=p*mas[i,k];
        i:=i+1;
    END;
    sum:=sum+p;
    k:=k+1;
END;
```

Везде, где не оговорено иное величина N будет означать длину массива.

Задача 27. Найти сумму элементов находящихся в квадратной матрице на главной диагонали и ниже ее.

Решение:

Индексы элементов находящихся на главной диагонали равны, будем называть этот единый индекс – ИНДЕКСОМ. Тогда о индексах элементов стоящих ниже, можно сказать, что один из них равен ИНДЕКСУ, а второй меньше либо равен. Запишем сказанное на КП (ИНДЕКС – переменная «k»):

Листинг 65

```
sum:=0;k:=0;
WHILE k<N DO
    i:=0;
    WHILE i<=k DO
        sum:=sum+a[i,k];
        i:=i+1;
    END;
    k:=k+1;
END;
```

Задача 28. Дана квадратная матрица с четным количеством строк. Выполнить перестановку четных и нечетных строк. Первая строка со второй, третья с четвертой и т.д.

Решение:

Задача имеет два разных решения. Первое основано на представлении матрицы двумерным массивом. Решение будет представлять собой два цикла. Внешний пустим по строкам, внутренний займется перестановкой элементов между парой строк.

Запишем решение полностью используя циклы с условием завершения (мы уже давно ими не пользовались)

Листинг 66

```
MODULE Example;
    IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    a:ARRAY 10, 10 OF INTEGER;
    k,i,N,c:INTEGER;
BEGIN
In.Open;
In.Int(N);
FOR k:=0 TO N-1 DO
    FOR i:=0 TO N-1 DO
        In.Int(a[k,i]);
    END;
END;
k:=0;
```

```
REPEAT
    i:=0;
    REPEAT
        c:=a[k,i];a[k,i]:=a[k+1,i];a[k+1,i]:=c;
        i:=i+1;
    UNTIL i>=N;
    k:=k+2;
UNTIL k>=N;
FOR k:=0 TO N-1 DO
    FOR i:=0 TO N-1 DO
        StdLog.Int(a[k,i]);
    END;
    StdLog.Ln;
END;
END Calc;
END Example.
```

Второе решение отличается способом представления матрицы. Но для того, чтобы понять суть нового решения, придется ввести понятие собственного типа данных. Типы INTEGER, REAL, BYTE и некоторые другие, называются базовыми или основными. Нам уже известен один вид сложного типа – массив. Собственный тип, – это конструкция из уже существующих типов, в том числе и уже описанных собственных. Зачем это может быть нужно, вопрос не простой, мы к нему еще будем возвращаться, а сейчас попробуем понять новую идею на примерах, как мы поступали и ранее с новыми понятиями и идеями.

Описание собственного типа начинается ключевым словом TYPE. Состоит описание из уникального (неповторяющегося) идентификатора и описания конструкции составленной из уже существующих типов. Например так:

```
TYPE
    mas=ARRAY 10 OF INTEGER;
```

Здесь идентификатор mas означает не массив целых, а тип массив целых. Это означает, что идентификатор mas можно использовать для объявления массивов, но нельзя использовать как массив. Следующее объявление:

```
VAR
    a:ARRAY 10 OF mas;
```

является объявлением массива из 10 элементов типа mas, то есть десяти массивов, для каждого из которых резервируется память под 10 целых чисел. По размеру резервируемой памяти это объявление идентично следующему:

```
a: ARRAY 10, 10 OF INTEGER;
```

но функциональность у него немного другая. В объявлении a:ARRAY 10 OF mas; определен не двумерный массив, а массив массивов. Это означает, что к отдельным подмассивам этого массива можно обращаться как к самостоятельным величинам. А сейчас вернемся к задаче перестановке строк и посмотрим, что дает новая конструкция:

Листинг 67

```
MODULE Example;
    IMPORT In, StdLog;
PROCEDURE Calc*;
TYPE
    mas=ARRAY 10 OF INTEGER;
VAR
    a:ARRAY 10 OF mas;
    k,i,N:INTEGER;
    c: mas;
BEGIN
In.Open;
In.Int(N);
FOR k:=0 TO N-1 DO
    FOR i:=0 TO N-1 DO
        In.Int(a[k,i]);
    END;
END;
k:=0;
REPEAT
    c:=a[k]; a[k]:=a[k+1];a[k+1]:=c;
    k:=k+2;
UNTIL k>=N;
FOR k:=0 TO N-1 DO
    FOR i:=0 TO N-1 DO
        StdLog.Int(a[k,i]);
    END;
StdLog.Ln;
END;
END Calc;
END Example.
```

От двух циклов обработки остался только один. Это благодаря тому, что в операторах присваивания

```
c:=a[k]; a[k]:=a[k+1];a[k+1]:=c;
```

присваивание выполняется не с отдельными числами, а с подмассивами. Отсюда мораль: грамотное построение собственного типа может существенно упростить логику программы.

Задача 29. Дана квадратная матрица. Распечатать k-ую диагональ вниз от главной.

Решение:

Выше уже было выяснено относительно главной диагонали, что индексы ее элементов равны. Очевидно, что для любой диагонали можно построить формулу для индексов. Для решения задачи именно это и необходимо: выразить зависимость

двух индексов от номера диагонали. Попробуем найти нужную закономерность на примере с матрицей размером 4 × 4.

Таблица 1.1. Таблица индексов

0, 0	0, 1	0, 2	0, 3
1, 0	1, 1	1, 2	1, 3
2, 0	2, 1	2, 2	2, 3
3, 0	3, 1	3, 2	3, 3

В ячейках таблицы проставлены значения индексов, значения элементов для дальнейших рассуждений роли не играют. Обратите внимание на первый индекс:

- главная диагональ (нулевая): 0, 1, 2, 3;
- первая диагональ: 1, 2, 3;
- вторая диагональ: 2, 3;
- третья диагональ: 3.

Отсюда ясна формула. Пусть k – номер диагонали. Тогда:

```
k<=ИНДЕКС1<=N
```

В нашем случае N = 3. k=0, 1, 2, 3. Далее посмотрим значение второго индекса:

- главная диагональ (нулевая): 0, 1, 2, 3;
- первая диагональ: 0, 1, 2;
- вторая диагональ: 0, 1;
- третья диагональ : 0.

Диапазон изменения второго индекса очевидно такой:

```
0<=ИНДЕКС2<=N-k
```

Интервалы изменения индексов в зависимости от k (номера диагонали) ясны. Но диагональ это линейный объект, хотя и находящийся в структуре двумерного объекта, поэтому следующим действием, необходимо выразить один из индексов через другой, например второй через первый. Очевидно формула такова: ИНДЕКС2 = ИНДЕКС1 – k

Листинг 68

```
i:=k;
WHILE i<=N DO
    StdLog.Int(a[i,i-k]);
    i:=i+1;
END;
```

Короткое решение при довольно длинном рассуждении, но так бывает достаточно часто. При работе с многомерными массивами очень важно точно определить взаимозависимости между индексами, это проблема требующая определенного времени. Для завершения главы рассмотрим еще одну несложную задачу, но с массивом большей размерности.

Задача 30. Дан трехмерный массив, целых положительных чисел. Представим, его как трехмерную решетку размерами NX × NY × NZ. В узлах решетки находятся целые числа. Такая решетка имеет форму параллелепипеда. Предположим, этот параллелепипед распилили на плоскости (двумерные массивы). Требуется найти номер плоскости, такой что сумма элементов расположенных на ней наибольшая.

Решение:

В условии задачи не сказано вдоль какой оси распилена решетка, но это и не имеет значения, так как ориентация решетки относительно системы координат всего лишь небольшая условность. Пусть решетка распилена вдоль оси Z. Тогда для каждой координаты z, необходимо вычислить сумму по всем x, y:

0<=x<NX; 0<=y<NY

Листинг 69

```
z:=0;
max:=0;num:=0;
WHILE z<NZ DO
    sum:=0;
    x:=0;
    WHILE x<NX DO
        y:=0;
        WHILE y<NY DO
            y:=y+1;
            sum:=sum+a[x,y,z];
        END;
        x:=x+1;
    END;
    IF sum>max THEN
        max:=sum;
        num:=z;
    END;
END;
```

Переменная num по завершении работы фрагмента будет содержать номер наибольшей плоскости, номер совпадающий с координатой z.

Задачи для самоконтроля

1. Дан двумерный массив размерности N × M. Найти сумму его элементов.
2. Выполнить транспонирование двумерной матрицы. Транспонированием называется операция преобразования матрицы, при которой столбцы меняются со строками.
3. Выяснить, является ли квадратная матрица симметричной относительно главной диагонали.
4. Выяснить, является ли квадратная числовая матрица магическим квадратом. Магический квадрат это числовая матрица, в которой для всех строк и всех столбцов сумма элементов одинакова.
5. Найти в двумерном массиве все седловые точки. Седловые точки возможны двух типов:
   a. Седловой точкой называется элемент матрицы наибольший в строке и наименьший в столбце.
   b. Седловой точкой называется элемент матрицы наименьший в строке и наибольший в столбце.
6. Выяснить, есть ли в заданной матрице размерами M × N квадрат со стороной L, состоящий из одних единиц.
7. Переписать двумерный массив размером M × N в одномерный.
8. Задача обратная 7-ой. Дан одномерный массив, длина которого есть произведение двух целых чисел M и N. Переписать данный одномерный массив в двумерный размером M × N.
9. Трехмерная кубическая решетка со стороной N заполнена нулями. Заполнить ее внутренние диагонали единицами и наглядно продемонстрировать успешность выполненной работы. Для демонстрации успешности достаточно распечатать срезы решетки вдоль любой координатной оси, представляющие собой двумерные массивы.
10. В двумерный массив записаны как положительные, так и отрицательные числа. Рассортировать их по двум различным одномерным массивам.

Процедуры

Термин «процедура» не является новым. Текст решения любой из задач, которых уже решено достаточно много, это процедура. Но решение не обязательно состоит только лишь из одной процедуры, их может быть несколько. Обычно решение разбивают на несколько процедур, если речь идет о больших задачах. Но многопроцедурное решение есть следствие более фундаментального принципа – декомпозиции. Напомним, что декомпозиция, это разбиение задачи на несколько логически независимых подзадач. А если подзадачи логически независимы, записывать их удобно в виде отдельных процедур. О теории еще поговорим, а сейчас в качестве иллюстрации получим еще одно решение уже решенной задачи – сортировка пузырьком.

Для решения, массив необходимо ввести из потока, затем выполнить его обработку и затем вывести. Таким образом, исходная задача разбивается на три. Дадим им имена и запишем в той последовательности в которой их требуется выполнять:

Input; – ввод массива из потока
Bubble; – пузырьковая сортировка
Output; – вывод.

Записав эти три строчки мы в каком-то смысле решили поставленную задачу. Можно оформить модуль:

Листинг 70

```
MODULE Example;
    IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    a:ARRAY 100 OF INTEGER;
N: INTEGER;
BEGIN
    Input;
    Bubble;
    Output;
END Calc;
END Example.
```

Но работать модуль конечно же не будет, так как три термина Input, Bubble, Output не поддержаны никаким кодом. Решим проблему, переписав код касающийся ввода, в тело процедуры Input, код сортировки в тело процедуры Bubble, код вывода в тело процедуры Output. Работающий вариант выглядит так:

Листинг 71

```
MODULE Example;
    IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    a:ARRAY 100 OF INTEGER;
    N: INTEGER;
PROCEDURE Input;
BEGIN
    In.Open;
    N:=-1;
    WHILE In.Done DO
    N:=N+1;
    In.Int(a[N]);
    END;
END Input;
```

```
PROCEDURE Bubble;
VAR
    i,k,c:INTEGER;
BEGIN
FOR i:=1 TO N-1 DO
    FOR k:=0 TO N-2 DO
        IF a[k]>a[k+1] THEN
            c:=a[k];a[k]:=a[k+1]; a[k+1]:=c;
        END;
    END;
END;
END Bubble;
PROCEDURE Output;
VAR
    k:INTEGER;
BEGIN
FOR k:=0 TO N-1 DO
    StdLog.Int(a[k]);
END;
END Output;
BEGIN
    Input;
    Bubble;
    Output;
END Calc;
END Example.
```

Невооруженным глазом видно, что текст стал даже больше. Стоило ли это делать? Конечно, для такой небольшой программы может быть и нет, но давайте немного рассудим. Текст главной процедуры сократился до трех команд. Главная процедура предельно ясна: ввод, обработка, вывод. И анализ каждой из них можно выполнять отдельно, не держа перед глазами весь текст. Это удобно даже для небольшой программы, для большой же разбивка программы на отдельные процедуры становится жизненно необходимой операцией.

Попробуем проиллюстрировать это утверждение серьезным примером. Очень большая и очень сложная программа – это программа играющая против человека в интеллектуальную игру или как еще их называют в игру с полной информацией. Пример такой игры шахматы. Игрой с полной информацией шахматы называются потому, что оба противника видят все, что находится на доске, скрыты только намерения.

Начнем разработку. Что нам необходимо реализовать? Очевидно, потребуется прорисовка шахматной позиции. Потребуется какой-то способ определения, завершена игра или нет и если завершена то в чью пользу. Необходимо дать возможность человеку сообщить программе о выборе своего хода и наконец необходим метод принятия решения компьютерной программой о своем ходе. Запишем сказанное на псевдокоде:

```
Прорисовка исходной позиции
Пока Игра не завершена делать
    Запросить ход человека
        Выполнить ход
        Если партия не завершена То
            Принять решение о ходе программы
            Выполнить ход
        Завершить программу и сообщить победителя
```

Пусть для упрощения человек начинает игру всегда. «Игра не завершена» это некая проверка, возможно достаточно сложная, но сейчас не будем разбираться, как именно работает проверка, можно принять только что:

- если результат проверки = 0 то игра не закончена;
- если результат = 1 то выиграл человек;
- если результат = 2 то выиграла программа.

Назовем процедуру Proverka. Процедура «Выполнить ход» своим результатом очевидно имеет изменение позиции, в этом смысле она сильно похожа на «Прорисовка исходной позиции» и быть может есть смысл их объединить, но возможно такое объединение слишком усложнит логику. Мы объединять их не станем. Пусть это будут различные процедуры. Кроме того, выполнение хода нуждается в информации о том, кто сделал ход и чем, в то время как исходная прорисовка ни в какой информации не нуждается.

Как сообщить процедуре «Выполнить ход» информацию о ходе? Ход это две вещи: Откуда и Куда. Следовательно, все определяется тем, какой структурой данных разработчик представит себе доску и фигуру. Пусть это будет пара величин. А именно:

- номер клетки, с которой выполняется ход;
- номер клетки, на которую выполняется ход.

Зная клетку, мы знаем и какая фигура на ней стоит и какого она цвета, поэтому данная информация достаточна. Принятые решения позволяют написать текст модуля:

Листинг 72

```
MODULE Game;
VAR
   N1, N2:INTEGER;
PROCEDURE Start;
END Start;
PROCEDURE Move(n1,n2:INTEGER);
BEGIN
END Move;
PROCEDURE Man;
BEGIN
```

```
END Man;
PROCEDURE Computer;
BEGIN
END Computer;
PROCEDURE Proverka(n:INTEGER):INTEGER;
BEGIN
RETURN 0;
END Proverka;
PROCEDURE Main*;
VAR
   Res:INTEGER;
BEGIN
   Start;
REPEAT
   Man; (*Ход за человека*)
   Move(N1, N2);
   Res:=Proverka(1);
   IF Res=0 THEN
      Computer; (*Ход за программу*)
      Move(N1, N2);
      Res:=Proverka(2);
   END;
UNTIL Res>0;
END Main;
END Game.
```

Еще почти ничего не известно, о том как будет выглядеть конкретный код реализующий например ход или ищущий ход за компьютер, но кое-что уже есть. Во-первых, есть общая структура процесса игры. Во-вторых, есть перечень подзадач подлежащих разработке и с каждой подзадачей сопоставлена конкретная процедура, каждую из которых можно далее писать отдельно от других. Поставленные подзадачи (процедуры), возможно также будут разбиты на более мелкие подзадачи, процесс разработки станет понятным и пошаговым, к нему можно будет привлечь нескольких разработчиков, при этом не объясняя каждому из них в чем заключается общая задача.

Разработка такой программы очень большой труд, достойный отдельной книги, мы же на этом этапе закончим, так как цель – демонстрация процедурного программирования достигнута.

Модуль состоящий из нескольких процедур становится достаточно сложным организмом. Оказывается что:

- процедуры возможно описывать на разных уровнях. Не просто внутри модуля, а в теле другой процедуры и таких вложений может быть сколь угодно много;
- описания переменных выполненные сразу после заголовка модуля и в теле процедур имеют различные свойства;
- значения величин процедурам можно передавать;

- процедуры способны возвращать значения в точку вызова;
- в теле процедуры возможно вызывать не только иные процедуры, но и ее же саму, это называется рекурсией.

Есть и еще интересные свойства, но даже перечисленных достаточно для хорошей работы. Приступим.

Локальные и глобальные переменные

Как всегда пример:

Листинг 73

```
MODULE Example;
VAR
    k, g:INTEGER;
PROCEDURE Proc1;
VAR
    k:REAL;
BEGIN
END Proc1;
PROCEDURE Proc2;
BEGIN
END Proc2;
END Example.
```

В модуле объявлены не две переменные, а три. Глобальные k и g, и локальная k. Локальная значит видимая только в той процедуре в которой она объявлена. Локальная переменная вещественного типа объявлена в процедуре Proc1. Процедура Proc2 ничего не знает про вещественную величину k, но обращение в Proc2 к переменной по имени k не приведет к ошибке, так как существует еще одна k объявленная для всего модуля. Поэтому обращение к переменной k в теле Proc2 будет означать обращение к целочисленной глобальной величине.

Глобальная, значит известная всем процедурам. То есть целочисленная k должна быть доступна и для Proc1 и для Proc2. Но есть в нашем примере один нюанс. В Proc1 имя k использовано для локальной переменной. Это приводит к невозможности доступа к глобальной k в теле Proc1 не взирая на ее ГЛОБАЛЬНОСТЬ.

Таким образом, переменная обнаруженная в теле процедуры будет истолкована, как локальная если она есть в списке локальных переменных, как глобальная если она есть в определениях модуля и ее при этом нет в списке локальных переменных и как ошибка, если ее нет ни в одном списке.

Ситуация несколько усложняется, если появляются процедуры вложенные в процедуры. Например так:

Листинг 74

```
MODULE Example;
VAR
```

```
    k, g:INTEGER;
PROCEDURE Proc1;
VAR
    k:REAL;
PROCEDURE Proc2;
BEGIN
END Proc2;
BEGIN
END Proc1;
END Example.
```

Положение процедуры Proc2 существенно изменилось. Сейчас Proc2 составная часть Proc1. Поэтому обращение к имени k в Proc2 сейчас означает обращение к вещественной переменной объявленной в Proc1, а не к глобальной целой k. В следующем случае:

Листинг 75

```
MODULE Example;
VAR
    k, g:INTEGER;
PROCEDURE Proc1;
VAR
    k:REAL;
PROCEDURE Proc2;
VAR
    k: BYTE;
BEGIN
END Proc2;
BEGIN
END Proc1;
END Example.
```

обращение в теле Proc2 к тому же имени k уже будет означать обращение к переменной объявленной для Proc2, то есть переменной байтового типа. А сейчас можно сформулировать и общее правило: при обращении к имени переменной в теле процедуры, ее объявление ищется в этой же процедуре, в случае неудачи в процедуре содержащей данную и т.д. В случае, если ни в одной из объемлющих процедур имени переменной обнаружить не удалось, то имя ищется в списке объявлений модуля а если и это не удалось, то возникает состояние ошибки.

Из сказанного ясно, что объявления можно делать по разному, но как определить точку в которой целесообразно объявлять конкретную переменную?

Ответ очень прост. Переменная должна быть объявлена в той процедуре, в которой она нужна. Конечно, это только общая фраза, ее еще необходимо пояснить примерами. Но сейчас еще один общий ответ, на общий вопрос – для чего и как передаются значения в процедуры и что значит передать значение.

Наиполезнейшее свойство процедуры заключается в том, что записав ее один раз, программист может пользоваться ей многократно в самых различных точках

модуля и даже в других модулях, если процедура помечена «*», то есть экспортируется. Но это совершенно не означает, что потребность в этой процедуре всегда одна и та же. Посмотрим на процедуру Sqrt (вычисление квадратного корня). Она бы полностью потеряла смысл, если бы вычисляла корень всегда от одного и того же числа. Значение от которого нужен корень определяется логикой того фрагмента, в котором выполняется вызов. Sqrt процедура библиотечная (уже кем-то написанная и сохраненная в специальном файле), но сказанное верно и в отношении процедур создаваемых программистом для своей задачи. Перейдем к примеру:

Задача 31. Вычислить сумму факториалов.
Решение:
И счет суммы и счет факториалов вещи понятные, поэтому новая организация программы будет понята легко. Запишем сразу текст на КП.

Листинг 76

```
MODULE Example;
    IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    sum: LONGINT;
    N,k:INTEGER;
PROCEDURE Factorial(n:INTEGER): LONGINT;
VAR
    fact:LONGINT;
    k:INTEGER;
BEGIN
fact:=1;
FOR k:=1 TO n DO
    fact:=fact*k;
END;
RETURN fact; (*Возврат значения в точку вызова*)
END Factorial;
BEGIN
In.Open;
In.Int(N);
sum:=0;
FOR k:=1 TO N DO
    sum:=sum+Factorial(k);
END;
StdLog.Int(sum);
END Calc;
END Example.
```

И в процедуре Calc и в процедуре Factorial есть переменная k, она играет роль параметра цикла. Эта переменная в обоих процедурах является локальной. Здесь работает неписанное правило требующее все параметры циклов задавать локально.

Иное возможно и не запрещается, но легко может привести к ошибке. Попробуйте определение величины k убрать из Factorial, результат работы программы серьезно изменится.

Величины N и sum не нужны за пределами процедуры Calc, величина fact не нужна за пределами процедуры Factorial. Поэтому данные величины определены, как локальные в своих процедурах.

Величина n объявленная в заголовке процедуры Factorial используется для передачи значения. n называется формальным параметром, значение передаваемое в момент вызова называется фактическим параметром. Передача значения в какой-то степени эквивалентно выполнению присваивания n:=k; пока именно так «в какой-то степени», есть в этом вопросе особенности которые мы рассмотрим в систематическом введении.

Еще несколько важных замечаний:

О обязательности формальных параметров. С точки зрения КП формальные параметры не обязательны. Мы могли бы формальный параметр исключить, можно исключить из описания Factorial и величину fact объявив их в объемлющей процедуре Calc. Но тогда в случае ошибочного использования этих величин программисту придется анализировать все тексты, где они могут быть использованы. В нашем же варианте величины локализованы и если с ними произошла ошибка, то искать ее можно только в Factorial. Иначе говоря, чем больше общих переменных, тем больше возможностей для совершения ошибок.

О команде возврата RETURN. Команда прерывает выполнение процедуры и возвращает вычисленное значение. RETURN может в тексте процедуры находится где угодно, их может быть несколько, но по крайней мере один должен быть. Команда RETURN чаще используется в процедурах с возвратом. Иначе они называются процедуры – функции. Такие процедуры отличаются от прочих тем, что в их заголовке указывается тип возвращаемого значения. Тип возвращаемого значения должен быть совместимым с типом выражения получающего значение, в противном случае компилятор выдаст сообщение о ошибке.

О расположении процедуры. В нашем случае процедура Factorial расположена в тексте процедуры Calc. Из этого следует, что процедурой Factorial можно пользоваться только в теле Calc или в теле процедур записанных в Calc после Factorial. В иных процедурах модуля или в иных модулях ее использовать нельзя. Если программист считает, что написанная им процедура может использоваться за пределами Calc, то модуль должен быть записан так:

Листинг 77

```
MODULE Example;
   IMPORT In, StdLog;
PROCEDURE Factorial(n:INTEGER): LONGINT;
VAR
   fact:LONGINT;
```

```
    k:INTEGER;
BEGIN
fact:=1;
FOR k:=1 TO n DO
    fact:=fact*k;
END;
RETURN fact; (*Возврат значения в точку вызова*)
END Factorial;
PROCEDURE Calc*;
VAR
    sum: LONGINT;
    N,k:INTEGER;
BEGIN
In.Open;
In.Int(N);
sum:=0;
FOR k:=1 TO N DO
    sum:=sum+Factorial(k);
END;
StdLog.Int(sum);
END Calc;
END Example.
```

Для рассчитываемой величины ничего не изменилось, но сейчас Calc и Factorial равноправны по отношению к другим процедурам модуля и Factorial возможно использовать за пределами Calc.

Процедура вполне может быть и без возвращаемых параметров. Мы уже рассмотрели пример (сортировка пузырьком) в котором процедуры были средством декомпозиции задачи, а все передаваемые величины объявлялись, как глобальные переменные.

Задача 32. Удалить все нули из целочисленного массива. Операцию удаления нуля оформить, как отдельную процедуру.

Решение:

Алгоритм может представлять собой цикл выполняющий проход по массиву, на каждом шаге которого относительно элемента принимается решение о удалении (удалять или нет) и если да, то вызывается процедура выполняющая удаление. Цель процедуры изменить состояние массива, поэтому возвращать новый массив нет смысла, нового массива просто нет, все операции выполняются на одном массиве, а передать номер позиции в которой находится удаляемый ноль возможно. Запишем фрагмент главного цикла (N – количество элементов массива):

Листинг 78

```
k:=0;
WHILE k<N DO
    IF a[k]=0 THEN
        Del(k);
```

```
        k:=k-1;
        N:=N-1;
    END;
    k:=k+1;
END;
```

Присваивание k:=k-1 необходимо. В случае удаления нуля, массив сдвигается влево на одну позицию, после чего параметр k смещается по массиву вправо, в результате параметр k относительно массива смещается не на одну позицию, а сразу на две. Если в массив записать два рядом стоящих нуля, то второй ноль останется не вычеркнутым. Присваивание N:=N-1, необходимо для учета изменения длины массива. Его можно выполнить, как в теле процедуры, так и там где он записан сейчас. К положению команды k:=k-1 программа может оказаться более чувствительной. А ее лучше сделать локальной, выше мы говорили, что параметры цикла лучше выглядят локальными.

Листинг 79

```
PROCEDURE Del(n: INTEGER);
VAR
    k: INTEGER;
BEGIN
k:=n;
WHILE k<N-1 DO
    a[k]:=a[k+1];
    k:=k+1;
END;
END Del;
```

Переменная N являющаяся длиной массива предполагается глобальной. Это разумно, так как и сам массив по логике программы глобальный (объявлен для всего модуля). Процедура не содержит ни одной команды RETURN. В них нет необходимости, процедура честно отрабатывает свой цикл и заканчивает работу, результатом будет глобальный массив. Но иногда все же полезно передать массив в качестве параметра. Рассмотрим механизм этого действия на уже решенном примере.

Задача 33. Найти сумму построчных произведений прямоугольной матрицы размером N × M. Для расчета произведения написать процедуру получающую на вход строку матрицы.

Решение:

Для решения придется двумерный массив представить, как массив массивов, только так можно передать в процедуру отдельную строку.

Листинг 80

```
MODULE Example;
IMPORT In, StdLog;
```

```
TYPE mas=ARRAY 10 OF INTEGER;
PROCEDURE Multiply(b:mas; N:INTEGER):INTEGER;
VAR
    p, k:INTEGER;
BEGIN
p:=1;
k:=0;
WHILE k<N DO
    p:=p*b[k];
    k:=k+1;
END;
RETURN p;
END Multiply;
PROCEDURE Calc*;
VAR
    a:ARRAY 10 OF mas;
    sum,N, M, k,j:INTEGER;
BEGIN
In.Open;
In.Int(N);In.Int(M);
FOR k:=0 TO N-1 DO
    FOR j:=0 TO M-1 DO
        In.Int(a[k,j]);
    END;
END;
sum:=0;
k:=0;
WHILE k<N DO
    sum:=sum+Multiply(a[k],M);
    k:=k+1;
END;
StdLog.Int(sum);
END Calc;
END Example.
```

Обсудим этот текст. На что следует обратить внимание:

- тип данных mas используется в обоих процедурах, поэтому он объявлен в заголовке модуля. Если объявление типа разместить в заголовке процедуры Calc, то для процедуры Multiply оно окажется недоступным;
- в процедуру Multiply передается два параметра. Первый является массивом. Второй играет роль фактической длины передаваемого массива. В нашем случае это не обязательно, так как все строки имеют одинаковую длину, но мы показали, что для Multiply одинаковость строк не является обязательным, что делает процедуру более универсальной.

Завершим главу более емким примером.

Задача 34. Найти целые решения следующего уравнения:

$a_n x^n + a_{n-1} x^{n-1} + \ldots\ldots + a_0 = 0$

Пусть для упрощения коэффициенты a_k – целые числа. Известно, что все целые корни такого уравнения находятся среди делителей свободного члена. Поэтому алгоритм решения должен найти все делители и каждый из них проверить, а не корень ли это. Проверка числа на свойство быть корнем заключается в вычислении значения многочлена $a_n x^n + a_{n-1} x^{n-1} + \ldots\ldots + a_0$ от аргумента «x» которое возможно и есть корень.

Заметим, что возможные целые корни находятся в числовом отрезке $[-a_0, a_0]$, за исключением нуля. Вычисления сводятся к суммированию степеней. Предположим, разработчик не имеет готовых модулей для вычисления корней и степеней. В этом случае исходная задача разбивается на следующие подзадачи:

- вычисление степени некоего заданного, целого числа;
- вычисление значения многочлена;
- поиск всех возможных делителей заданного числа, с проверкой делителя, корень или нет.

Первая подзадача не представляет собой ничего сложного. Оформим ее решение в виде процедуры:

Листинг 81

```
PROCEDURE St(x, N: INTEGER):INTEGER;
VAR
    p,k:INTEGER;
BEGIN
p:=1;
FOR k:=1 TO N DO
    p:=p*x;
END;
RETURN p;
END St;
```

Далее, поработаем над вычислением значения от многочлена. Искомая процедура должна на вход получить массив, старшую степень многочлена и значение «x» от которого собственно и будет считаться многочлен.

Листинг 82

```
PROCEDURE Summ(a: mas; N:INTEGER; x:INTEGER): INTEGER;
VAR
    sum,k:INTEGER;
BEGIN
sum:=0;
```

```
    FOR k:=0 TO N DO
        sum:=sum+a[k]*St(x,k);
    END;
    RETURN sum
    END Summ;
```

Весь так сказать инструментарий готов, можно написать главную процедуру – поиска делителей и их проверки.

Листинг 83

```
PROCEDURE Calc;
VAR
    k : INTEGER;
BEGIN
k:=1;
WHILE k<=a[0] DO
    IF a[0] MOD k=0 THEN
        IF Summ(a,N, k)=0 THEN
            StdLog.Int(k)
        END;
        IF Summ(a,N, -k)=0 THEN
            StdLog.Int(-k)
        END;
    END;
    k:=k+1;
END;
END Calc;
```

Комментировать процедуру не будем, она достаточно прозрачна. Осталось добавить код для ввода значений массива и скомпоновать все в единый модуль. Вот его текст:

Листинг 84

```
MODULE Example;
IMPORT In, StdLog;
TYPE mas=ARRAY 100 OF INTEGER;
PROCEDURE Calc*;
VAR
    k, N : INTEGER;
    a: mas;
PROCEDURE Summ(a: mas; N:INTEGER; x:INTEGER): INTEGER;
VAR
    sum,k:INTEGER;
PROCEDURE St(x, N: INTEGER):INTEGER;
VAR
    p,k:INTEGER;
BEGIN
```

```
    p:=1;
    FOR k:=1 TO N DO
        p:=p*x;
    END;
    RETURN p;
    END St;
    BEGIN
    sum:=0;
    FOR k:=0 TO N DO
        sum:=sum+a[k]*St(x,k);
    END;
    RETURN sum
    END Summ;
    BEGIN
    In. Open;
    In.Int(N);
    k:=0;
    WHILE k<=N DO
        In.Int(a[k]);
        k:=k+1;
    END;
    k:=1;
    WHILE k<=a[0] DO
        IF a[0] MOD k=0 THEN
            IF Summ(a,N, k)=0 THEN
                StdLog.Int(k)
            END;
            IF Summ(a,N, -k)=0 THEN
                StdLog.Int(-k)
            END;
        END;
        k:=k+1;
    END;
    END Calc;
    END Example.
```

Глобальные переменные полностью отсутствуют. Все величины необходимые для работы процедур либо определены в тексте процедур, либо получены ими через списки формальных параметров. Модуль состоит только из одной процедуры – Calc. Процедура Summ нужна только процедуре Calc, поэтому она определена как вложенная в Calc. Процедура St нужна только для работы Summ, поэтому она определена, как вложенная в Summ. Кстати вложенность St в Summ не накладывает на St никаких обязательств перед Calc. Попробуйте где-нибудь в теле Calc записать например команду k:=St(1,2); компилятор сообщит, что записан неопределенный идентификатор. Та же запись в теле процедуры Summ не создаст проблем. Еще один небольшой пример и можно переходить к задачам для самоконтроля.

Задача 35. Вычислить корень степени N из числа A>1. При условии, что пользоваться процедурами модуля математики запрещается.

Решение:

Не стоит воспринимать, как прихоть требование обойтись без готовых процедур. На самом деле, это очень важно уметь обходиться минимумом возможностей для решения задачи. Тот кто может много на бедном инструментарии, сможет еще больше на богатом, а вот обратное неверно. Кроме того, задачи такого типа хорошая тренировка ума, а обучение чему-либо, это в первую очередь тренировка ума, и уже потом усвоение знаний выработанных другими. Если этих двух аргументов вам недостаточно, то вот еще один. Желание воспользоваться готовыми инструментами основано на ошибочном убеждении, что все, что можно сделать, уже сделано. В действительности нерешенных проблем очень много, да и если нужная вам микропроблема решена, то откуда вы знаете, что она решена с требуемым для вас качеством?

У автора этих строк была ситуация, в которой старый и заслуженный компилятор, как оказалось считал значения обычного синуса с погрешностью недопустимой для конкретной задачи. Так что всякое бывает!

Вернемся к задаче. В этом примере мы используем еще один важный методологический прием. Если задача достаточно сложна, то для не слишком опытного разработчика бывает полезно решить более простую задачу, развивая решение которой можно постепенно выйти на исходную.

Для нашего случая, такой более простой задачей будет вычисление квадратного корня из A. Наиболее простой способ поиска квадратного корня это метод половинного деления отрезка. Его суть в следующем: заметим, что корень находится в отрезке [1, A]. Отрезок можно даже сузить, но пусть будет такой. Обозначим приближение к корню через величину b и пусть b=(1+A)/2, то есть середина.

Скорее всего это очень грубое приближение. Выясним в какую сторону ошибка, то есть приближение слишком велико или слишком мало. Для этого возведем приближение в квадрат и сравним его с A. Возможны следующие варианты:

- $b^2 > A$ – это означает, что приближение слишком велико и в действительности корень находится на отрезке [1, b];
- $b^2 <= A$ – это означает, что приближение слишком мало и в действительности корень находится на отрезке [b, A].

Далее, примем новый отрезок за исходный и продолжим вычисление до тех пор, пока длина отрезка не станет меньше заданной точности. После этого, последний раз вычислим середину отрезка, – это и будет искомый корень с заданной точностью. Ниже программа:

Листинг 85

```
MODULE Example;
IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
```

```
    A,b,min,max:REAL;
BEGIN
In.Open;
In.Real(A);
min:=1;
max:=A;
WHILE max-min>0.01 DO
    b:=(min+max)/2;
    IF b*b>A THEN
        max:=b;
    ELSE
        min:=b;
    END;
END;
StdLog.Real((min+max)/2);
END Calc;
END Example.
```

Здесь точность фиксирована, конечно в реальной программе, ее лучше вводить. Полученный текст по отношению к исходной задаче имеет только одно слабое место, а именно следующее условие:

```
IF b*b>A THEN
```

Только из-за этого условия, возможности программы ограничены квадратным корнем. Если бы здесь считалась произвольная степень, то программа могла бы считать произвольный корень. Следовательно, все что осталось, это написать процедуру вычисления произвольной степени, а мы это уже делали в предыдущей задаче (но только для целых чисел) и скомпоновать с имеющейся процедурой. Ниже полностью работающий текст:

Листинг 86

```
MODULE Example;
IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    A,b,min,max:REAL;
    N:INTEGER;
PROCEDURE St(x: REAL; n:INTEGER): REAL;
VAR
    p: REAL;
    k: INTEGER;
BEGIN
p:=1;
FOR k:=1 TO n DO
    p:=p*x;
END;
RETURN p;
```

```
END St;
BEGIN
In.Open;
In.Real(A);
In. Int(N);
min:=1;
max:=A;
WHILE max-min>0.01 DO
    b:=(min+max)/2;
    IF St(b, N)>A THEN
        max:=b;
    ELSE
        min:=b;
    END;
END;
StdLog.Real((min+max)/2);
END Calc;
END Example.
```

Задачи для самоконтроля

Некоторые из этих задач вами уже решались, но без использования дополнительных процедур.

1. Выяснить, есть ли в заданной матрице размерами M × N квадрат со стороной L, состоящий из одних единиц. В дополнительной процедуре реализовать проверку совпадения квадрата с частью матрицы в текущей точке.
2. Реализовать сортировку вставками числового массива. В отдельную процедуру выделить операцию вставки. Описание сортировки посмотрите в главе «Массивы»
3. Определить все делители числа N исключая N и 1. Самостоятельно реализовать процедуру определения остатка от деления и затем использовать ее в решении.
4. Вычислить все совершенные числа не превосходящие заданное целое N. Использовать собственную процедуру проверяющую число на совершенность.
5. Трехмерная кубическая решетка со стороной N заполнена нулями. Заполнить ее внутренние диагонали единицами и наглядно продемонстрировать успешность выполненной работы. Для демонстрации успешности достаточно распечатать срезы решетки вдоль любой координатной оси, представляющие собой двумерные массивы. Пусть собственная процедура занимается вопросами распечатки срезов.
6. Дано натуральное число. Построить его каноническое разложение. Для определения степени делителя числа написать дополнительную процедуру.

7. Дан трехмерный массив, целых положительных чисел. Представим, его как трехмерную решетку размерами NX × NY × NZ. В узлах решетки находятся целые числа. Такая решетка имеет форму параллелепипеда. Предположим, этот параллелепипед распилили на плоскости (двумерные массивы). Требуется найти номер плоскости, такой что сумма элементов расположенных на ней наибольшая. Дополнительную процедуру написать для вычисления суммы элементов лежащих на плоскости.
8. Двоичное число задано целочисленным массивом длины N. Выяснить сколько можно выполнить операций сложения этого числа с единицей до состояния переполнения. Операцию прибавления единицы оформить, как дополнительную процедуру.
9. Выяснить, есть ли в числовом массиве различные числа, и если да, то сколько их. Дополнительную процедуру написать для проверки числа на повторяемость (алгоритм см. выше)
10. Дан символьный массив и число L. Выполнить циклический сдвиг массива на L шагов. Использовать дополнительную процедуру выполняющую сдвиг на 1 шаг.

Рекурсия

При обсуждении общих свойств процедур уже было сказано, что вызывать процедуру возможно в ее собственном теле. Например, синтаксически следующая конструкция вполне законна:

Листинг 87

```
PROCEDURE St;
BEGIN
St;
END;
```

Смысла правда она не имеет, более того, если процедура будет вызвана она создаст ошибку. Действительно, первый вызов St приведет ко второму вызову, второй к третьему и т.д. Процесс мог бы быть бесконечным, спасет ситуацию (относительно конечно) только то, что под каждый вызов процедуры требуется память, память же не бесконечна, поэтому довольно быстро возникнет состояние переполнения и работа программы прервется. Ниже полный пример:

Листинг 88

```
MODULE Example;
PROCEDURE Calc*;
PROCEDURE St;
BEGIN
```

```
St;
END St;
BEGIN
St;
END Calc;
END Example.
```

Результатом запуска процедуры Calc будет сообщение – stack overflow (переполнение стека). Но конечно же не всегда это так бессмысленно. Рассмотрим содержательный пример. Рассчитаем факториал от числа N. Мы уже занимались этой задачей, теперь получим еще одно решение – рекурсивное.

Для того, чтобы получить рекурсивное решение математической задачи, необходимо представить математическое решение в рекуррентной форме. Определение факториала формулой N!=1*2*3*…*N не рекуррентное. Заметим однако, что N!=1*2*3*….*(N-1)*N. Из этой формулы можно выделить 1*2*3*….*(N-1)==(N-1)! отсюда следует что:

N! = N * (N-1)!

Здесь, значение факториала определено через сам же факториал, но от меньшего аргумента. Это и есть рекуррентное определение. Оно пошагово сводит расчет величины к ситуации, в которой значение величины известно по определению. Из формулы выше не понятно, что это за значение, поэтому ее нужно доопределить. Вот так:

1. N! = N * (N-1)!
2. 1! = 1.

Сейчас рекуррентное определение полное. Рассмотрим, как на основании этих формул, можно построить процесс расчета факториала. Рассчитаем например 5!.

1. На этом шаге нельзя вычислить факториал, но первая формула нам дает, что 5! = 5 * 4!. Если 4! будет вычислен, то будет вычислен и 5!
2. 4! = 4 * 3!. Свели вычисления к еще меньшему аргументу.
3. 3! = 3 * 2!
4. 2! = 2 * 1!
5. Единственный шаг, на котором мы реально можем вычислить факториал, а именно 1! = 1.

Далее, выполняется обратный подъем. Вернувшись на четвертый шаг вычислим 2!=2*1=2, вернувшись на третий шаг вычислим 3!=3*2=6, вернувшись на второй шаг вычислим 4!=4*6=24 и вернувшись на первый шаг решим исходную задачу 5!=5*24=120.

А написание рекурсивной программы практически сводится к записи на языке программирования рекуррентного определения. Ниже завершенный текст программы на КП.

Листинг 89

```
MODULE Example;
IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    N:INTEGER;
PROCEDURE Factorial(n: INTEGER):INTEGER;
BEGIN
IF n=1 THEN
    RETURN 1; (*Прямое вычисление*)
ELSE
    RETURN n*Factorial(n-1); (*Переход к меньшему аргументу*)
END;
END Factorial;
BEGIN
In.Open;
In.Int(N);
StdLog.Int(Factorial(N));
END Calc;
END Example.
```

Для полного понимания механизма работы программы, необходимо познакомиться с понятием стека. Рекурсивные вызовы имеют важную особенность. Предположим вызов за номером k полностью отработал, получил всю необходимую информацию от вызовов нижнего уровня и пришло время передать некую информацию вызову с номером k-1. Это означает, что должна существовать область памяти для хранения данных экземпляра процедуры с номером k-1, а точнее для каждого экземпляра вызываемой процедуры должна существовать своя область памяти, в которой хранятся ее данные.

Эти области, будем называть их далее блоками, имеют одинаковую структуру, так как все экземпляры вызываемой процедуры идентичны. Кроме того:

- каждый блок нужен только лишь на то время, пока соответствующий ему экземпляр находится в работе, после чего память, занимаемую блоком можно вернуть в свободную область;
- первый блок будет уничтожен последним, а созданный последним освободит занимаемую память первым.

Структура памяти работающая по такому принципу: первым зашел, последним вышел называется стеком. Разработчику необходимо знать, что под стек выделяется даже не вся оперативная память имеющаяся у компьютера, а только часть, поэтому рекурсивные процедуры должно разрабатывать очень аккуратно, точно рассчитывая необходимый объем данных. Еще одна проблема рекурсивных процедур это возможно большое количество вызовов, на каждый из которых

также уходит процессорное время. Поэтому прежде чем, принимать решение о поиске рекурсивного решения, следует подумать о возможности решения не рекурсивного.

Кстати в науке о программировании есть утверждение, почти теорема, что для любой задачи, есть как рекурсивное, так и не рекурсивное решение.

Важный вопрос. Зачем же нужна рекурсия, если она ест ресурсы процессора, нуждается в специально организованной памяти, которой еще может и не хватить?

Ответ лежит в плоскости логики. Зачастую рекурсивное решение проще. Следующий пример задача Дейкстры:

Задача 36. Числовая функция определена условиями:

1. F(1) = 1
2. F(2N) = F(N)
3. F(2N + 1) = F(N) + F(N + 1)

Для не рекурсивного решения придется провести кое-какой математический анализ и найти хорошую числовую закономерность. Это вполне возможно, но можно этого и не делать. Заметим просто, что определение функции уже рекурентно. О чем говорят записанные формулы: вторая формула утверждает, что при четном аргументе функция переходит в функцию от вдвое меньшего аргумента, третья формула утверждает, что при нечетном аргументе, функция выражается через сумму двух функций, аргументы которых отличаются друг от друга на единицу и в сумме равны исходному, и наконец первая формула это единственный случай, когда значение функции считается непосредственно. Решение запишем в виде процедуры:

Листинг 90

```
PROCEDURE F(n: INTEGER):INTEGER;
BEGIN
IF n=1 THEN RETURN 1;
ELSIF (n MOD 2=0) THEN RETURN F(n DIV 2);
ELSE RETURN F(n DIV 2)+F(n DIV 2 + 1)
END;
END F;
```

Попробуйте самостоятельно найти нерекурсивное решение, может быть оно получится не намного длиннее, но то, что логика его будет существенно сложнее, это факт. Отсюда можно сделать вывод о полезности рекурсии:

Рекурсивные решения выгодны, если речь идет о ветвящихся вычислительных процессах.

Мы обязательно рассмотрим еще пару примеров с ветвящимися процессами, но пока немного потренируемся с линейными, помня, что это только учебные примеры и в реальной жизни такие задачи надо решать не рекурсивно.

Задача 37. Найти N – член арифметической прогрессии с заданным a_1 и разностью d.

Решение:

Не рекуррентная формула гласит, что $a_N = a_1 + d(N - 1)$. Рекуррентная формула будет такова:

$a_N = a_{N-1} + d$; непосредственно считаемый случай здесь не нужен, первый элемент прогрессии является заданным.

Листинг 91

```
PROCEDURE F(n, a, d: INTEGER):INTEGER;
BEGIN
IF n=1 THEN RETURN a;
ELSE RETURN F(n-1, a, d)+d;
END;
END F;
```

Задача 38. Вычислить квадратный корень из числа A>1.

Решение:

Как и прежде, мы должны найти рекуррентную формулировку задачи. В качестве основы для рассуждений возьмем уже имеющийся вариант. Вот этот:

Листинг 92

```
WHILE max-min>0.01 DO
    b:=(min+max)/2;
    IF b*b>A THEN
        max:=b;
    ELSE
        min:=b;
    END;
END;
```

Цикл завершает свою работу при max-min<=0.01; Видимо в рекурсивном варианте это условие и будет условием завершения процесса вызовов. Заметим также, что потребности в возврате каких-либо значений нет. Когда условие завершения выполнится, необходимо распечатать значение очередного приближения и завершить работу. Не обязательно распечатку полученного корня выполнять в первом вызове рекурсивной процедуры. Это можно сделать и в самом глубоком, а затем свернуть все вызываемые экземпляры, так сказать в холостую.

Листинг 93

```
PROCEDURE F(min, max, A:REAL);
VAR
    b:REAL;
BEGIN
```

```
IF max-min>0.01 THEN
    b:=(max+min)/2;
    IF b*b>A THEN
        max:=b;
    ELSE
        min:=b;
    END;
    F(min,max,A);
ELSE
    StdLog.Real((max+min)/2);
END;
END F;
```

Решенный пример, показывает, что схемы организации рекурсии могут быть различными. Таким образом без возврата, можно организовать и рекурсивный процесс расчета факториала, но здесь отсутствие возврата момент принципиальный. Для нашей задачи потребуется возврат двух значений min и max, а процедура – функция на КП не может вернуть более одного значения, с использованием известной нам технологии. Впрочем, есть и другие способы организации возврата, но ими мы займемся в систематическом введении.

Задача 39. Вычислить N – число Фиббоначи.

Решение:

Здесь точно такая же проблема с передачей фактических параметров. Каждому экземпляру будущей процедуры необходимо два числа для расчета третьего. Следовательно, и возвращать необходимо два, если воспользоваться схемой рекурсии из задачи о факториалах. Но можно поступить так же как и в предыдущей задаче.

Листинг 94

```
PROCEDURE F(a1,a2, N:INTEGER);
VAR
    a3:INTEGER;
BEGIN
IF N>2 THEN
    a3:=a1+a2;
    a1:=a2;
    a2:=a3;
    F(a1,a2,N-1);
ELSE
    StdLog.Int(a2);
END;
END F;
```

Еще одна рекурсивная процедура без возврата. Но к сожалению такая схема рекурсии возможна только при условии, что вычислительному процессу достаточно

одной конечной точки. Это означает, что для большинства ветвящихся процессов схема без возврата работать не будет. Например, такая схема неприменима в задаче Дейкстры.

Задача 40. Получить все перестановки N – элементов целочисленного массива.
Решение:

Задача имеет хорошее нерекурсивное и хорошее рекурсивное решение, нас интересует второе. Простой и понятный способ построения перестановки это циклический сдвиг массива. Пусть N – количество элементов массива, тогда циклическим сдвигом можно получить N – перестановок. Пример:

1. 1, 2, 3
2. 3, 1, 2
3. 2, 3, 1

Возьмем любую перестановку, полученную сдвигом N – элементов. Выделим в ней подмассив в N-1 элемент. Из него можно получить N-1 перестановку. Соответственно для любой перестановки N-1 можно выделить подмассив N-2 и построить N-2 перестановок и т.д. Таким образом можно получить:

N*(N-1)*(N-2)*….*2=N! перестановок.

Процесс получения перестановок очевидно, будет заключаться в выполнении двух действий:

1. Определение подмассива на котором необходимо выполнить циклический сдвиг.
2. Выполнение сдвига.

Вторая операция очевидна. В отношении первого действия заметим следующее: Для подмассива состоящего из k – элементов можно выполнить k – циклических сдвигов. Отсюда следует, что для каждого подмассива необходимо завести счетчик сдвигов. В нерекурсивном варианте это может быть массив, элементы которого увеличиваются на 1 при каждом сдвиге, в рекурсивном варианте это может быть номер вызова процедуры. Ниже полное решение:

Листинг 95

```
MODULE Example;
IMPORT In, StdLog;
PROCEDURE Calc*;
VAR
    a:ARRAY 10 OF INTEGER;
    N:INTEGER;
PROCEDURE Print;
VAR
    k:INTEGER;
BEGIN
FOR k:=0 TO N-1 DO
    StdLog.Int(a[k]);
```

```
END;
StdLog.Ln;
END Print;
PROCEDURE Rec(N:INTEGER);
VAR
    i,j,c:INTEGER;
BEGIN
i:=0;
WHILE i<N DO
    c:=a[N-1];
    j:=N-1;
    WHILE j>0 DO
        a[j]:=a[j-1];
        j:=j-1;
    END;
    a[0]:=c;
    IF i<N-1 THEN
        Print;
    END;
    IF N>0 THEN
        Rec(N-1);
    END;
    i:=i+1;
END;
END Rec;
BEGIN
In.Open;
N:=-1;
WHILE In.Done DO
    N:=N+1;
    In.Int(a[N]);
END;
Print; (*Распечатка начальной перестановки*)
Rec(N);
END Calc;
END Example.
```

Задача 41. Ханойская башня. Старая классическая задача встречающаяся во всех учебниках программирования. Смысл задачи в следующем: дано три подставки, на первой из них лежит некоторое количество дисков, таких что для любой пары дисков тот, что сверху, имеет радиус меньший того, что ниже. Необходимо переложить все диски на третью подставку, не нарушая следующих правил:

- за один раз можно брать только один диск;
- диск можно класть только на диск большего радиуса или на пустую подставку.

Попробуем обнаружить рекуррентную природу задачи. Это легко сделать, посмотрев на два рисунка.

Рис. 1.1. Ханойская башня, исходное положение

Рис. 1.2. Ханойская башня, промежуточное положение

Если удастся получить ситуацию, изображенную на рис. 1.2, то исходная задача перемещения пирамиды с левой подставки на правую решится в два шага. Во-первых, диск с левой подставки перемещается на правую, а затем решается задача перемещения меньшей пирамиды со средней подставки на правую. А для того, чтобы получить такую ситуацию, надо сначала решить задачу перемещения ханойской башни без нижнего диска на среднюю подставку. Таким образом, исходная задача о перемещении башни из пяти дисков сводится к двум задачам о перемещении четырех дисков.

Рекуррентный характер задачи налицо, следовательно, поиск рекурсивного решения вполне оправдан. Тривиальный случай (когда действие выполняется непосредственно) для построения рекурсивной процедуры тоже понятен – это башня из одного диска. Входные данные для рекурсивной процедуры следующие:

- номера подставок в следующем порядке: подставка, с которой необходимо переместить текущую пирамиду, вспомогательная подставка, и подставка, на которую требуется переместить пирамиду;
- высота перемещаемой пирамиды.

Алгоритм рекурсивного процесса на псевдокоде может выглядеть так:

```
Если перемещаемая высота = 1 То
    Переместить один диск с исходной подставки на подставку – цель.
    Иначе
        Вызвать процедуру, перемещающую пирамиду с высотой на 1 меньше
        с исходной подставки на вспомогательную.
        Переместить нижний диск с исходной подставки на подставку – цель.
        Вызвать процедуру, перемещающую пирамиду с вспомогательной под-
        ставки на подставку – цель.
```

Листинг 96

```
MODULE Example;
IMPORT In, StdLog;
TYPE mas=ARRAY 10 OF INTEGER;
PROCEDURE Main*;
VAR
    a:ARRAY 4 OF mas;
    h:ARRAY 4 OF INTEGER;
    n,k:INTEGER;
PROCEDURE Print;
VAR
    k,j:INTEGER;
BEGIN
FOR k:=1 TO 3 DO
    StdLog.String('__');
    FOR j:=1 TO h[k] DO
        StdLog.Int(a[k,j])
    END;
    StdLog.String('__');
END;
StdLog.Ln;
END Print;
PROCEDURE hanoy(a1,a2,a3,m:INTEGER);
PROCEDURE perest;
BEGIN
h[a3]:=h[a3]+1;
a[a3,h[a3]]:=a[a1,h[a1]];
a[a1,h[a1]]:=0;
h[a1]:=h[a1]-1;
Print;
END perest;
BEGIN
IF m=1 THEN
    perest;
ELSE
    hanoy(a1,a3,a2,m-1);
    perest;
    hanoy(a2,a1,a3,m-1);
END;
END hanoy;
BEGIN
In.Open;
In.Int(n);
h[1]:=n;h[2]:=0;h[3]:=0;
FOR k:=1 TO n DO
    a[1,k]:=n-k+1;
END;
Print;
```

```
hanoy(1,2,3,n);
END Main;
END Example.
```

Каждая подставка представлена элементом массива a. Первый индекс массива – подставка, второй индекс проходит по дискам лежащим на подставках. Нулевые элементы массива не используются, так как нумерацию подставок и дисков более естественно начинать с единицы, а несколько лишних элементов массива проблемы не составляют.

Полученное решение – хороший пример формального подхода. Мы уже говорили о том, что для построения алгоритма весьма желательно выработать хорошее представление о процессе. Попробуйте представить процесс перемещения дисков. Для двух дисков это элементарно, для трех затруднительно, а попробуйте представить себе процесс для пяти или шести дисков. Это будет уже не так просто. Или даже попробуйте, поняв алгоритм на четырех дисках, повторить его для шести. Совершенно не очевидно, что у вас это получится. Кстати, даже в анализе задачи мы нашли существенную сложность. Рекуррентное определение говорит о том, что исходная задача сводится к двум задачам, при этом целевое использование подставок меняется. Вот эта постоянная перемена предназначения подставок сильно осложняет понимание процесса.

Но разбираться детально в механизме в общем-то и нет необходимости. Мы уже описали некую общую технологию построения рекурсивных программ. Она состоит из ряда формальных шагов: определение условия завершения, организация рекурсивного вызова, выполнение текущих операций. Если это сделано корректно, то беспокоится о правильном понимании процесса уже не стоит, правильность процесса гарантирует компилятор.

В рассмотренной задаче на каждом шагу выполняется элементарное перемещение дисков и два рекурсивных вызова, отличающихся друг от друга назначением подставок (какая из них конечная, а какая вспомогательная). Точное определение назначения подставок при каждом вызове избавляет нас от необходимости понимания всего процесса перемещений.

Задачи для самоконтроля (рекурсивность решения предполагается)

В этом разделе для самоконтроля только 5 задач и все они линейные. Рекурсивных задач с ветвящимися процессами будет очень много в третьей главе – практикуме.

1. Вычислить сумму ряда $1 - 2 + 3 - \ldots. + (-1)^{N+1}N$.
2. Вычислить сумму N – членов арифметической прогрессии при заданных a_1 и разности d.
3. Вычислить сумму N – членов ряда Фиббоначи.
4. Вычислить сумму вида $1 + (1 + 2) + (1 + 2 + 3)+\ldots(1 + 2 +\ldots.+ N)$.
5. Реализовать алгоритм Евклида.

Записи

До сего момента мы использовали только одну разновидность сложных структур данных – массив. Вспомним, для определения элементов массива годится любой основной тип, но что принципиально для массива, тип массива является типом всех его элементов. Невозможно, объявить, первый элемент целым, второй символьным и т.д. Впрочем это и не надо. Но все же возможны ситуации в которых однородность структур данных становится существенным минусом. В качестве примера рассмотрим структуры данных из задачи о Ханойской башне. Вот они:

```
a:ARRAY 4 OF mas;
h:ARRAY 4 OF INTEGER;
```

Два массива, описывают одну сущность. Отслеживание связи между массивами полностью на совести программиста. В задаче о Ханойской башне это не вызывает особенной проблемы, но лишь потому, что программа не велика. Отслеживание связей может превратиться в заметную проблему, с увеличением объема программы. Но конечно и просто разумно описать все данные в одном месте.

Конструкция записи, позволяет определить структуру ханойской башни следующим образом:

```
TYPE
    mas=RECORD
        Disk: ARRAY 10 OF INTEGER;
        h:INTEGER;
    END;
VAR
    a: ARRAY 4 OF mas;
```

Сейчас «a» означает не массив массивов, как ранее, а массив сложных структур, каждая из которых состоит из двух компонентов: массив дисков, находящихся на подставке и высота башни стоящей на подставке. Заметим, что в данной конструкции, мы, во-первых, ушли от двумерного массива, а во-вторых, для хранения данных о высоте вообще нет необходимости в массиве, так что можно утверждать, что конструкция не только внутренне логичнее, но и проще.

Как получить доступ к компоненту записи. Для ответа на вопрос, запишем более простую конструкцию:

```
VAR
rec: RECORD
a:INTEGER;
N: CHAR;
END;
```

Здесь объявлена запись с именем rec и двумя компонентами, также имеющими имена. Имя rec общее для обоих компонентов. Для получения доступа к компоненту необходимо указать имя записи, к которой он принадлежит и уже потом его собственное имя. Это также, как фамилия человека указывает его принадлежность

к роду и затем имя указывает на него самого. Доступ к компонентам осуществляется так:

```
rec.a:=1;
rec.N:='a';
```

Это общий принцип. Он достаточно прост, но для закрепления разберем несколько более емких примеров.

Пример 1. Массив записей

```
TYPE
    shablon=RECORD
        a: INTEGER;
        N: CHAR;
    END;
VAR
    mas: ARRAY 10 OF shablon;
```

Доступ: mas[1].a:=1;

Пример 2. Массив записей содержащих массив

```
TYPE
    shablon=RECORD
        a:ARRAY 10 OF INTEGER;
        N: CHAR;
    END;
VAR
    mas: ARRAY 10 OF shablon;
```

Доступ: a[1].a[9]:=1;
Здесь имя компонента совпадает с именем записи. Это вполне допустимо. Переменные типа запись можно присваивать друг другу. Ниже пример

Листинг 97

```
PROCEDURE Calc;
TYPE
    shablon=RECORD
        a:INTEGER;
        N:CHAR;
    END;
VAR
    a,s:shablon;
BEGIN
a:=s;
END Calc;
```

Но если записи разного типа, то присвоение уже невозможно. Даже если они объявлены так:

Листинг 98

```
PROCEDURE Calc;
VAR
    s: RECORD
        a:INTEGER;
        N:CHAR;
    END;
    a:RECORD
        a:INTEGER;
        N:CHAR;
    END;
BEGIN
a:=s; (*Здесь несовместимое присваивание*)
END Calc;
```

А вот так, опять допустимо:

Листинг 99

```
PROCEDURE Calc;
VAR
    a,s:RECORD
        a:INTEGER;
        N:CHAR;
    END;
BEGIN
a:=s;
END Calc;
```

Возможность вложения записи в запись позволяет создавать структуры данных любой сложности. Пример:

Листинг 100

```
PROCEDURE Calc;
TYPE
    shablon=RECORD
        N:INTEGER;
        a:ARRAY 10 OF INTEGER
    END;
VAR
    a:RECORD
        rec:shablon;
        N:CHAR;
    END;
BEGIN
a.rec.a[5]:=5;
END Calc;
```

Здесь в качестве компоненты записи выступает запись, в которой есть компонент массив. Поэтому чтобы добраться до элемента массива необходимо указать

имя объемлющей записи, затем вложенной и лишь затем имя массива и индекс элемента. Понятно, что увеличение глубины вложенности отражается лишь в удлинении имени, в котором, начиная с имени самой объемлющей записи, перечисляются через точку имена всех вложенных компонентов.

Записи, совершенно необходимы разработчикам баз данных, но их применение разумеется значительно шире. Поставим задачу разработки длинной арифметики. Под длинной арифметикой подразумевают набор операций с числами имеющими длину невыразимую ни одним основным типом. Ситуация будет таковой например, если числа имеют длину в несколько десятков знаков.

Такого рода задачи не встречаются на каждом шагу, поэтому создавать такие типы, как основные и разрабатывать соответствующие операции было бы слишком накладно, но все же это очень полезная задача. Напишем только процедуру суммирования двух чисел.

Определимся со структурой данных, представляющей число. Длинные числа могут иметь различную длину. Для выполнения же операций длина должна быть известна. Учет длины возможно осуществить следующими способами:

- записывать все числа одной длиной, при этом старшие не значащие разряды заполнять нулями;
- в первый старший не значащий разряд записывать специальный символ означающий конец записи числа. Для того массив придется объявлять символьным типом;
- для каждого числа вводить дополнительную переменную величину имеющую смысл длины числа или номера старшего (или наоборот младшего) разряда.

Без структуры RECORD последний вариант означает появление двух несвязанных между собой структур данных. С использованием записей, длину числа и массив цифр (число, конечно же, будем представлять целочисленным массивом) самого числа можно разместить в одной структуре записи.

```
TYPE
    Num=RECORD
        a:ARRAY 1000 OF INTEGER;
        N:INTEGER;
    END;
```

Листинг 101

```
PROCEDURE Sum(a,b:Num);
VAR
    max,i,sum:INTEGER;
    res: Num
BEGIN
IF a.N>b.N THEN
max:=a.N;
ELSE
```

```
    max:=b.N;
    END;
    i:=0;
    sum:=0;
    (*Процесс поразрядного суммирования*)
    WHILE i<=max DO
        IF i<=a.N THEN
            sum:=sum+a.a[i];
        END;
        IF i<=b.N THEN
            sum:=sum+b.a[i];
        END;
        res.a[i]:=sum MOD 10;
        sum:=sum DIV 10;
        i:=i+1;
    END;
    (*Определение ситуации переполнения для добавления разряда*)
    IF sum>0 THEN
        res.N:=max+1;
        res.a[max+1]:=sum;
    ELSE
        res.N:=max;
    END;
    END Sum;
```

Полученная процедура ничего не возвращает, просто сохраняет число-результат в величине res. Этот недостаток вы сможете исправить при желании, после более близкого знакомства с методами передачи данных в процедуры (вторая глава).

В завершении главы рассмотрим две задачи, в которых ограничимся представлением структур данных.

Задача 42. Дано некоторое количество прямоугольных плит разного размера. Необходимо выяснить, можно ли покрыть ими заданную площадь.

Решение:

Так как о плитах известно только то, что все они прямоугольные, то видимо задача решается перебором. Что можно сказать о структуре данных?

Каждая плита определяется двумя размерами длиной и шириной. Кроме того, необходима переменная для учета количества плит каждого размера. Эту информацию можно описать следующей структурой данных:

```
    TYPE
        P=RECORD
            Lx, Ly, N:INTEGER;
        END;
    VAR
    s:ARRAY 10 OF P;
```

Задача 43. Построить модель движения планет солнечной системы.
Решение:
Наша цель, всего лишь описать структуру данных. Для этого учтем следующее:

- текущее положение планет описывается тремя координатами;
- движение описывается тремя компонентами скоростей и тремя компонентами ускорений;
- для расчета ускорений согласно законам Ньютона необходимо знать массы планет.

Договоримся, для упрощения ситуации, что все величины выражаются вещественными числами и что математические проблемы расчетов всех указанных величин нас сейчас не интересуют. Заметим, что все необходимые величины, кроме массы входят в расчеты тройками, при этом каждый элемент тройки связан с системой координат. Этот факт можно отразить следующим типом

```
TYPE
    point=RECORD
        x, y, z: REAL;
    END;
```

Имея такой тип, можно создать тип – планета

```
TYPE
    planets=RECORD
        coord, v, a: point;
        m: REAL;
    END;
```

А структура данных описывающая солнечную систему это очевидно массив:

```
VAR
    planet: ARRAY 9 OF planets;
```

Доступ к координате x, третьей планеты опишется, следующим сложным именем: planet[2].coord.x, а например к соответствующей компоненте ускорения следующей цепочкой planet[2].a.x, имя x присутствует в обоих цепочках, но это имена разных величин.

Задачи для самоконтроля (задачи на разработку структуры данных):

1. Структура данных для описания геометрической фигуры представленной координатами N – точек.
2. Описать структура пикселя:
   - координаты x, y
   - цвет

3. Структура данных паспорт РФ. Для строковых данных использовать символьные массивы. Учесть данные:
   - Серия
   - Номер
   - ФИО
   - Когда и кем выдан
4. Структура оглавления книги. Оглавление состоит из нескольких уровней
   - Тома, состоящие из частей и имеющие название.
   - Части состоящие из глав и имеющие названия
   - Главы, имеющие название и интервал страниц
5. Структура для хранения информации о шахматном ходе. Для каждого хода учесть:
   - Цвет игрока.
   - Поле – начало хода
   - Поле – завершение хода. Имя полей, в шахматах состоит из двух компонент: буква и цифра.
   - Наименование фигуры. Можно использовать первые буквы наименований фигур.
6. В игре ГО позиция состоит из камней (фишек) выставленных на перекрестиях линий доски. Стандартная доска ГО 19 × 19. Важной позиционной структурой является группа. Группу образуют камни являющиеся соседями по горизонтали или вертикали. Создать структуру для описания группы ГО. В структуре должна быть сохранена информация о каждом камне входящем в группу и количество камней входящих в группу. Для каждого камня должна быть записана информация о его положении на доске и о его 4-х возможных соседях.
7. Структуру общественного транспорта города можно представить следующим образом: Единица структуры – маршрут. Определяется маршрут номером и типом транспорта. В тип транспорта кроме вместимости транспортного средства полезно включить стоимость проезда, для упрощения будем считать, что для каждого маршрута она фиксирована по отношению к остановкам, но может отличаться от маршрута к маршруту. Кроме описания типа транспорта в маршрут необходимо включить перечень остановок (названия) и к каждой остановке привязать маршруты, включающие эту остановку. Опишите соответствующую структуру данных.
8. Программу можно представить в виде набора блоков двух типов: либо блок выполняет некую последовательность операций и затем передает управление другому блоку, либо блок проверяет на истинность некое условие и в зависимости от результатов проверки передает управление на соответствующие блоки (например при истинном условии на блок A при ложном на блок B). Опишите структуру данных, в которой учитывается только название, тип блока, способ передачи управления и естественно условие (как переменная типа BOOLEAN).

9. Результатом производственного процесса могут быть изделия нескольких видов. Каждый вид идентифицируется наименованием. Каждый вид изделия требует затрат ресурсов имеющихся у предприятия (электроэнергия, сырье, полуфабрикаты и т.д.). Для каждого изделия необходимо также описать и расход ресурса, куда входит название ресурса и количество. Изделие имеет себестоимость. Описание ресурса кроме наименования также включает стоимость.
10. Анализ арифметического выражения компилятором предполагает построение дерева операций (пример ниже). Создайте конструкцию для хранения структуры данных дерева. Для упрощения будем полагать, что все операции выполняются над целыми числами, участвуют только 4 операции: сложение, вычитание, умножение, деление, выражение может содержать скобки.

Пример дерева операций: (6*((1-2) + 3)+ 4) -7/2

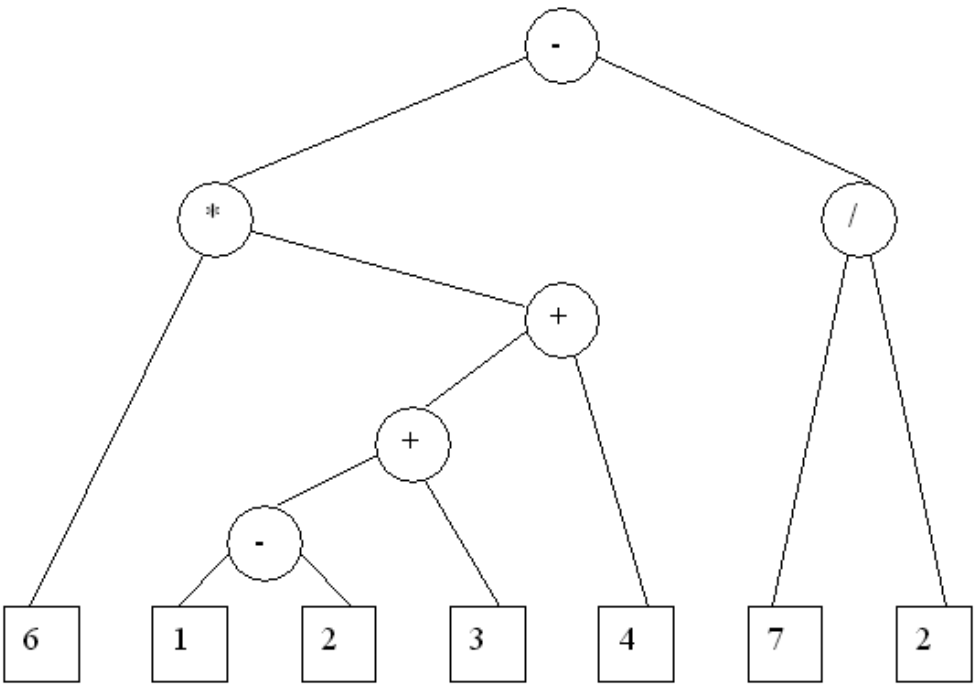

Рис. 1.3. Дерево операций

Указательные типы

Объявление вида VAR a: INTEGER; приводит к выделению памяти под переменную величину целого типа. Память, при этом оказывается занятой на все время работы процедуры, для которой описана переменная. Если переменная окажется не нужна, занятую ей память вернуть все равно нельзя. В случае переменной основного типа это наверное не представляет проблемы. Если же речь идет о величинах занимающих значительную память (например массивы записей), то издержки существенно возрастают.

Описанная ситуация общая для любой программисткой области, поэтому любой язык программирования предоставляет структуры данных созданием и уничтожением которых можно управлять. Такие структуры данных называются динамическими. Пример использования динамического массива:

Листинг 102

```
MODULE Example;
IMPORT In, StdLog;
PROCEDURE Calc*;
TYPE
    mas=ARRAY 100 OF INTEGER;
VAR
    a: POINTER TO mas;
    N,k:INTEGER;
BEGIN
NEW (a);
In.Open;
N:=-1;
WHILE In.Done DO
    N:=N+1;
    In.Int(a[N]);
END;
FOR k:=0 TO N-1 DO
    StdLog.Int(a[k]);
END;
END Calc;
END Example.
```

Как видно, для создания динамического массива необходимы две вещи: во-первых, требуется его объявить, как и обычный массив. Отличие от обычного массива в резервировании памяти. В момент объявления динамического массива память не выделяется. Для выделения памяти необходимо выполнить процедуру NEW.

По окончанию работы с динамической величиной занимаемую ей память можно вернуть в свободную область. Для этого в КП нет специальных процедур, эту работу автоматически выполняет сборщик мусора. Он самостоятельно находит уже

неиспользуемую память и возвращает ее в свободную область. Для того, чтобы обработчик смог выполнить свою работу достаточно указатель на неиспользуемую структуру пометить значением NIL. После этого попытка доступа к данным вызовет сообщение о ошибке. Для демонстрации завершим предыдущий пример так:

Листинг 103

```
FOR k:=0 TO N-1 DO
    StdLog.Int(a[k]);
END;
a:=NIL;
FOR k:=0 TO N-1 DO
    StdLog.Int(a[k]);
END;
END Calc;
```

Первый оператор цикла выполнит свою работу как обычно, а уже второй записанный после a:=NIL; вызовет сообщение о ошибке во время работы программы. Разработчик только должен иметь ввиду, что ошибки такого рода не обнаруживаются во время компиляции, поэтому использование динамических структур требует большой внимательности.

КП разрешает создавать указательные типы только со сложными структурами данных: массивами и записями. Следующая попытка:

```
s: POINTER TO INTEGER;
```

будет воспринята компилятором как ошибочная. Еще одна возможность объявить динамическую структуру:

```
a: POINTER TO ARRAY OF INTEGER;
```

В объявлении такого вида, указатель привязывается к открытому массиву, то есть массиву объявленному без указания длины. В этом случае выделение памяти производится следующим образом:

```
NEW (a,10); или NEW(a, N);
```

Процедуре new, кроме имени указателя сообщается и размер массива, под который требуется память. Открытый массив, также как и обычный может быть многомерным.

Пример двумерного открытого массива:

```
a: POINTER TO ARRAY OF ARRAY OF INTEGER;
```

Аналогично можно объявить массив любой размерности. Но в объявлении многомерных, открытых массивов есть существенная особенность. Вернемся ненадолго к обычным массивам. Объявление вида:

```
a: ARRAY N1, N2 OF INTEGER;
```

есть сокращенная форма следующей записи:

```
a: ARRAY N1 OF ARRAY N2 OF INTEGER;
```

Естественно для открытых массивов сокращенная форма уже невозможна, не ясно как перечислить через запятую несуществующие длины.

Форма объявления динамических записей такая же как и динамических массивов. Пример:

```
rec: POINTER TO RECORD
        a:INTEGER;
        c:CHAR;
    END;
```

или так:

```
TYPE
    record=RECORD
        a:INTEGER;
        c:CHAR;
    END;
VAR
    rec: POINTER TO record;
```

Для доступа к компонентам записи используется операция разыменования «^». Например: `rec^.a:=1;`

В остальном порядок работы с указательными типами такой же как и с обычными, поэтому специальных примеров рассматривать не будем. Вы для тренировки можете взять любые задачи на массивы, записи и переписать их с использованием указательных типов.

Связные списки

Техника работы с указательными типами, как уже было замечено не заключает в себе почти ничего нового, кроме одной процедуры выделяющей память и операции возврата памяти, которую впрочем выполняет сборщик мусора не управляемый программистом. Но тем не менее указательный тип несет в себе принципиально новую возможность. Рассмотрим следующую конструкцию:

```
TYPE
    pointer=POINTER TO record;
    record=RECORD
        a:INTEGER;
        next: pointer;
    END;
VAR
    uk: pointer;
```

В структуре записи есть любопытный компонент `next`, указывающий на запись того же типа, компонентом которой он является. Это означает, что динамическую структуру типа `record`, можно связать с другой такой же структурой и даже

более того, можно выстроить цепочку структур типа record, таких что каждая будет нести в себе информацию о следующей. Такая цепочка называется связным списком. Для примера построим программу с двумя действиями: построение связного списка и проход по нему.

Листинг 104

```
MODULE Example;
IMPORT In, StdLog;
PROCEDURE Calc*;
TYPE
    pointer=POINTER TO record;
    record=RECORD
        a:INTEGER;
        next: pointer;
    END;
VAR
    uk,uk1: pointer;
    k:INTEGER;
BEGIN
NEW(uk);uk1:=uk;
uk^.a:=1;
k:=2;
WHILE k<=10 DO
    NEW(uk^.next);
    uk:=uk^.next;
    uk^.a:=k;
    k:=k+1;
END;
k:=1;
uk:=uk1;
WHILE k<=10 DO
    StdLog.Int(uk^.a);
    uk:=uk^.next;
    k:=k+1;
END;
END Calc;
END Example.
```

Разберем процесс работы по частям. Создание связного списка начинается с выделения памяти для первой записи: NEW(uk); к команде uk1:=uk вернемся немного позже. После выделения памяти под первую запись можно придать значение компоненту «a», что и делается до цикла командой uk^.a:=1;

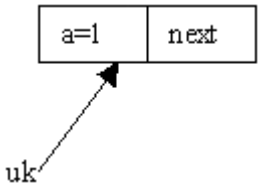

Рис 1.4. Элемент связного списка

Компонент next записи показывает в никуда, переменная uk указывает на запись. Далее управление передается в цикл. Первая операция NEW(uk^.next); приводит к созданию новой записи связанной с первой:

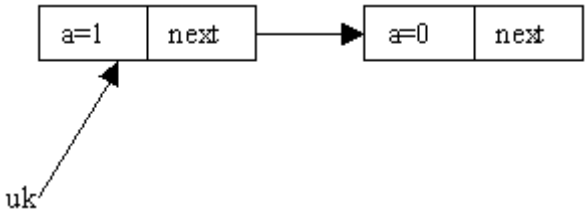

Рис. 1.5. Создание элемента связного списка

Запись создана, ее компонент a инициализирован нулем, компонента next указывает в никуда. Следующая задача присвоить значение компоненте a второй записи, но сделать это невозможно, так как ко второй записи нет доступа uk по прежнему указывает на первую. Операция uk:=uk^.next; меняет ситуацию:

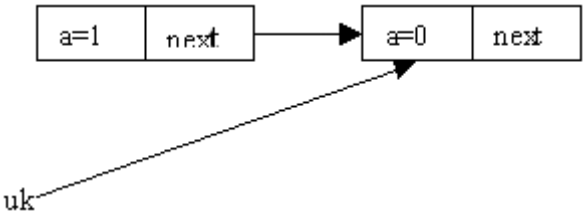

Рис. 1.6. Переход по связному списку

uk указывает на вторую запись, поэтому становится возможным присвоение uk^.a:=k; без предыдущего присвоения, эта операция изменила бы значение компонента a первой записи. А сейчас получаем следующую картину

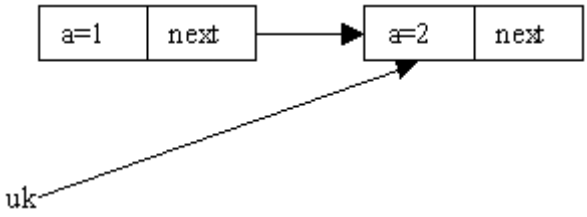

Рис. 1.7. Присвоение значения следующему элементу

И следующий шаг цикла повторяет все указанные операции, только уже в отношении следующей записи. Повторный проход связного списка заключается в печати компонента «a» текущей записи и в переходе на следующую. Сейчас можно объяснить зачем после создания первой записи выполняется команда uk1:=uk. В процессе создания связного списка указатель uk создает записи и уходит вперед по связному списку, естественно забывая адреса предыдущих записей. Вернуться по связному списку назад нельзя, так как записи не помнят своих предшественников. Выход следующий, необходимо в момент создания первой записи списка запомнить ее адрес еще в одном указателе. В нашем случае это указатель uk1. Тогда после прохода по связному списку мы можем вернуться к началу, адрес которого хранится в указателе uk1. Этот возврат выполняется перед циклом второго прохода присвоением: uk:=uk1;

В этой и последующих задачах, в используемых записях будет только одно содержательное поле и оно для упрощения всегда будет целого типа. Конечно же, содержательных полей может быть сколько угодно много. Указателей на запись того же типа может быть также различное количество и они могут использоваться для разных целей, например, можно создать указатель содержащий адрес на предыдущий элемент связного списка, тогда появится возможность движения по списку в обе стороны.

Далее, рассмотрим несколько полезных операций над связными списками.

Задача 44. Дано два связных списка. Назовем их A и B. Подклеить связный список B к списку A.

Решение:

Список A заканчивается некоторой записью, поле next которой никуда не указывает. Для того чтобы подклеить к концу A список B достаточно адрес первой записи B присвоить упомянутому выше полю, для чего:

- установим некий указатель на конец списка A (для чего конечно список A необходимо пройти от начала до конца);
- установим другой указатель на начало списка B;
- выполним операцию присвоения.

Договоримся, поле записи, содержащий адрес следующей записи списка всегда называть next. Заметим также, что A и B, в тексте программы это имена указателей отвечающих за общение со списками, мы же эти имена используем и для обозначения списков. Это удобно, но необходимо помнить, что указатель не привязывается жестко к конкретному списку, на что именно указывает конкретный указатель в определенный момент времени это вопрос не имен, а логики программы. Нет ничего противозаконного в присвоении указателю адреса любой записи, любого списка, при условии совпадения типов указателя и записи.

Листинг 105

```
MODULE Example;
IMPORT StdLog;
```

```
PROCEDURE Calc*;
TYPE
    pointer=POINTER TO record;
    record=RECORD
        a:INTEGER;
        next: pointer;
    END;
VAR
    A,B,A1,B1:pointer;
    k:INTEGER;
BEGIN
NEW(A);A1:=A;
NEW(B);B1:=B;
A.a:=1;B.a:=1;
FOR k:=2 TO 10 DO
    NEW(A.next);NEW(B.next);
    A:=A.next;B:=B.next;
    A.a:=k;B.a:=k*k;
END;
B:=B1;
A.next:=B;
A:=A1;
FOR k:=1 TO 20 DO
    StdLog.Int(A.a);
    A:=A.next;
END;
END Calc;
END Example.
```

В тексте программы не используется операция разыменования «^» для доступа к компонентам записи. В предыдущих примерах она использовалась. Это означает, что в КП операция разыменования может выполняться автоматически. Далее, для распечатки результата используется только один указатель – A, с его помощью выполняется проход и по той части, которая ранее была списком A и по той части которая ранее была списком B. Это как раз и подтверждает, сказанное выше о взаимоотношениях списков и указателей на них указывающих.

Задача 45. Дан связный список. Выбросить из него с позиции L, N – записей.
Решение:

В сообщении о КП сказано, что освобождением памяти от ненужных динамических переменных занимается специальная программа – сборщик мусора, выполняющая свою работу автоматически, без ведома программиста. Величина считается ненужной, если на нее не указывает ни один указатель. Например, если

```
a: POINTER TO ARRAY OF INTEGER;
то a:=NIL;
```

приведет к высвобождению памяти.

С уничтожением записей связного списка ситуация немного сложнее. Предположим, что указатель A в некоторый момент времени видит некую запись. И в этот момент выполнится операция A:=NIL. Это безусловно приведет к потере связи между указателем и связным списком, но этого будет недостаточно для высвобождения памяти, так как текущая запись очевидно связана с предыдущей, а следовательно существует указатель показывающий на данную запись.

Еще одно соображение. Рассмотрим фрагмент (тип записи, такой как и во всех примерах)

Листинг 106

```
NEW(A);
A.a:=1;
FOR k:=2 TO 10 DO
    NEW(A.next);
    A:=A.next;
    A.a:=k;
END;
```

Указатель A по мере построения списка уходит вперед. Запомнить же начало списка в дополнительном указателе программист не удосужился. Это означает, что первая запись оказалась никому не нужной и будет уничтожена. После освобождения памяти от первой записи, ненужной окажется уже вторая и т.д. И спустя некоторое время, в памяти компьютера останется только одна запись, та которую по завершению цикла видит указатель A.

Из сказанного следует, что для уничтожения цепочки связного списка, надо разорвать список в точке L и перекинуть связь за N записей,

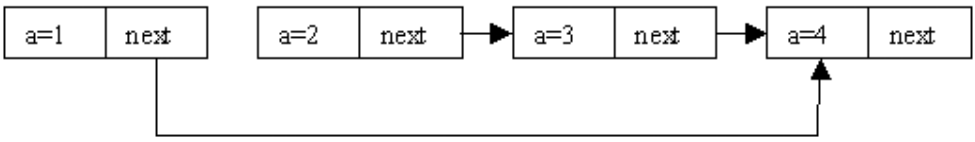

Рис .1.8. Переопределение связи

так как это показано на рисунке. Ниже фрагмент программы.

Листинг 107

```
A:=A1; (*переход на начало списка*)
(*переход на запись с номером L*)
FOR k:=1 TO L-1 DO
    A:=A.next
END;
C:=A;
```

```
(*Отсчет N записей*)
FOR k:=1 TO N DO
    A:=A.next;
END;
C.next:=A;
```

В программном фрагменте никакому указателю не присваивается NIL. Это и не нужно. После выполнения присваивания C:=A; запись с номером L+1 окажется «ничьей».

Задача 46. Обрезать связный список с позиции L.
Решение:
А сейчас NIL необходим.

Листинг 108

```
A:=A1; (*переход на начало списка*)
(*переход на запись с номером L*)
FOR k:=1 TO L-1 DO
    A:=A.next
END;
A.next:=NIL;
```

Задача 47. Дано два связных списка A и B. Вставить список B в список A, начиная с позиции L.
Решение:
Выполним следующие действия:

1. Пройдем по списку B до позиции L, запоминая при этом адрес предшествующей записи.
2. Предшествующую запись свяжем с началом списка B.
3. Конец списка B, свяжем с продолжением списка A (запись L).

Сказанное, проиллюстрировано на рис. 1.9:

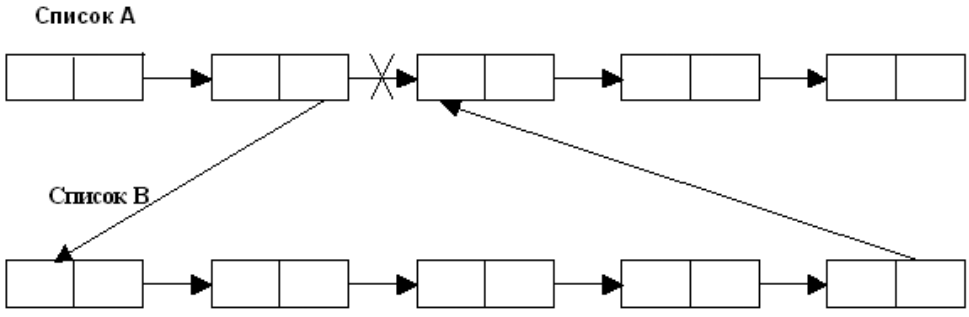

Рис. 1.9. Вставка связного списка

Листинг 109

```
PROCEDURE Calc*;
TYPE
    pointer=POINTER TO record;
    record=RECORD
        a:INTEGER;
        next: pointer;
    END;
VAR
    A,B,A1,B1,C:pointer;
    k,L:INTEGER;
BEGIN
In.Open;
In.Int(L);
NEW(A);A1:=A;
NEW(B);B1:=B;
A.a:=1;B.a:=1;
FOR k:=2 TO 10 DO
    NEW(A.next);NEW(B.next);
    A:=A.next;B:=B.next;
    A.a:=k;B.a:=k*k;
END;
A:=A1;
FOR k:=1 TO L-1 DO
    C:=A;
    A:=A.next;
END;
C.next:=B1;
B.next:=A;
A:=A1;
FOR k:=1 TO 20 DO
    StdLog.Int(A.a);
    A:=A.next;
END;
END Calc;
```

Задача 48. Создать связный список с возможностью обхода в двух направлениях.
Решение:

Обеспечить возможность прохода в двух направлениях можно двумя указателями, один из которых, будем как и прежде называть next, связывает текущую запись с последующей, и второй, будем называть его back, связывает текущую запись с предыдущей.

Листинг 110

```
PROCEDURE Calc*;
TYPE
    pointer=POINTER TO record;
```

```
    record=RECORD
        a:INTEGER;
        next,back: pointer;
    END;
VAR
    uk,uk1,uk2: pointer;
    k:INTEGER;
BEGIN
NEW(uk);uk1:=uk;
uk.a:=1; k:=2;
WHILE k<=10 DO
    NEW(uk.next);
    uk2:=uk;
    uk:=uk.next;
    uk.back:=uk2;
    uk.a:=k;
    k:=k+1;
END;
k:=1;
WHILE k<=10 DO
    StdLog.Int(uk.a);
    uk:=uk.back;
    k:=k+1;
END;
END Calc;
```

Три ключевых команды:

```
uk2:=uk;
uk:=uk.next;
uk.back:=uk2;
```

Компонент back должен сохранить адрес очередной записи в последующей. Адрес очередной на каждом шаге цикла создания списка, находится в указателе uk. Но его использовать нельзя, так как при переходе к следующей записи (то есть той в которой требуется заполнить поле back) он забывает адрес «текущей» (в кавычках, так как она после перехода уже не будет текущей, она станет предыдущей, а uk всегда показывает на текущую). Поэтому и используется такая схема: адрес присваиваемый на последующем шаге полю back запоминается в специальном указателе на предыдущем шаге.

Это были полезные операции над списками. Они вам обязательно понадобятся, для решения реальных задач. А сейчас, для завершения исследования свойств линейных связных списков, рассмотрим одну реальную задачу. Пусть это будет уже решенный нами пузырек, только сейчас реализуем его не на массивах, а на линейном связном списке.

Решение:

Логика программы останется без всякого сомнения точно такой же, изменится только структура данных. На что это окажет влияние разберите самостоятельно:

Листинг 111

```
PROCEDURE Calc*;
TYPE
    pointer=POINTER TO record;
    record=RECORD
        a:INTEGER;
        next: pointer;
    END;
VAR
    uk,uk1,uk2: pointer;
    a,N,k,j,c:INTEGER;
BEGIN
NEW(uk);uk2:=uk;
In.Open;
(*Предполагается наличие хотя бы одного числа во входном потоке*)
In.Int(uk.a); N:=1;
REPEAT
    In.Int(a);
    IF In.Done THEN
        NEW(uk.next);
        uk:=uk.next;
        uk.a:=a;
        N:=N+1;
    END;
UNTIL ~In.Done;
FOR k:=1 TO N-1 DO
    uk:=uk2;uk1:=uk2.next;
    FOR j:=1 TO N-k DO
        IF uk.a>uk1.a THEN
            c:=uk.a;uk.a:=uk1.a;uk1.a:=c;
        END;
    uk:=uk.next;uk1:=uk1.next;
    END;
END;
uk:=uk2;
REPEAT
    StdLog.Int(uk.a);
    uk:=uk.next;
UNTIL uk=NIL;
END Calc;
```

Циклы с условием завершения используются для того, чтобы о них не забыть. И главное, обратите внимание на цикл вывода. В нем не используется переменная N – знающая количество записей связного списка. Указатель идет по списку до тех пор пока не встретит значение NIL, иначе говоря пока не упрется в конец списка. Это еще одно небольшое преимущество перед массивами. Нет жесткой необходимости запоминать сколько было создано записей. Это можно выяснить при любом проходе.

Задачи для самоконтроля (во всех задачах предполагается, что в связном списке записаны целые числа):

1. Найти сумму числовых полей связного списка.
2. Найти наибольший элемент связного списка.
3. Определить, сколько в списке различных чисел.
4. В списке имеются, как положительные, так и отрицательные числа. Разделить их по двум различным связным спискам.
5. Дан связный список с двусторонней связью (next – вперед, back – назад). Переопределить указатели связи (next – назад, back – вперед)
6. Создать кольцевой связный список (последний элемент связан с первым) и осуществить сдвиг кольца на N позиций.
7. Удалить из списка все нули.
8. Дано два списка. Выяснить, являются ли они равносоставленными.
9. Создать список с дополнительным указательным полем, позволяющим двигаться по списку с шагом 2.
10. Создать список с дополнительным указательным полем, позволяющим двигаться только по положительным элементам списка.

Деревья

Во всех примерах выше, связные списки – линейные. В линейном списке у каждой записи (кроме первой) есть одна предшествующая и есть (кроме последней) одна последующая. В связном списке, который мы далее будем называть деревом, у каждой записи один предшественник, но потомков может быть несколько. Смотрите рисунок:

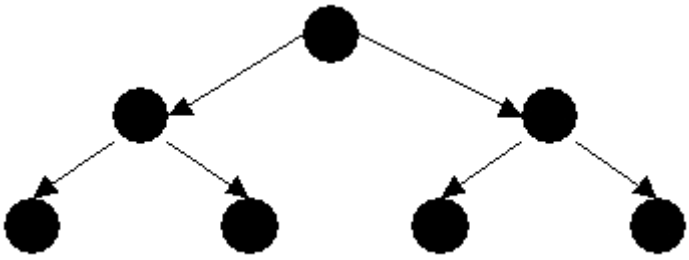

Рис. 1.10. Двоичное дерево

На рисунке для упрощения изображено двоичное дерево, то есть дерево, в котором у каждого узла (кроме последнего), есть ровно два потомка. Конечно же двоичное дерево не единственно возможное. Количество узлов потомков может быть любым, более того, совершенно не обязательно, чтобы количество потомков у всех узлов было одинаковым.

Для моделирования дерева необходимо, в записи объявить два указателя на новую запись. Связный список с двумя указателями мы уже использовали, строя список с возможностью обратного хода, но сейчас немного другая ситуация. Для списка – дерева, необходимы два указателя на *новые* записи, в списке с обратным ходом указатель back использовался для хранения адреса уже имеющейся записи.

Заметим также, что задача построения дерева имеет явно рекурсивный характер. Действительно, каждый узел, за исключением последних порождает двоичное дерево. Следовательно, двоичное дерево можно определить, как узел, порождающий два двоичных дерева, каждое из которых также начинается с некоторого узла, который ..... и т.д.

Следовательно, программу можно построить, как рекурсивную процедуру, деятельность которой сводится к получению адреса записи – узла созданного ранее и двум вызовам себя же для создания левой ветки и правой. А вот как это выглядит.

Листинг 112

```
MODULE Example
IMPORT In, StdLog;
TYPE
    pointer=POINTER TO record;
    record=RECORD
        a:INTEGER;
        left,right: pointer;
    END;
PROCEDURE Tree(tree:pointer;n:INTEGER);
BEGIN
tree.a:=n;
IF n<4 THEN
    NEW(tree.left);Tree(tree.left,n+1);
    NEW(tree.right);Tree(tree.right,n+1);
END;
END Tree;
PROCEDURE View(tree:pointer;n:INTEGER);
BEGIN
StdLog.Int(tree.a);
IF n<4 THEN
    View(tree.left,n+1);
    View(tree.right,n+1);
```

```
END;
END View;
PROCEDURE Main*;
VAR
tree:pointer;
BEGIN
NEW(tree);
Tree(tree,1);
View(tree,1);
END Main;
END Example.
```

В процедуре Main нет указателя запоминающего начало списка, перед запуском процедуры Tree создающей список. Во всех задачах рассмотренных ранее, этот оператор присутствовал. Здесь он тоже присутствует, но неявным образом. В момент вызова процедуры Tree(tree,1), создается копия указателя tree и именно с ней продолжается работа по созданию дерева. Указатель же tree объявленный в Main, остается на месте и продолжает указывать на корень дерева. Именно поэтому, при вызове процедуры View, можно воспользоваться тем же указателем tree. При вызове View с tree опять будет снята копия и передана в View, указатель tree и в этот раз останется на месте, указывая на корень дерева.

Задача 49. В двоичном дереве поменять местами все левые и правые поддеревья.

Решение:

Условие необходимо пояснить рисунком. Пусть дано такое дерево:

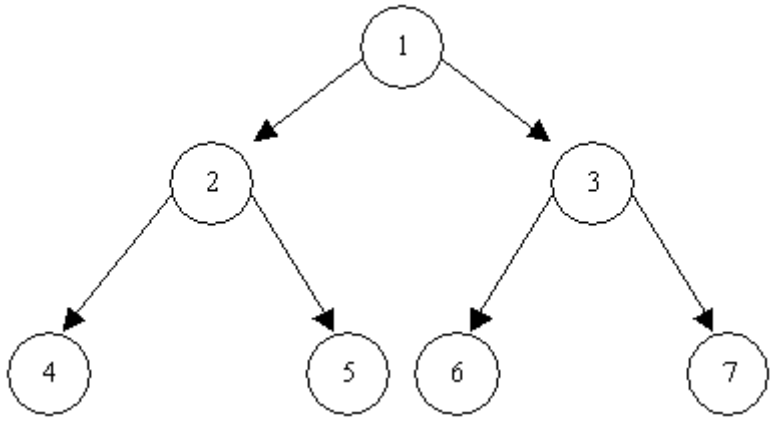

Рис. 1.11. Исходное дерево

Дерево результат, должно выглядеть так

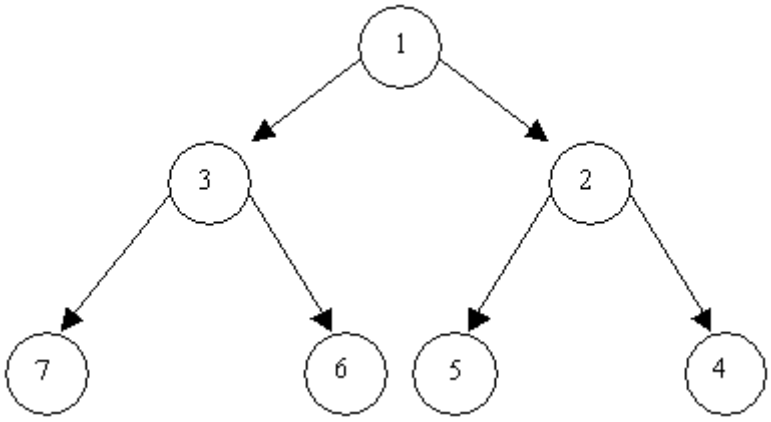

Рис 1.12. Дерево результат

Данная задача демонстрирует еще одно важное преимущество списков перед массивами. Древовидные структуры можно представлять в виде массивов, но для этого необходимо создавать искусственные конструкции мало похожие на деревья. Посредством связных списков можно моделировать деревья непосредственно.

Второе преимущество заключается в очень простой процедуре преобразования деревьев. Например, в нашей задаче необходимо просто поменять местами левые и правые ветви. А это всего лишь три команды присваивания. Как поменять два значения местами, можно посмотреть в задаче сортировки массива пузырьком.

Нет необходимости приводить весь текст решения, он в значительной степени совпадет с предыдущим. В процедуре создающей список изменим способ заполнения числового поля. Для удобства контроля результата, полезно все значения в узлах дерева иметь различные, для чего создадим глобальную переменную sum, инициализируем ее нулем и при заполнении очередного узла, будем sum увеличивать на единицу. Вот так:

```
tree.a:=sum;
sum:=sum+1;
```

Немного изменится процедура Main. Для целей задачи, дерево необходимо распечатывать дважды, до обработки и после. И появляется вызов дополнительной процедуры Change собственно и выполняющей поставленную задачу, по обмену ветвей.

Листинг 113

```
PROCEDURE Main*;
VAR
    tree:pointer;
BEGIN
NEW(tree);
sum:=0;
Tree(tree,1);
View(tree,1);
Change(tree,1);
StdLog.Ln;
View(tree,1);
END Main;
```

И наконец главная процедура Change, получающая на вход очередной узел, меняющая местами его ветви и вызывающая две свои копии, для следующих узлов.

Листинг 114

```
PROCEDURE Change(tree:pointer;n:INTEGER);
VAR
    c:pointer;
BEGIN
IF n<4 THEN
    c:=tree.left;
    tree.left:=tree.right;
    tree.right:=c;
    Change(tree.left,n+1);
    Change(tree.right,n+1);
END;
END Change;
```

Задача 50. Двоичное дерево заполнено, как положительными, так и отрицательными числами. Удалить все узлы (вместе с их поддеревьями), содержащие отрицательные числа.

Решение:

Это тот, случай, когда в дереве – результате от некоторых узлов может выходить менее чем две ветви. Более того, глубина такого дерева результата может быть различной и более того, дерево может быть пустым, если в корне окажется отрицательное число. Процедура создания дерева серьезно не изменится, единственно заполнять его будем числами из входного потока.

Существенно изменится процедура просмотра дерева. До обработки дерево можно конечно просмотреть уже полученной ранее процедурой View, но после обработки View уже не сможет выполнить свою работу, так как изменится количество узлов, ветвей из них выходящих. Для модернизации View вспомним, что указатель в никуда есть NIL.

Листинг 115

```
PROCEDURE Main;
VAR
tree:pointer;
BEGIN
In.Open;
NEW(tree);
Tree(tree,1);
View(tree);
Work(tree);
StdLog.Ln;
View(tree);
END Main;
```

Процедура Main практически не изменилась. Место процедуры Change заняла процедура Work.

Листинг 116

```
PROCEDURE View(tree:pointer);
BEGIN
StdLog.Int(tree.a);
IF tree.left#NIL THEN
    View(tree.left);
END;
IF tree.right#NIL THEN
    View(tree.right);
END;
END View;
```

Процедура View изменилась существенно. Сейчас путь вглубь дерева продолжается только в том случае, если это возможно, то есть соответствующий указатель не равен NIL.

Листинг 117

```
PROCEDURE Tree(tree:pointer;n:INTEGER);
BEGIN
In.Int(tree.a);
IF n<4 THEN
    NEW(tree.left);Tree(tree.left,n+1);
    NEW(tree.right);Tree(tree.right,n+1);
END;
END Tree;
```

Изменение в Tree касаются только ввода, ввод чисел осуществляется из внешнего потока.

Листинг 118

```
PROCEDURE Work(tree:pointer);
VAR
    c:pointer;
BEGIN
IF tree.left#NIL THEN
    c:=tree.left;
    IF c.a<0 THEN
        tree.left:=NIL;
    ELSE
        Work(tree.left);
    END;
END;
IF tree.right#NIL THEN
    c:=tree.right;
    IF c.a<0 THEN
        tree.right:=NIL;
    ELSE
        Work(tree.right);
    END;
END;
END Work;
```

Work – процедура выполняющая работу требуемую по условию. Как она это делает?

- на вход процедура получает неотрицательный узел. Для упрощения договоримся, что первый узел содержит неотрицательное число (первое число в потоке);
- затем процедура просматривает на шаг вперед ветки left и right;
- если какая либо ветка приводит к узлу с отрицательным значением, она обрезается значением NIL.

Задача 51. Создать дерево с возможностью возврата от потомка к предку.

Решение:

Фактически речь идет о дополнительном указателе в записи, могущим хранить адрес предка, то есть узла от которого передан процесс создания дерева. Интересующая нас структура может выглядеть например так:

```
TYPE
pointer=POINTER TO record;
    record=RECORD
        a:INTEGER;
        left,right,back: pointer;
    END;
```

Полностью программу, при желании можете написать самостоятельно. Мы здесь ограничимся одной процедурой создающей дерево.

Листинг 119

```
PROCEDURE Tree(tree:pointer;n:INTEGER);
VAR
    uk:pointer;
BEGIN
In.Int(tree.a);
IF n<4 THEN
    NEW(tree.left);
    uk:=tree.left;
    uk.back:=tree;
    Tree(tree.left,n+1);
    NEW(tree.right);
    uk:=tree.right;
    uk.back:=tree;
    Tree(tree.right,n+1);
END;
END Tree;
```

После создания очередного узла, процедура посредством дополнительного указателя на время переходит к этому узлу и записывает в его поле back адрес узла текущего и уже затем вызывается новая копия tree.

Задачи для самоконтроля (все узлы дерева заполняются целыми числами, все используемые деревья двоичные, если не оговорено иное)

1. Вычислить сумму чисел записанных в узлах двоичного дерева.
2. Найти наибольшее число на дереве.
3. Дерево построено следующим образом:
   - Значения для узлов читаются из входного потока
   - Узел с четным, положительным значением порождает две ветви
   - Узел с нечетным, положительным значением порождает одну ветвь
   - Узел с отрицательным значением является тупиком.

   Требуется вычислить максимальную глубину дерева. Вычислять глубину дерева в процессе его построения запрещается.
4. На дереве есть некоторое количество нулей. Вычислить самый короткий путь к нулевому значению.
5. Найти путь с наибольшим весом. Весом назовем сумму чисел вдоль пути. В качестве ответа достаточно вывести значение веса.
6. Выполнить обмен числовыми значениями между соседними узлами. Соседние узлы – это узлы, имеющие общего предка.
7. В дереве все узлы, кроме тупиковых, заполнены нулями. Тупиковые узлы заполнены положительными числами. Заполнить все дерево числами,

используя следующее правило: каждый узел получает наибольшее значение от своих потомков.

8. Дерево заполнено целыми, как положительными, так и отрицательными числами. Найти поддерево с максимальным весом. Весом назовем сумму значений узлов поддерева. В качестве ответа достаточно напечатать значение веса.
9. Дано некоторое число L. Построить дерево используя следующую числовую функцию:
   - F(1)=1; узел соответствующий аргументу 1 есть узел тупиковый, его числовое значение 1.
   - F(2N)=F(N); числовое значение узла равно 2N, узел порождает только один узел в который отправляется аргумент N.
   - F(2N + 1)=F(N + 1) + F(N); числовое значение узла равно 2N+1, узел порождает два узла, в один из которых отправляется аргумент N+1 и в другой аргумент N.
   - В главе о рекурсии, мы уже рассматривали эту функцию. Сейчас же задача состоит в том, чтобы заполнить некоторое дерево промежуточными значениями этой функции. В корне дерева число L. В тупиковых узлах единицы.
10. Создать дерево, узлы которого хранят не одно числовое значение, а линейный связный список. Все связные списки могут иметь разную длину. Заполняются списки из входного потока положительными числами. Ноль во входном потоке представляет собой команду завершения формирования очередного линейного связного списка.

Файлы

Речь в главе пойдет о так называемых файлах данных. Собственно это уже не средство языка. Файлы, как и весь набор средств ввода – вывода представляют собой дополнительные возможности поставляемые модулями среды BlackBox. Но файлы данных сущность настолько важная, что обойти их в неформальном введении просто нельзя.

В примере ниже две процедуры. Одна из них записывает целые числа в файл. Вторая выполняет обратную работу – читает записанные числа. Операции простые, но для их выполнения необходимы некоторые дополнительные действия:

- определить папку в которой находится или будет создан файл данных;
- привязать переменную файлового типа к соответствующему методу (чтения или записи);
- для записи зарегистрировать файл (после этого он будет реально создан на магнитном носителе в определенной папке).

Листинг 120

```
MODULE Example;
IMPORT Files,Stores, StdLog;
PROCEDURE Do*;
VAR
    f: Files.File;
    wr: Stores.Writer;
    loc: Files.Locator;
    res: INTEGER;
    x,a:INTEGER;
BEGIN
(*Определим папку*)
loc := Files.dir.This("");
loc := loc.This("Files.dat");
(*Увязка файловой переменной*)
(*NEW – новая*)
f := Files.dir.New(loc, FALSE);
wr.ConnectTo(f);
(*Операция записи*)
FOR a:=1 TO 10 DO
    x:=a*a;
    wr.WriteInt(x);
END;
(*Регистрация файла*)
f.Register("file", "dat", TRUE, res);
END Do;
PROCEDURE Do1*;
VAR
    f: Files.File;
    rd:Stores.Reader;
    loc: Files.Locator;
    res: INTEGER;
    x,a:INTEGER;
BEGIN
loc := Files.dir.This("");
loc := loc.This("Files.dat");
(*Old – существующая*)
f := Files.dir.Old(loc, "file.dat", Files.shared );
rd.ConnectTo(f);
FOR a:=1 TO 10 DO
    rd.ReadInt(x);
    StdLog.Int(x);
END;
END Do1;
END Example.
```

Задача 52. Из входного потока вводится последовательность чисел. Записать в файл только положительные, затем в другой процедуре прочитать и вывести на печать.

Решение:

Задача несколько отличается от предыдущей тем, что вторая процедура не знает сколько чисел в файле, а это означает, что читать надо до тех пор, пока не будет считано все, что записано. Запись в файл можно выполнить, так же как и в предыдущем примере, поэтому изменим только процедуру вывода:

Листинг 121

```
PROCEDURE Do1*;
VAR
    f: Files.File;
    rd:Stores.Reader;
    loc: Files.Locator;
    res: INTEGER;
    x,a:INTEGER;
BEGIN
loc := Files.dir.This("");
loc := loc.This("Files.dat");
(*Old – существующая*)
f := Files.dir.Old(loc, "file.dat", Files.shared );
rd.ConnectTo(f);
rd.ReadInt(x);
WHILE ~rd.rider.eof DO
    StdLog.Int(x);
    rd.ReadInt(x);
END;
END Do1;
```

Вывод отличается только следующим циклом:

```
Выполняем первую операцию чтения
rd.ReadInt(x);
Пока операции чтения успешны выполнять
WHILE ~rd.rider.eof DO
    Печатать результат от предыдущей операции чтения
    StdLog.Int(x);
    Выполнить новую операцию чтения
    rd.ReadInt(x);
END;
```

Так как детальное изучение модулей поставляемых с BlackBox, в том числе и модуля Files не является целью неформального введения, то на этом изложение первой части можно считать законченным.

Глава 2

Систематическое введение в КП

Введение

Приступать к изучению данной главы не рекомендуется без проработки первой. Если же материал неформального введение вами усвоен, то видимо язык КП вы понимаете неплохо и все что нужно это дополнить ваши знания языка и немного их систематизировать. Идеально систематизирована информация о языке в сообщении о языке. Сообщение является составной частью документации прилагающейся к среде программирования и принципиально его вполне достаточно. Но сообщение о языке – это очень формализованный текст читать который без специальных навыков достаточно затруднительно. Поэтому вторая глава по своему содержанию и форме представляет собой развернутое сообщение о языке. Определения языка даны средствами формализма Бэкуса-Наура и имеют дополнительные пояснения, дано несколько больше примеров, чем это есть в сообщении. Имеется дополнительная информация, позволяющая лучше понять конструкции языка и его идейную основу. В основном структура сообщения сохранена, но есть некоторые отступления, которые по мнению автора помогут лучше разобраться в тексте. Глава не имеет никаких вопросов для самопроверки и задач для практикума. Этой цели посвящена третья глава – целиком представляющая собой практикум по программированию.

Общие вопросы

Главная проблема общения между человеком и компьютером это огромный смысловой разрыв между естественным языком человека и языком машины. Вот некоторые из различий:

- в естественном языке огромный набор понятий, компьютер использует крайне ограниченный и даже скудный понятийный аппарат;
- естественный язык отражает целое множество мыслительных инструментов используемых человеческим интеллектом. Человек, может обобщать, абстрагировать и т.д. и т.д. Компьютер способен только к воспроизведению алгоритмов;
- человеческие понятия многозначны, их конкретное наполнение зависит от различных контекстов, от культуры и степени развития конкретного человека. Машинные понятия однозначны и практически не изменяются при переходе от машины к машине. Нельзя сказать, что совсем не меняются, но эти изменения не столь значительны.

Можно привести и другие различия, но даже сказанного достаточно, чтобы понять, между человеческим изложением решения задачи и машинно-пригодным находится огромная пропасть. С одной стороны алгоритм, записанный на естественном языке невозможно подвергнуть компиляции (переводу в машинный код), с другой стороны алгоритм, записанный на языке машинных кодов очень труден для понимания.

Выход из положения был найден в виде языка посредника. Такой язык опирается на небольшое количество базовых, строго определенных понятий (однозначно

понимаемых), смысл которых достаточно близок к понятийному аппарату используемому человеком. Такие языки были названы языками высокого уровня.

С появлением языков высокого уровня программирование, как вид деятельности не только стало возможным для большого количества специалистов, работающих в разных прикладных областях, но и дало большие преимущества для профессиональных программистов. Впрочем, можно сказать, что с появлением уже первых языков термин «профессиональный программист» стал в значительной степени размываться.

О борьбе с ошибками

Язык высокого уровня не смотря на высокий уровень строгости не понятен компьютеру и нуждается в переводе на машинный язык. Это в свою очередь создает необходимость разработки специальных программ – трансляторов обеспечивающих возможность такого перевода.

Появление языков высокого уровня и их трансляторов вызвало к жизни важный вопрос – как бороться с ошибками программиста. Вопрос этот конечно стоял всегда, но в эпоху программирования в машинных кодах, ответ на него давался автоматически: все что делает программа лежит на совести программиста. Появление трансляторов ответ на этот вопрос усложнило. Транслятор, конечно не участвует в разработке алгоритма, но фактически участвует в написании программы. Он занимается ее переводом, и в процессе перевода выполняет анализ текста, а значит может обнаруживать какие-то ошибки. Время затрачиваемое программистом на борьбу с ошибками сопоставимо со временем разработки программы, а зачастую и превышает его, поэтому возможность автоматизации поиска ошибок безусловно очень важна.

И вот здесь оказалось, что для минимизации ошибок, далеко не все равно, как устроен язык. Чем больше предоставляет язык возможностей программисту, тем больше программист может совершить ошибок и тем сложнее их будет обнаружить. Поэтому проектирование языка это поиск золотой середины между простотой и ясностью с одной стороны и обилием возможностей с другой.

Существуют различные точки зрения о том, где находится эта золотая середина. Мы же будем придерживаться того мнения, что потери времени на исправление ошибок дороже самых широких возможностей. И язык программирования должен предоставлять лишь то, что является жизненно важным. Язык это базовый минимум, позволяющий писать кристально ясные, хорошо читаемые программы.

Кстати такой подход совершенно не противоречит идее больших возможностей. Надо просто различать две различные сущности: язык программирования и среду программирования, которая может содержать многочисленные расширения языка и обеспечивать любой уровень сложности.

Язык и определение алгоритма

Программирование в своей основе опирается на понятие алгоритма, которое имеет строгое определение, но так уж получилось, что не одно. Поэтому для

разработчика языка прежде всего необходимо решить вопрос, на какое представление о алгоритме он будет опираться. Мы не будем сейчас уходить глубоко в теорию алгоритмов, отметим только, что существуют два основных подхода:

- декларативное программирование;
- императивное программирование.

При декларативном походе программа (алгоритм) понимается, как некая система определений того, что должно получится. При императивном подходе программа понимается, как последовательность действий, выполнение которых некоторым исполнителем приводит однозначно, к требуемому результату.

Наш выбор – императивное программирование. Возможно понимание алгоритма, как последовательности действий наиболее близко человеческому интеллекту. Это так по крайней мере с точки зрения автора этого текста, этой точки зрения придерживаются многие намного более авторитетные люди в области программирования, но конечно специалисты по декларативным языкам найдут аргументы в пользу своей точки зрения.

Минимальный набор действий

Из сказанного выше вытекает задача определения минимально необходимого набора действий. Заметим, что на уровне процессора все происходящее сводится к преобразованиям чисел. Все, что мы на высоком уровне можем делать с числами, записывается арифметическими выражениями вида:

```
Результат = Выражение
```

Это первая базовая языковая возможность. Она называется «ПРИСВАИВАНИЕ». Ее смысл в вычислении выражения записанного справа от равенства и присвоение полученного выражения величине, чье имя указано слева от равенства.

Может возникнуть потребность выполнить некоторую последовательность действий многократно, без многократной их записи. Соответствующая языковая конструкция называется циклом.

Последовательность выполняемых действий может разветвляться в зависимости от результата вычисления некоторых условий. Для организации ветвлений язык предоставляет условную конструкцию. Графически цикл и ветвление можно представить схематически.

Блок-схема цикла (рис 2.1) читается так: пока истинно условие выполняется последовательность действий, если условие ложно, управление передается на команду следующую за циклом.

Блок-схема ветвления (рис. 2.2) читается так: выбор исполняемой последовательности действий происходит в зависимости от истинности условия. После исполнения выбранной последовательности управление передается на команду следующую за ветвлением.

Конечно, это самые общие конструкции, их реализация в языке высокого уровня может быть различной, даже более того, в одном и том же языке, успешно сосуществуют различные реализации, имеющие отличные друг от друга свойства и особенности.

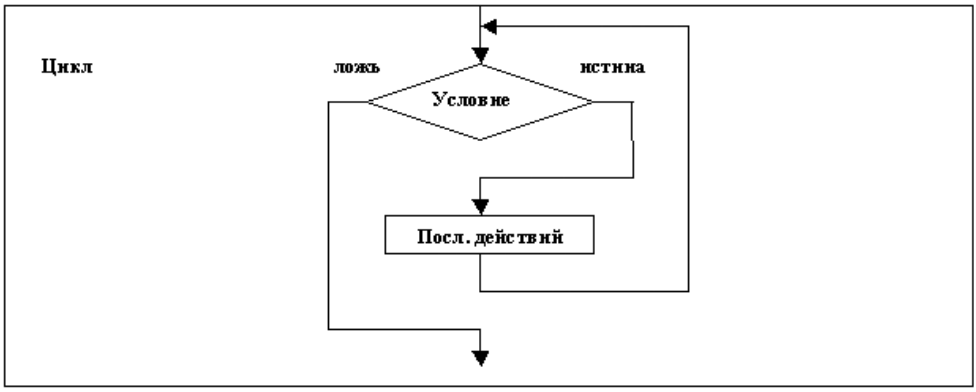

Рис. 2.1. Конструкция цикла

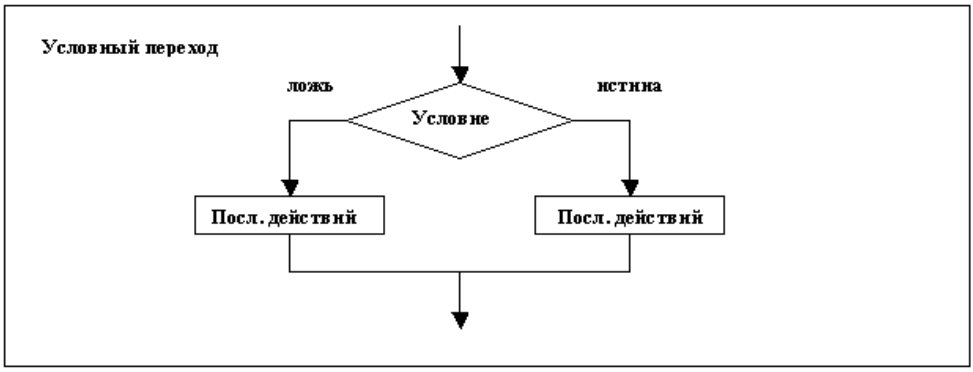

Рис. 2.2. Ветвления

Следующая важная языковая конструкция это процедура (функция, подпрограмма). Процедура – это в некотором смысле программа. Во всяком случае она обладает всеми свойствами программы. Она реализует собственный алгоритм, оформляется отдельно от всех прочих текстов (других процедур, модулей и т.д). Смысл процедур в поддержке процесса декомпозиции задачи. Если задача достаточно велика, при проектировании решения задача разбивается на логически независимые подзадачи, каждая из которых оформляется в виде отдельного решения – процедуры и уже затем из процедур, как из кирпичиков собирается полное законченное решение большой задачи.

Отношение КП к набору операций. Компонентный Паскаль построен по принципу минимальности набора действий. Это не в ущерб функциональности. В языке присутствуют три типа цикла, два типа условного оператора, набор достаточный для полноценно функционирующего языка программирования.

Типы данных

Язык определяется двумя сущностями: набором выполняемых операций и набором структур данных. Данное (величина) определяется именем, значением и типом. Имя дается программистом, на имя не возлагается никакого серьезного функционального смысла, значение величины определяется логикой программы, в процессе ее работы. Наиболее значимо для величины понятие типа. Тип данных определяет множество значений, которое может принимать величина и набор операций, которые над этой величиной можно выполнять.

Структуры данных любого языка программирования бывают двух типов: основные и составные. Основные типы это наиболее простые. Это например различного вида числа и литеры. Составные это массивы – упорядоченные множества однотипных данных и записи – множества данных различного типа. Кроме того, данные делятся на статические – память под которые выделяется на этапе компиляции и динамические, процессом создания и удаления которых можно управлять во время работы программы.

Виды типизации

Вернемся к вопросу о программных ошибках. Значительная их часть сводится к неправильному использованию данных. Состояние ошибки возникает если некая операция использует данные неразрешенного типа. Нет большого смысла складывать литеры, например так:

```
Число=Литера + Литера
```

Такую операцию есть смысл запретить. Наверное можно запретить присваивания в результате которого число занимающее много памяти присваивается переменной под которую выделено мало памяти. В общем незаконными являются операции передачи значений от переменной (выражения) одного типа к переменной другого типа, если ***типы переменных (выражений) не соответствуют друг другу по размеру выделенной памяти и набору допустимых операций***.

Вопрос соответствия в разных языках программирования разрешается по разному, но все возможные варианты сводятся к двум терминам: «сильная типизация» и «слабая типизация».

Слабая типизация. Возможно почти все. Значение любого типа можно передать почти любой величине. Степень этого «почти» в разных языках различна. Не совпадающие типы величин при передаче автоматически или как еще говорят «неявно» преобразуются. Программист должен хорошо понимать правила преобразования и он сам несет ответственность за результат. Языки со слабой типизацией дают большую свободу в обращении с памятью и величинами, но платить за это приходится высокой вероятностью трудно обнаруживаемых ошибок.

Сильная типизация. Передача значения возможна только при совместимости типов. Например, целое можно присвоить целому. В этом случае мы имеем полное совпадение типов. Целое можно присвоить вещественному. Здесь типы не совпадают, но они совместимы, в том смысле, что целое значение можно разместить

в области памяти выделенной под вещественную величину и они совместимы по арифметическим операциям. Вещественное же целому присвоить нельзя, так как под вещественное число требуется больше памяти. Нельзя присвоить литеру никакому числовому типу. Нельзя выполнить присвоение двух записей с разным набором полей. Сильная типизация делится на два вида:

- *сильная структурная типизация*. Два типа считаются одинаковыми, если они совпадают с точностью до структуры. Говорить о таком виде типизации можно разумеется только в отношении типов обладающих структурой (записей). Две записи считаются одинаковыми если они имеют одинаковый набор полей, при этом имена их типов могут различаться;
- *сильная именная типизация*. Две переменных считаются одинакового типа, если совпадают имена их типов. Легко понять, что именная типизация накладывает более жесткие ограничения. Записи имеющие одинаковый тип с точки зрения именной типизации очевидно будут совпадать и с точностью до структуры, обратное же неверно.

Отношение КП к типизации. Компонентный Паскаль является языком сильной именной типизации. Для базовых типов вводится совместимость, то есть присвоение допустимо не только для величин одинаковых типов.

Управление памятью. Сборка мусора

Управление памятью заключается в двух действиях выполняемых по ходу работы программы: выделение памяти под структуры данных и возращение памяти в свободную область. Для статических переменных все вопросы с выделением памяти решаются на этапе компиляции. Для динамических величин при необходимости их использования вызывается специальная процедура выделяющая необходимый объем памяти. Более интересен вопрос возврата памяти в свободную область (для этого действия есть специальный термин «сборка мусора»). Здесь две возможности:

Ответственность за удаление ненужных данных возлагается на программиста. В этом случае в языке предусматривается специальная процедура, вызов которой означает уничтожение структуры данных. Предполагается, что программист не ошибется в оценке ситуации и применит процедуру к ненужным данным в правильной точке.

Конечно же это слишком сильное предположение. Не ошибающихся программистов не бывает, это во-первых, а во-вторых, вопрос о ненужности данных можно решить автоматически, при условии, что точно определено, что значит фраза «структура данных не нужна».

Для КП структура данных считается подлежащей удалению, если нет ни одного указателя (переменная специального вида) содержащего адрес области памяти, в которой хранится структура. Действительно, если нет ни одного указателя на данное, то у программиста просто нет возможности к данному обратится и следовательно разумно принять решение о освобождении памяти, даже если эта ситуация возникла вследствие программисткой ошибки.

Таким образом, для высвобождения данных достаточно всем указателям связанным с не нужными данными присвоить специальное значение NIL (адрес в никуда). Освобождение памяти также произойдет если всем указателям связанным с данным будут присвоены какие-либо иные адреса (не NIL).

Формальные грамматики. Формализм Бэкуса-Наура

Программа есть осмысленное предложение записанное на специальном языке. Смысл текста программы определяется целями программиста и формулировкой задачи, то есть является внешним по отношению к языку. Поэтому предложение записанное на языке программирования может иметь смысл, но это не является его обязательной характеристикой. Внутренней, обязательной характеристикой является соответствие набору правил, описывающих, что является правильным предложением вне зависимости от его смысла. Набор таких правил называется синтаксисом. Следовательно, описание языка программирования есть описание его синтаксиса.

Правила синтаксиса можно описывать неформально. Например, допустимо следующее правило:

Описание цикла с шагом начинается с ключевого слова **FOR**

Или

За ключевым словом ***VAR*** следует блок описания переменных

Данные правила действительно описывают некий синтаксис, но они не точны, неоднозначны и не решают главной задачи построения системы синтаксических правил.

А для построения языка программирования требуется, чтобы синтаксис был описан на определенном строгом языке. То есть ситуация точно такая же, как с описанием алгоритма. Алгоритм является однозначно понимаемым текстом, поэтому для его записи нужен специальный язык, называемый языком программирования. Язык программирования описывается системой синтаксических правил. Каждое такое правило является однозначно понимаемым текстом, поэтому для его написания нужен опять специальный язык называемый формальной грамматикой. Существует два типа формальных грамматик:

Порождающая грамматика. Порождающая грамматика представляет собой алгоритм позволяющий из некоторого минимального набора предложений построить все допустимые предложения данного языка.

Распознающая грамматика. Распознающая грамматика представляет собой алгоритм проверки текста, позволяющий за конечное число шагов, выяснить является ли текст программой на языке описываемом данной системой правил.

Везде, далее, говоря о формальной грамматике, будем иметь ввиду порождающую грамматику. Три важнейших понятия грамматики это терминальный и нетерминальный символы и лексема.

- **терминал (терминальный символ)** – объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение;
- **нетерминал (нетерминальный символ)** – объект, обозначающий какую-либо сущность языка (формулу, выражение и т.д.) и не имеющий конкретного символьного значения;
- **лексема** – последовательность символов, ограниченная специальными символами, например пробелами. И терминальный и нетерминальный символы суть лексемы.

Предложениями языка, заданного грамматикой, являются все последовательности терминалов, выводимые (порождаемые) из начального нетерминала по правилам вывода. Таким образом грамматика языка – это множество терминальных и нетерминальных символов и множество правил вывода. Для описания синтаксиса языка компонентный Паскаль используется так называемый расширенный формализм Бэкуса-Наура (РФБН).

РФБН – это следующий набор правил: альтернативы разделяются символом |. Квадратные скобки [и] означают необязательность заключенного в них выражения, а фигурные скобки { и } означают его возможное повторение (0 или более раз). В случае необходимости для группирования лексем используются круглые скобки (и). Нетерминальные лексемы начинаются с большой буквы (например, Statement). Терминальные лексемы либо начинаются с маленькой буквы (например, ident), либо записаны только большими буквами (например, BEGIN), либо обозначаются цепочками литер (например, «:=»).

Основные термины

Любой язык программирования предполагает небольшой набор основных понятий через которые разворачиваются все языковые смыслы. Перечислим эти понятия (порядок перечисления ни в коем случае не характеризует значимости): *идентификатор, операция, операнд, оператор, переменная, константа, выражение, тип, локальный, глобальный, экспорт, импорт.*

Идентификатор – уникальное имя программного блока (процедуры или модуля) или величины.

Операция – арифметические или логические операции над данными, или операции над символьными цепочками.

Операнд – Аргумент операции. Грамматическая конструкция, обозначающая выражение, задающее значение аргумента операции, иногда операндом называют место, позицию в тексте, где должен стоять аргумент операции.

Оператор – обозначение действия. Операторы различают элементарные и структурированные. Отличие элементарного от структурированного в том, что элементарный оператор не содержит частей, которые сами являются операторами.

Переменная – Величина, чье значение может быть изменено в процессе работы программы и следовательно определяемое в ходе исполнения программы.

Константа – Величина чье значение не может изменяться в ходе работы программы и следовательно определяемое на этапе компиляции.

Выражение – конструкция, описывающая вычислительные правила, в соответствии с которыми комбинируются константы и текущие значения переменных для вычисления других значений посредством применения операций и процедур-функций. Выражения состоят из операндов и операций. Круглые скобки могут использоваться для выражения конкретных связей между операциями и операндами.

Тип – Описание данного. Содержит информацию, позволяющую определить объем памяти, для хранения величины данного типа, набор допустимых над данным операций и совместимость с другими типами данных.

Локальный – понятие используется для ограничения области использования имени. Локальность имени (например переменной) означает, что структурой с данным именем можно пользоваться только в пределах процедуры в которой было дано объявление.

Глобальный – понятие обратное понятию «локальный», означает что структура связанная с именем известна в пределах всего модуля.

Экспорт – все программные конструкции и структуры данных и процедуры определяются в пределах модуля и по умолчанию за пределами модуля не видны. Операция экспорта позволяет передать имя программной конструкции или структуры данных за границы модуля.

Импорт – операция обратная экспорту. Импорт операция позволяющая получить информацию о программной конструкции или структуре данных из другого модуля.

Главное понятие ***программу*** через описанные выше основные понятия можно определить следующим образом: программа это последовательность операторов управляющих вычислением выражений и присвоением полученных значений переменным.

Построение предложений

Программа – это предложение написанное на языке Компонентный Паскаль. Предложение состоит из слов называемых лексемами. Лексема это последовательность символов словаря разделенных пробелами. Лексема может быть:

- идентификатором (именем чего либо);
- числом;
- операцией;
- ограничителем. Понятие ограничителя необходимо для доопределения смысла текста. Ограничителем может быть скобка, ключевое слово языка и т.д.

Пробелы не являются разграничителями лексем в двух случаях: если они появляются внутри литерных цепочек и если они находятся внутри комментариев. Комментарием, то есть текстом не являющимся текстом программы, считается текст записанный между парой литер «(*» и парой литер «*)».

И последнее. Большие и маленькие буквы в словаре КП считаются разными.

Понятие числа

Числа в КП могут быть записаны в двух системах счисления: десятичной и шестнадцатиричной. Если число записано с суффиксом «H» или «L» то это шестнадцатиричное число иначе десятичное. Суффикс «L» предназначен для обозначения 64-х битных констант. Вещественные числа всегда содержат в своей записи десятичную точку. Число 3 будет воспринято как целое. Для обозначения вещественного числа необходимо использовать запись 3.0. Вещественное число может содержать масштабный множитель E. Тогда запись числа распадается на мантиссу и порядок. Мантисса – это последовательность цифр до масштабного множителя E. Порядок это знак и последовательность цифр после E. Если знак отсутствует, то порядок считается положительным. Число 1.2E2 читается как вещественное число 120. Для записи шестнадцатиричных чисел допустимы следующие знаки в качестве цифр: «A», «B», «C», «D», «E», «F».

Понятие идентификатора

Программу, в некотором смысле можно определить, как набор идентификаторов (имен), для каждого из которых задано описание (смысл) и область видимости (блок программы в котором идентификатором можно пользоваться).

Сообщение о языке дает следующее определение идентификатора

```
ident = (letter | "_") {letter | "_" | digit}.
letter = "A" .. "Z" | "a" .. "z" | "А".."Ц" | "Ш".."ц" | "ш".."я".
digit  = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Примечание. Данное определение безусловно работает, если вы используете среду BlackBox. Если ваша рабочая среда иная, то возможно там символы кириллицы запрещены для построения идентификаторов. Нужно заметить, что для языков программирования более типична ориентация на символы латинского алфавита.

Построим имя идентификатора пользуясь данным определением. Понятие идентификатора здесь выражается термином ident. Первая круглая скобка говорит о том, что идентификатор может начинаться с letter либо с символа подчеркивания. Далее дано определение нетерминала letter. Это может быть строчная, либо прописная буква латинского алфавита, строчная, либо прописная буква кириллицы.

Следовательно, мы можем выбрать в качестве первого символа идентификатора любую букву, либо символ подчеркивания. Далее, в определении находятся фигурные скобки, в которых записаны три альтернативы: нетерминал letter, определение которого уже рассмотрено, символ подчеркивания и нетерминал digit, представляющий собой один из цифровых символов. Таким образом после первого символа возможно многократное повторение букв, цифр и подчеркиваний. Примеры правильных идентификаторов:

```
A1, ADCD, A_123, df34_8
```

Заметим, из определения следует, что идентификатор не может начинаться с цифры. Эту возможность запрещает первое правило, четко утверждающее, что

первый символ это буква, либо подчеркивание, а использование цифр возможно только со второй позиции.

Правила объявления и видимости идентификаторов:

Идентификаторы используются для обозначения различных объектов: констант, типов, переменных, процедур. Идентификатор должен быть объявлен с указанием типа для переменных величин и для каждого идентификатора определяется область видимости, то есть блок программы в котором данным идентификатором можно пользоваться. Программный блок – это модуль, процедура, запись. Поведение идентификатора по отношению к области видимости описывают четыре правила:

Уникальность имени. Идентификатор может обозначать только один объект в данной области видимости (т.е. никакой идентификатор не может быть объявлен в блоке дважды);

Неверно:

Таблица 2.1. Примеры ошибочного объявления

VAR	VAR	VAR
A, b:INTEGER;	A:INTEGER;	A: REAL;
A:REAL;	A, B:INTEGER;	A: ARRAY 10 OF INTEGER;

Локальный характер использования. На объект можно сослаться только в его области видимости;

Неверно:

```
PROCEDURE Example1;
VAR
   A:INTEGER;
BEGIN
END Example1;
PROCEDURE Example2;
BEGIN
A:=1; (*Область видимости величины A это процедура Example1*)
END Example2;
```

Правило определения идентификаторов через другие идентификаторы. Описание типа T, содержащее ссылки на другой тип T1 могут стоять в точках, где T1 еще не известен. Но тогда описание типа T1 должно следовать далее в том же блоке, в котором локализован T;

Пример:

```
TYPE
   Mas=A;
   A=ARRAY 10 OF INTEGER;
```

В определении на момент объявления идентификатора Mas идентификатор A не описан, но описание A находится с описанием Mas в одном программном блоке, поэтому здесь нет ошибки.

Следующий пример:

```
PROCEDURE P1;
TYPE
    Mas=A;
PROCEDURE P2;
TYPE
    A=ARRAY 10 OF INTEGER;
END P2;
BEGIN
END P1;
```

Представляет собой более сложную ситуацию. Можно рассматривать два идентификатора A и Mas как объявленные в одном программном блоке – процедуре P2. С этой точки зрения правила объявления выполнены, но такая точка зрения неверна, для компилятора процедуры P1 и P2 разные программные блоки, даже несмотря на то, что P12 вложена в P1. Небольшая перестановка объявлений устраняет ошибку:

```
PROCEDURE P1;
TYPE
A=ARRAY 10 OF INTEGER;
PROCEDURE P2;
TYPE
Mas=A;
END P2;
BEGIN
END P1;
```

В таком варианте компилятор не обнаружит ошибки. Идентификаторы A и Mas по прежнему расположены в разных программных блоках, но сейчас определяющий идентификатор A по тексту предшествует определяемому Mas в объемлющей процедуре. То что процедура P1 является объемлющей момент принципиальный. Следующий пример демонстрирует почему:

```
PROCEDURE P1;
TYPE
A=ARRAY 10 OF INTEGER;
BEGIN
END P1;
PROCEDURE P2;
TYPE
Mas=A;
BEGIN
END P2;
```

Здесь описание идентификатора A по тексту предшествует описанию идентификатора Mas, но эти два описания расположены в несвязанных между собой процедурах. Поэтому такое объявление ошибочно.

Использование составных имен

Имя может указывать на единичный объект, как уже было сказано выше и может указывать на составной объект, то есть объект, состоящий из других объектов. Таких сложных конструкций в КП две: записи и объекты. Составной частью записи является поле, составное частью объекта – метод. Доступ к имени, являющимся частью сложного имени, осуществляется через точку.

Имя объемлющей структуры «.» Имя вложенной структуры
Вложенная структура также может быть составной.

Уточнение идентификатора. Объявленный идентификатор может использоваться только в том программном блоке, в котором он был определен. Самый верхний уровень в котором можно выполнить определение, это модуль. Но структура описанная в модуле может быть экспортирована за пределы модуля. Тогда в случае использования ее идентификатора за пределами определяющего модуля, идентификатор должен быть уточнен идентификатором модуля. Если, например идентификатор A экспортирован из модуля M, то за пределами модуля M обращение к нему выполняется так: M.A

Величины. Типы данных. Объявление и виды типов

В перечне основных понятий мы определили программу, как управление вычислением выражений. Выражение содержит операции, константы, константные выражения, скобки. Поэтому прежде чем приступать к разбору проблем управления (операторов языка) необходимо дать детальное представление перечисленных понятий.

Значение выражения это величина некоторого типа. Поэтому, прежде разговора о величинах, необходимо дать представление о типах.

Информация о типе формируется в КП несколькими способами. Во-первых, существуют так называемые основные типы, – это простейшие типы представляющие собой наиболее часто встречающиеся виды данных. Фактически это то, без чего невозможно написание простейшей программы: некоторые виды чисел и литеры. Из основных типов формируется два вида составных: массивы и записи из которых можно составлять сложные структуры данных. КП, также позволяет формировать объявление типа, такое объявление описывает не свойства величин, а свойства типов.

Величины, описываемые непосредственно в блоке объявлений называются статическими величинами. Память под них выделяется на этапе компиляции. Есть в КП возможность объявить динамическую величину, то есть величину описанную, но пока не существующую, до особого распоряжения программиста. Типы динамических величин определяются также, как и типы статических. Динамическая величина, это не особенный тип, это способ объявления и способ существования величины.

И наконец в КП существуют так называемые константы – величины для которых на этапе компиляции определяется не тип, а сразу значение и уже по значению компилятор принимает решение о объеме выделяемой памяти и что с этой величиной можно делать. Начнем изучение величин с констант.

Согласно сообщению о языке, описание константы связывает идентификатор с неизменяемым значением. Неизменяемость значения означает возможность определения значения на этапе компиляции. Блок описания констант начинается ключевым словом CONST.

Примеры правильного определения констант:

```
CONST
   N=5;
   A=12.45;
   C=4*N+6*A;
```

Константа C определена только в том случае, если определены константы используемые в ее описании. Порядок описания существенно важен. Следующее описание

```
CONST
   C=4*N+6*A;
   N=5;
   A=12.45;
```

с точки зрения компилятора ошибочно. При обсуждении свойств идентификаторов утверждалось, что объявления величин могут записываться непоследовательно при условии, что они записаны в одном блоке. В примере на первый взгляд именно такая ситуация, величины A, N, C описаны в одном блоке, но тем не менее это ошибка. Все же ситуация немного отлична. Правило объявления идентификаторов работает действительно именно так, как было указано, но здесь мы имеем дело не с объявлением, а определением. Отличие в том, что при определении указывается и значение величины.

Поэтому здесь работает правило определения константных выражений которое говорит, что константное выражение возможно вычислить простым просмотром (правило дано чуть ниже). А это возможно только в том случае, если к моменту компиляции все величины входящие в выражение имеют точно определенные значения.

Тип константы не указывается, определенность значения константы на этапе компиляции позволяет компилятору выделить всю необходимую информацию из вычисленного значения. Описание константы C показывает также, что значением константы может быть константное выражение. Это допустимо согласно определения константы, так как константное выражение, также имеет неизменяемое значение.

Константные выражения

В сообщении о КП дано следующее определение: *константное выражение* – это выражение, которое может быть вычислено при простом текстуальном просмотре

без фактического выполнения программы. Его операнды суть константы или предопределенные процедуры – функции, которые могут быть вычислены при компиляции.

Формальное определение

ConstantDeclaration = IdentDef "=" ConstExpression.
ConstExpression = Expression.

Примеры константных выражений уже приводились выше, поясним лишь, что предопределенные процедуры – функции это функции известные компилятору, именно поэтому их значение может быть вычислено на этапе компиляции, естественно при условии, что аргумент таких функций, также константное выражение.

Антипод понятия константа – понятие переменной. Из самого термина ясно, что переменная – это величина, чье значение может изменяться в ходе выполнения программы. Поэтому на этапе компиляции определяется тип переменной, и не определяется значение. Объявление переменой согласно сообщения о языке выглядит так:

VariableDeclaration = IdentList ":" Type.

IdentList – это список имен переменных. Поэтому одно объявление может быть применено к группе имен переменных. Далее за двоеточием следует имя типа.

Имя переменной величины – это идентификатор, поэтому правила формирования имен и возможные ограничения следует смотреть в параграфе описания идентификаторов. A объявление типа согласно сообщению о языке имеет следующий вид:

TypeDeclaration = IdentDef "=" Type.
Type = Qualident | ArrayType | RecordType | PointerType | ProcedureType.

TypeDeclaration – объявление типа
IdentDef – определяемый идентификатор
Type – тип
Qualident – уточненный идентификатор
ArrayType – тип массива
RecordType – тип записи
PointerType – Указательный тип
ProcedureType – Процедурный тип

Особенно следует обратить внимание на термин Qualident. Уточнить тип идентификатора возможно, как основным типом, так и собственным. То есть под определение Qualident попадает объявление следующего вида:

```
TYPE
    Mas=ARRAY 10 OF INTEGER;
    MyMass=Mas; (*уточненный идентификатор MyMass*)
```

Следует также обратить внимание на процедурный тип. Величины этого типа связываются не со структурами данных, а процедурами. В этом процедурный тип стоит особо от других типов данных. Необходимо также в отношении процедурного типа заметить, что он в настоящее время считается устаревшим средством и оставлен в КП только для поддержки уже разработанного ПО. Согласно сообщению, поддержка процедурных типов может быть сокращена в следующих версиях языка. Поэтому для программистов выбравших в качестве языка язык семейства Паскаль видимо следует воздерживаться от употребления этого средства при разработке нового ПО.

Описание типов переменных величин, начнем с основных. Как уже было сказано выше, основные типы предназначены для описания различных видов чисел и литер. Все остальные возможности вынесены в составные типы и собственные. Основные типы перечислены в следующей таблице.

Основные типы данных

Таблица 2.2. Список основных типов данных

Имя типа	Значения
BOOLEAN	логические значения TRUE и FALSE
SHORTCHAR	Литеры набора Latin-1 (0X .. 0FFX)
CHAR	Литеры набора Unicode (0X .. 0FFFFX)
BYTE	целые от MIN(BYTE) до MAX(BYTE)
SHORTINT	целые от MIN(SHORTINT) до MAX(SHORTINT)
INTEGER	целые от MIN(INTEGER) до MAX(INTEGER)
LONGINT	целые от MIN(LONGINT) до MAX(LONGINT)
SHORTREAL	Вещественные числа от MIN(SHORTREAL) до MAX(SHORTREAL), значение INF (INF предопределенное значение которым замещается вещественная величина в случае выхода за пределы допустимого интервала. Знак INF совпадает со знаком исходного значения)
REAL	Вещественные числа от MIN(REAL) до MAX(REAL), значение INF
SET	множества целых чисел из диапазона от 0 до MAX(SET)

Функции MIN и MAX используются для описания интервалов типов в силу того, что реальное значение границ интервалов зависит от реализации. Указанные функции используются для определения границ интервалов основных типов.

Основные типы образуют иерархию типов. Смысл иерархии в следующем: если тип A является младшим по отношению к типу B, то величина типа A может быть присвоена величине типа B. Например: Вещественная величина:=Целая величина, но никак не наоборот. Целые типы являются младшими по отношению к действительным. В КП есть две цепочки иерархии для чисел и для литер.

Иерархия числовых типов:

REAL>SHORTREAL>LONGINT>INTEGER>SHORTINT>BYTE

Иерархия литерных типов:

CHAR>SHORTCHAR

Для составных типов понятие иерархии не определено.

Составные типы. Типы массивов

Массив – структура, являющаяся упорядоченным множеством элементов одного и того же типа. Количество элементов массива называется его *длиной*. Обращение к элементам массива выполняется с помощью индексов, являющихся целыми числами из диапазона от 0 до длина – 1. В сообщении о языке дано следующее определение:

```
ArrayType   = ARRAY [Length {"," Length}] OF Type.
Length      = ConstExpression.
```

Описание длины может отсутствовать, так как длина указана в квадратных скобках. Если длина не указывается, то такой массив называется открытым. Длина открытого массива определяется в процессе работы программы, из чего следует, что открытый массив применяется только в следующих ситуациях:

- для объявления указательного типа;
- для объявления типа элемента открытого массива;
- для объявления типа формального параметра в процедуре.

Описание длины представляет собой список констант, следовательно данное описание предполагает возможность многомерных массивов. Количество размерностей правилами языка не ограничено и определяется только объемом доступной памяти.

Тип вида

```
ARRAY L0, L1, ..., Ln OF T
```

интерпретируется как сокращенная запись для

```
ARRAY L0 OF
   ARRAY L1 OF
   ...
      ARRAY Ln OF T
```

Этот вид записи можно воспринимать и как многомерный массив, элементами которого являются элементы указанного типа и как массив массивов, в качестве элементов которого можно использовать массивы меньшей размерности.

Например при следующем описании:

```
mas1, mas2: ARRAY 10,10 OF INTEGER;
```

следующее присвоение : mas1[k]:=mas[2]; будет вполне законным.

Различие между полной и сокращенной записью существенно сказывается в объявлении открытых массивов. А именно для открытого массива сокращенная запись невозможна, так как сокращенная запись требует явного указания длины для размерностей.

Примеры объявления массива:

```
mas: ARRAY 5 OF REAL; пять действительных чисел
mas: ARRAY 2, 3, 4 OF CHAR; трехмерный массив символов
mas: ARRAY 100 OF Type; одномерный массив элементов типа Type
```

Примеры открытых массивов:

```
mas: ARRAY OF INTEGER; открытый массив целых
PROCEDURE P(mas:ARRAY OF INTEGER); массив как формальный параметр
```

Составные типы. Типы записей

Тип данных запись моделирует структуру данных – множество элементов разного типа. При этом запись является неупорядоченным множеством. Доступ к компонентам записи осуществляется по имени. Компоненты записи называются полями, их количество фиксировано и для каждого поля указывается тип. Сообщение о языке дает следующее определение:

```
RecordType  = RecAttributes RECORD ["("BaseType")"]
      FieldList {";" FieldList} END.
RecAttributes          = [ABSTRACT | EXTENSIBLE | LIMITED].
BaseType    = Qualident.
FieldList   = [IdentList ":" Type].
IdentList   = IdentDef {"," IdentDef}.
```

Начнем анализ структуры записи с урезанного определения:

```
RecordType  = RECORD FieldList {";" FieldList} END.
FieldList   = [IdentList ":" Type].
IdentList   = IdentDef {"," IdentDef}.
```

Из определения видно, что объявление записи состоит из ключевых слов BEGIN и END между которыми располагается список объявлений полей.

Примеры:

```
Rec1= RECORD
    A:INTEGER;
    B: REAL;
    C: CHAR;
END;
```

В примере объявлен тип Rec1 с тремя полями.

```
Rec2:RECORD
    mas: ARRAY 10 OF INTEGER;
    N:INTEGER;
END
```

В примере объявлена величина с двумя полями, одно из которых составного типа.

Полем записи вполне может оказаться сложная структура, в том числе и структура типа записи. Например, вполне законна следующая конструкция:

```
Rec2:RECORD
    mas: ARRAY 10 OF Rec1;
    N:INTEGER;
END;
```

Существует одно ограничение. Тип не может быть своей составной частью. Об этом уже говорилось выше, здесь это ограничение проиллюстрируем примером ошибочного объявления:

```
Rec=RECORD
    Rec1:Rec;
END;
```

Такое объявление называется рекурсивным и оно является запрещенным.

Выше говорилось о том, что КП есть язык сильной именной типизации. В отношении записи это означает, что записи совместимы с точностью до одинаковых имен типов. Поэтому, например следующее объявление:

```
TYPE
    s1=RECORD
        a:INTEGER;
        b:REAL;
    END;
    s2=RECORD
        a:INTEGER;
        b:REAL;
    END;
```

есть объявление двух различных типов, несмотря на структурную идентичность и присваивание величин двух таких типов будет признано компилятором незаконным.

Вопросы доступа:

Доступ к полям осуществляется по имени поля указанному через точку после имени записи. Например:

```
Rec:RECORD
    a:INTEGER;
    b:REAL;
END;
```

Доступ: `Rec.a:=1; Rec.b:=0;`

Если поле записи представляет собой сложную структуру, то доступ к ее компонентам выполняется по правилам предписанным уже для этой сложной структуры.

Пример:

```
Rec2:RECORD
    mas: ARRAY 10 OF INTEGER;
    N:INTEGER;
END;
```

Доступ: `Rec2.mas[3]:=1; Rec2.N:=1;`

Точно таким же образом определяется доступ к компонентам полей и в том случае, если поля сами являются структурами типа RECORD. Пример:

```
Rec=RECORD
    a:INTEGER;
    b:REAL;
END;

Rec1:RECORD
Rec2:Rec;
    a:INTEGER;
    b:REAL;
END;
```

Доступ: `Rec1.Rec2.a:=1; Rec1.a:=1;`

Глубина вложенности структур может быть сколь угодно большой, техника доступа остается той же изменяется лишь длина цепочки имен.

Замечание о экспорте

Поля записи в отношении экспорта могут быть публичными (то есть видимыми за пределами описывающего модуля) и приватными (то есть доступными только в пределах описывающего модуля). Для того, чтобы поле стало публичным его необходимо пометить звездочкой. В отсутствии метки поле считается приватным.

Вернемся к полному определению записи. Тип записи используется в КП для поддержки парадигмы объектно-ориентированного программирования. Подробное изложение технологии ООП не входит в задачу этой книги, поэтому будем полагать, что вы либо с ней знакомы, либо в случае необходимости изучите ее принципы по другим источникам. Заметим только, что для понимания функциональности типа RECORD нет необходимости вникать в детали ООП.

В общих чертах технология ООП заключается в возможности связывания процедур – именуемых в этом случае методами со структурами данных. Как это исполнено в КП вы можете познакомиться в параграфе посвященном процедурам. Вторая фундаментальная возможность – это передача части структуры одного типа в структуру другого типа. В КП такой перенос выполняется на основе записей и называется наследованием.

Перенос структуры записи не обязан быть полным. Полное копирование типа записи в другой тип записи зачастую ведет к лишним затратам памяти. Новый тип чаще всего нуждается лишь в части функциональных возможностей уже имеющегося. Кроме того, сокрытие части информации о типе позволяет создать механизм

защиты информации (в данном случае полей записи). В КП вопросы доступа решаются символом «*». Поле помеченное звездочкой будет экспортировано в процессе наследования, а поле без этого символа останется скрытым. Доступные поля, в терминах ООП называются публичными, а закрытые – приватными. Тип экспортирующий поля называется предком, а тип получающий поля называется потомок.

В отношениях между типом предком и типом потомком возможны 4 варианта.

Первый вариант. Обычный тип записи без атрибутов, такой тип не предназначен для наследования. Используется только для создания структур данных.

Второй вариант. Тип пригоден для создания структур данных и наследования, но только в пределах своего модуля. Атрибут такого типа LIMITED. Для того, чтобы обеспечить ограниченность типа, его ограниченность должна быть передана потомкам.

Третий вариант. Тип пригоден для создания структур данных и наследования. Наследование возможно за пределами своего модуля. Атрибут типа EXTENSIBLE.

Четвертый вариант. Так называемый абстрактный тип. Не предназначен для создания реальных данных, используется только для наследования. Атрибут типа Abstract.

Пример расширенного наследования:

Запись предок: Rec=EXTENSIBLE RECORD a:INTEGER; END;

Здесь объявлена запись допускающая расширенное переопределение. Эту запись можно рассматривать как предка. А ниже пример возможного потомка

Запись потомок: Rec1=RECORD (Rec) b:REAL; END;

Запись потомок, имеет свое собственное поле и поле полученное от предка. Поэтому после следующего объявления величины:

```
VAR
   A:Rec1;
```

Будет допустимым следующее обращение: A.a:=0;

Пример ограниченного наследования. Запись предок:

```
s1=LIMITED RECORD
   a:INTEGER;
END;
```

Ошибочное наследование. Запись потомок:

```
 s2=RECORD (s1)
   b:REAL;
END;
```

Правильное наследование. Запись потомок:

```
s2=LIMITED RECORD (s1);
END;
```

Примечания.

- нельзя экспортировать тип записей являющихся потомком скрытого (не экспортированного типа);

- аналогично вы можете построить примеры наследования абстрактных типов. Мы же ограничимся приведенными примерами, так как детальное изучение технологии ООП не входит в наши цели.

Указательный тип

Функциональное назначение переменной указательного типа – это управление динамическими величинами, то есть величинами, создаваемыми и удаляемыми в процессе работы программы. Работа с динамической величиной состоит из двух действий:

- объявление указателя на динамическую величину;
- создание динамической величины в процессе работы программы, в необходимой точке.

Объявление указательного типа согласно сообщения о языке выполняется так:

```
PointerType = POINTER TO Type.
```

Здесь PointerType – идентификатор указательного типа. Type – тип будущей динамической величины. В КП возможными динамическими величинами могут быть только массивы и записи. Следующее объявление: A=POINTER TO INTEGER будет признано компилятором ошибочным. Компилятор КП берет на себя проверку связей между объявлениями. Поэтому следующая попытка обмана:

```
a=INTEGER;
uk=POINTER TO a;
```

будет успешно распознана компилятором. Ограничение на основные типы это жесткое, категоричное ограничение, но оно оправдано. Сложно придумать задачу, которая действительно нуждалась бы в указателях на основные типы.

Примеры правильного объявления указательных типов:

Указатель на массив:

```
mas=ARRAY 10 OF INTEGER;
A=POINTER TO mas;
```

Указатель на тип записи:

```
rec=RECORD
    a:INTEGER;
END;
B=POINTER TO rec;
```

Указатель на массив без объявления специального типа:

```
a:POINTER TO ARRAY 10 OF INTEGER;
```

Правильное объявление типа и переменной обеспечивает возможность создания динамической величины, для действительного создания величины необходимо выполнить предопределенную процедуру NEW. Результатом вызова процедуры NEW(a) будет размещение величины a в свободной памяти, памяти выделяется

столько, сколько необходимо для типа этой величины. Исключение из правила – размещение открытого массива. Так как на этапе объявления такого массива его длина не известна, то его длину необходимо сообщить процедуре NEW.

- NEW(имя массива, N) для одномерного массива
- NEW(имя массива, N1, N2, …Nm) для многомерного

Все поля и элементы размещенной динамической величины очищаются. Числовые значения инициализируются нулем, указатели значением NIL.

Базовым типом для указателя может быть запись. А записи как уже известно в КП поддерживают парадигму объектно-ориентированного программирования и могут наследовать и передавать свои компоненты. Поэтому если указатель использует в качестве базового типа запись, то становится необходимым исследовать отношение указателей к механизму наследования.

Это отношение выражается следующим правилом: если тип является потомком указательного типа, то он является и потомком типа являющегося базовым для его предка. Поясним сказанное примером:

```
PROCEDURE example;
TYPE
    P1=RECORD
    a:INTEGER;
END;
P2=POINTER TO P1;
P3=P2;
VAR
    P:P3;
BEGIN
P.a:=1;
END example;
```

Примечания:

- существует предопределенный тип ANYREC имеющий смысл – указатель на любой тип. Любой указательный тип считается его потомком;
- операция ^ является операцией разыменования. Она позволяет получить доступ к величине связанной с указателем. Если А есть указатель на величину некоторого типа, то A^ – есть значение величины. Но на практике операцией разыменования можно не пользоваться. И для работы с адресом и для работы со значением можно пользоваться идентификатором указателя без знака операции ^. Принятие решения о том, что именно имеет в виду программист осуществляется исходя из контекста.

Процедурный тип

Процедурный тип – это указатель на процедуру или предопределенное значение NIL. После присваивания переменной процедурного типа имени процедуры, процедуру можно вызывать используя процедурную переменную. Пример ниже показывает как объявлять и использовать переменные процедурного типа.

```
PROCEDURE example(x:INTEGER);
BEGIN
x:=x+1;
StdLog.Int(x);
END example;

PROCEDURE P2;
TYPE
    Function=PROCEDURE(x:INTEGER);
VAR
    a:Function;
BEGIN
a:=example;
a(7);
END P2;
```

Процедурные переменные используются только для процедур созданных программистом. Процедурная переменная не может указывать на предопределенную процедуру и не может указывать на метод. В отношении собственно процедур существует только одно ограничение – процедура должна быть описана на верхнем уровне модуля. Попытка связать переменную с локальной процедурой будет воспринято как ошибка.

Пример ошибки:

```
PROCEDURE P2;
TYPE
    Function=PROCEDURE(x:INTEGER);
VAR
    a:Function;
PROCEDURE example(x:INTEGER);
BEGIN
x:=x+1;
StdLog.Int(x);
END example;
BEGIN
a:=example;
a(7);
END P2;
```

Других ограничений нет. Связываемая с переменной процедура, может быть как собственно процедурой, так и процедурой – функцией.

Важное примечание. Еще раз напомним что: процедурный тип считается устаревшей особенностью языка и в КП введен, только для поддержки уже разработанного программного обеспечения. Предполагается в дальнейшем ограничить использование этого инструмента.

Цепочки литер

Литерная цепочка это не специальный тип, это способ хранения литер в литерном массиве. В КП для представления литерных цепочек нет предопределенных идентификаторов. Для представления цепочек используются массивы типа CHAR

и SHORTCHAR, также литерные цепочки могут храниться в строковых константах. Литерная цепочка становится чем-то отличным от массива благодаря специальному символу 0X. Этот символ является завершающим символом цепочки. Он например используется для определения длины цепочки предопределенной функцией LEN. В описании этой функции (параграф «Предопределенные функции») можно посмотреть и пример. Литерная цепочка обозначается именем литерного массива с добавлением символа $. То есть если a есть имя литерного массива, то a$ – есть имя литерной цепочки хранящейся в данном массиве.

Литерная цепочка может быть заключена в одинарные (апострофы) либо двойные кавычки. Открывающие кавычки и закрывающие должны совпадать. Открывающая кавычка не должна находится внутри цепочки. Литерная цепочка длины 1 может использоваться, как литера.

Пример:

Если a:ARRAY 100 OF CHAR; то a:='1234567890'; строковая константа

Операции

Операции можно определить, как действия выполняемые в выражениях. В КП различаются операции:

- арифметические;
- логические;
- над множествами;
- над литерными цепочками.

Операции КП можно также разделить на классы по приоритету. Приоритетом называется сила связывания. Иначе говоря приоритет это характеристика на основании которой принимается решение о том, какое действие из двух (соседствующих в выражении) должно выполняться. И по приоритету операции делятся на четыре класса:

- ~ операция отрицания;
- мультипликативные операции (операции обладающие свойствами умножения);
- аддитивные операции (операции обладающие свойствами сложения);
- отношения (операции определяющие отношения между двумя операндами, например отношения порядка: больше, меньше, равно).

Принадлежность к классу и означает описание приоритета. Поэтому операции из одного класса обладают одинаковым приоритетом. Если в выражении идут подряд несколько операций из одного класса, то порядок их выполнения определяется текстовым порядком. Операции одного приоритета выполняются слева направо. Порядок не менее важен чем приоритет. Рассмотрим следующее выражение: 2/3/5/6/7/8. Его значение самым сильным образом зависит от порядка выполнения операций.

Сообщение о языке дает следующее определение операций:

```
Expression       = SimpleExpression [Relation SimpleExpression].
SimpleExpression = ["+" | "-"] Term {AddOperator Term}.
Term             = Factor {MulOperator Factor}.
Factor           = Designator | number | character | string | NIL | Set |
                   "(" Expression ")" | "~" Factor.
Set              = "{" [Element {"," Element}] "}".
Element          = Expression [".." Expression].
Relation         = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOperator      = "+" | "-" | OR.
MulOperator      = "*" | "/" | DIV | MOD | "&".
```

Первая строчка определения SimpleExpression [Relation SimpleExpression] говорит о том, что выражение может представлять собой два простых выражения связанных операцией отношения. Например: Выражение1>Выражение2. Формирование выражения только двумя простыми – момент принципиальный. Если его убрать, то станут возможными выражения вроде следующего: 30>a>1, которое можно интерпретировать различными способами.

Подобные выражения возможно однозначным способом определять с помощью скобок. Верное скобочное выражение будет выглядеть так: (30>a) & (a>1), то здесь мы видим не одно выражение, а два соединенных знаком мультипликативной операции. То что два операнда могут разделяться аддитивными и мультипликативными операциями видно из следующих строк определения:

```
SimpleExpression = ["+" | "-"] Term {AddOperator Term}.
Term             = Factor {MulOperator Factor}.
```

Три определения:

```
Relation         = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOperator      = "+" | "-" | OR.
MulOperator      = "*" | "/" | DIV | MOD | "&".
```

Перечисляют набор разрешенных операций.

Логические операции

- OR логическое ИЛИ (логическое сложение). Двуместная операция. Операнды операции суть логические выражения, в частности переменные типа BOOLEAN. Результат операции имеет тип логический. Результат есть истина если хотя бы один из операндов истинен и ложь, если ложны оба операнда. Второй операнд вычисляется только в том случае, если первый принимает значение FALSE, в противном случае значение результата определяется по первому операнду;
- & логическое И (логическое умножение). Двуместная операция. Операнды операции суть логические выражения, в частности переменные типа BOOLEAN. Результат операции имеет тип логический. Результат есть истина

если оба операнда по значению истинны и ложь, если ложен хотя бы один операнд;

- ~ отрицание. Одноместная операция. Операнд логическое выражение, в частности переменная типа BOOLEAN. Результат есть истина, если операнд ложен, и ложь если операнд истинен.

Арифметические операции

- + сумма;
- - разность;
- * произведение;
- / вещественное частное;
- DIV операция целочисленного деления. Применимо только к целым операндам. При этом x DIV y = ENTIER(x/y). Пример использования: a:=x DIV 5;
- MOD нахождение остатка целочисленного деления. Применимо только к целым операндам.

Тип результата операций совпадает со старшим (по иерархии типов) типом одного из операндов, то есть если например, тип одного из операндов REAL, тип результата в любом случае будет REAL. Из типов LONGINT и INTEGER будет выбран LONGINT. Если типы обоих операндов по иерархии ниже INTEGER, то тип результата обязательно INTEGER. Исключение составляет операция «/». Тип ее результата всегда REAL вне зависимости от результата.

Замечание о переполнении результата: Если по выполнению операции дающей результат вещественного типа возникает состояние переполнения (значение выходит за рамки определенные для данного типа), то значение заменяется на предопределенное значение обозначаемое идентификатором inf со знаком исходного результата. Подобного же действия для целочисленных операций не определено.

Операции над множествами (тип SET)

- + объединение. Множество результат включает в себя элементы обоих множеств операндов. Пример: (1, 2) + (3, 7, 9) = (1, 2, 3, 7, 9);
- - разность. Множество результат включает в себя множество элементов левого операнда не совпадающих с элементами множества правого операнда. Пример: (1, 2, 3) – (1, 4) = (2, 3);
- * Пересечение. Множество результат состоит из элементов присутствующих в обоих множествах и правого и левого операнда.
  Пример: (1, 2, 3)*(2, 9, 3, 0) = (2, 3);
- / симметрическая разность. Множество результат представляет собой объединение двух разностей. Обозначим левый операнд – A и правый операнд – B. Тогда симметрическую разность можно выразить следующей формулой: A/B=((A-B)+(B-A)). Пример: (1, 2, 3, 4)/(2, 3, 5)=(1, 4, 5). Симметрическую разность также можно представить, как объединение за вычетом пересечения A/B=(A + B) – (A * B).

Операции над цепочками

- + конкатенация. Операция выполняется на цепочками литер. Цепочка результат содержит литеры левого операнда, за которыми следуют литеры правого операнда. Тип цепочки – результата Shortstring если оба операнда имеют тип Shortstring, иначе тип результата String.

Отношения

- = равно (отношение сравнения) возвращает истину, если операнды равны и ложь в случае неравенства;
- # неравно (отношение сравнения) возвращает истину, если операнды неравны и ложь в случае равенства;
- < меньше (отношение сравнения) возвращает истину, если левый операнд меньше правого и ложь в противном случае;
- <= меньше или равно (отношение сравнения) возвращает истину если левый операнд меньше либо равен правому и ложь в противном случае;
- > больше (отношение сравнения) возвращает истину, если левый операнд больше правого и ложь в противном случае;
- >= больше или равно (отношение сравнения) возвращает истину, если левый операнд больше либо равен правому и ложь в противном;
- IN проверка, принадлежит или нет число множеству. пример: x IN s. Здесь x – целое число, s – множество. Операция дает значение истина, если x является элементом s и значение ложь в противном случае;
- IS проверка, является ли заданная величина величиной определенного динамического типа. Операция применима только в следующих случаях: если проверяемый тип является потомком статического типа, если проверяемый тип является указателем на тип записей, или если проверяемая переменная является IN или VAR параметром типа записей в процедуре (о IN и VAR параметрах см. в параграфе о процедурах).

Все операции – отношения возвращают тип логический (BOOLEAN). Операции =, #, <, <=, >, >= применимы ко всем числовым типам, литерным и цепочкам литер. Отношения =, # также применим к указательным типам, процедурным, логическим и типу множество.

Операторы

Как уже говорилось, операторы в языке программирования обозначают действия. Основные действия это:

- присваивание значения выражения некоторой величине, ему соответствует оператор присваивания;
- исполнение повторяющегося действия. Для его реализации в КП предусмотрено 4 вида цикла;

- ветвление, то есть действие выполняемое в зависимости от истинности или ложности некоторого условия. Для его реализации в КП есть два вида условного оператора;
- для функционирования процедур, предусмотрено два оператора: RETURN для выхода из процедуры и оператора вызова процедуры;
- оператор EXIT специальный оператор для прекращения работы оператора цикла LOOP;
- конкретизация типа (оператор WITH) позволяет для некоторой операторной последовательности определить тип данных над которым данная последовательность выполняет операции.

Операторы КП делятся на два вида: элементарные и структурированные. Элементарный оператор не имеет структуры, то есть не содержит других операторов в качестве своих составных частей. Составной оператор наоборот может содержать операторные последовательности. Кроме того допускается пустой оператор обозначающий отсутствие действия.

Программу написанную на языке высокого уровня можно определить, как последовательность операторов, но из-за существования структурированных операторов, фактически говорить о единой последовательности нельзя. Некоторые ее части будут выполняться многократно, выполнение других ее частей зависит от результатов проверки определенных условий. Поэтому и нельзя утверждать, что операторная последовательность именуемая программой начиная свое выполнение с некоторого первого оператора выполнит последовательно все описанные действия в том порядке, в котором они записаны и закончит некоторым последним. Очевидно, только то, что начнет свою работу программа с точно определенного оператора.

Однако, можно утверждать, что программа состоит из операторных последовательностей, обладающих указанным свойством, то есть начинающихся точно определенным оператором, выполняющих свои операторы в том порядке в котором они записаны, ровно по одному разу и заканчивающиеся всегда одним и тем же оператором. Начнем рассмотрение операторов с понятия операторной последовательности. В сообщении о языке дано следующее определение:

StatementSequence = Statement {";" Statement}.

То есть, операторная последовательность начинается с любого оператора, затем каждый последующий оператор отделяется от предыдущего знаком «;» точка с запятой. Последовательность обязательно линейная, но ее составляющими могут быть составные операторы содержащие другие операторные последовательности. Пример:

```
FOR I:=1 TO 10 DO
    Первая операторная последовательность
END;
IF t=1 THEN
    Вторая операторная последовательность
ELSE
    Третья операторная последовательность.
END;
```

Текст содержит одну операторную последовательность, в которую вложено еще три. Операторная последовательность верхнего уровня состоит из двух операторов: оператора цикла и условного. Последовательности обозначенные, как первая, вторая и третья являются вложенными.

Условный оператор

Структурированный оператор определяющий, какая из операторных последовательностей должна выполняться. Решение принимается на основании вычисления логического выражения. Полная форма оператора согласно сообщению выглядит следующим образом:

```
IfStatement =
    IF Expression THEN StatementSequence
    {ELSIF Expression THEN StatementSequence}
    [ELSE StatementSequence]
    END.
```

Вспомним, что фигурные скобки означают повторение, возможно ноль раз, квадратные скобки представляют необязательное значение. Следовательно минимально необходимая форма следующая:

```
IfStatement =
    IF Expression THEN StatementSequence
    END.
```

Рассмотрим ее. Expression – логическое выражение (в сообщении о языке используется термин «охрана»). Полностью запись можно прочитать так: *Если истинно выражение, то выполняется операторная последовательность следующая за ключевым словом THEN.*

Следующая форма

```
IfStatement =
    IF Expression THEN StatementSequence
    ELSE StatementSequence
    END.
```

Здесь намеренно убраны квадратные скобки, так как форму без ключевого слова ELSE мы уже рассмотрели. Данная запись читается так: *Если истинно выражение то выполняется операторная последовательность следующая за ключевым словом THEN, иначе выполняется операторная последовательность следующая за ключевым словом ELSE.*

Ключевое слово ELSIF переводится как Иначе Если. То есть иначе, если выполняется условие то выполнение передается на операторную последовательность следующую за соответствующим ключевым словом THEN. Условий ELSIF может быть много, таким образом вариант ELSIF позволяет отказаться от вложенных условий следующего вида:

```
IF выражение THEN
   Операторная последовательность
ELSE
   IF выражение THEN
   ELSE
      IF выражение THEN
         Операторная последовательность
      ELSE
      END
   END;
END;
```

Заметим, что ключевое слово END в условном операторе используется единожды, для закрытия составной конструкции. Операторная последовательность ТО отделяется от операторной последовательности ИНАЧЕ ключевым словом ELSE. Эта языковая особенность КП позволяет программисту не беспокоиться о правильном завершении составной конструкции в случае изменения входящих в нее операторных последовательностей. Это утверждение верно и для других структурных операторов.

Обобщим сказанное. Условный оператор – это конструкция связывающая операторные последовательности с логическими выражениями называющимися их охранами. Для передачи выполнения соответствующей операторной последовательности вычисляются значения охран (логических выражений) в том порядке, в котором они записаны в тексте, до тех пор пока какая-либо охрана не даст значения ИСТИНА. После чего выполнение передается последовательности соответствующей данной охране. Если ни одна охрана не дала истинного значения, то выполнение передается операторной последовательности записанной после ключевого слова ELSE. Если ELSE отсутствует, то выполнение передается на первый оператор следующий за END закрывающим условную структуру.

Оператор выбора

Оператор выбора также, как и условный управляет передачей выполнения на операторные последовательности в зависимости от результатов вычисления некоторого выражения. Вычисляемое выражение должно давать значение целого или литерного типа. Вычисленный результат ищется среди множества меток, каждая из которых связана с некоторой операторной последовательностью. При первом же совпадении выполняется операторная последовательность связанная с найденной меткой. В сообщении о языке дано следующее определение:

```
CaseStatement     = CASE Expression OF Case {"|" Case}
                    [ELSE StatementSequence] END.
Case              = [CaseLabelList ":" StatementSequence].
CaseLabelList     = CaseLabels {"," CaseLabels}.
CaseLabels        = ConstExpression [".." ConstExpression].
```

- Expression – выражение целого или литерного типа
- Case – альтернатива

- CaseLabelList – список меток
- CaseLabels – метка
- StatementSequence – операторная последовательность
- ConstExpression – константное выражение

Заметим, что даже простейшая форма оператора требует хотя бы одной альтернативы. Оператор вида

```
CASE k OF
END;
```

Приведет к ошибке во время исполнения. Фигурные скобки указывают на возможность множества альтернатив, разделенных знаком «|». Пример:

```
CASE k OF
    1:a:=1;
    | 2:a:=2;
    | 3:a:=3;
END;
```

1, 2, 3 в примере – это метки после которых, записываются исполняемые операторные последовательности. В нашем примере каждая последовательность состоит из одного оператора, что конечно же не обязательно. Каждая альтернатива примера содержит лишь одну метку. Это также не обязательное ограничение, как видно из определения, каждая альтернатива определяется множеством меток. При этом множество меток может задаваться двумя способами: интервалом значений и перечислением.

Пример перечисления меток:

```
CASE k OF
    1,2,3:a:=1;
    |4,5,6:a:=2;
END;
```

Пример интервала меток:

```
CASE k OF
    1..3:a:=1;
    |4,5,6:a:=2;
END;
```

Вторая альтернатива, как и в предыдущем примере построена перечислением, первая альтернатива определена интервалом значений. Интервалы, также могут включаться в перечисления, как и элементарные метки. Пример:

```
CASE k OF
    1..3, 9:a:=1;
    |4,5,6:a:=2;
END;
```

Полная форма оператора содержит лексему ELSE после которой записывается операторная последовательность исполняемая в случае, если значение вычисленного выражения не было обнаружено ни в одном из множеств меток. Пример:

```
CASE k OF
    1..3, 9:a:=1;
    |4,5,6:a:=2;
    ELSE a:=3;
END;
```

Важная особенность. Конструкция CASE обязательно должна выполнить одну из вложенных операторных последовательностей. Если этого не произойдет среда программирования аварийно прервет выполнение программы и сообщит о состоянии ошибки. Это означает, что обязательно выполнение одного из двух условий:

- Значение вычисленного выражения переключателя обнаружено в одном из множеств меток.
- Значение не обнаружено, но в структуре оператора присутствует лексема ELSE.

Множества меток не должны пересекаться. Если хотя бы одна метка окажется элементом сразу двух множеств это будет расценено как ошибка. И эта ошибка будет обнаружена на этапе компиляции.

Метка является константным выражением, что следует из определения:

CaseLabels = ConstExpression [".." ConstExpression].

Попытка указать в качестве метки переменную величину, также будет расценена как ошибка на этапе компиляции.

Циклические конструкции КП

Циклические конструкции управляют многократным исполнением операторных последовательностей. В таких структурах следует различать операторные последовательности называемые телом цикла и конструкции описывающие собственно цикл.

Компонентный Паскаль предоставляет программисту четыре варианта циклических конструкций: два условных, один с шагом и один безусловный. Две условных формы это: цикл с условием продолжения и цикл с условием завершения, их работа или наоборот остановка определяется некоторым логическим выражением (охраной в терминах сообщения). Поведение цикла с шагом определяется некоторой целой величиной называемой шагом или иногда параметром цикла, изменяющейся с фиксированным шагом. Безусловный цикл не предусматривает остановки, для его прерывания нужно выполнить оператор прерывания.

Условные циклы КП обладают хорошей универсальностью. Принципиально одной формы условного цикла достаточно. Все остальные формы можно рассматривать лишь с точки зрения удобства. Начнем рассмотрение с цикла с условием продолжения.

Цикл с условием продолжения (WHILE)

Цикл с условием продолжения проверяет значение своей охраны до выполнения операторной последовательности. Отсюда его название. Кроме того, условие (охрана) цикла есть условие продолжения работы, то есть операторная

последовательность являющаяся телом цикла выполняется до тех пор, пока условие истинно. Ложность условия приводит к прекращению работы. В сообщении о языке дано следующее определение:

WhileStatement = WHILE Expression DO StatementSequence END.

Согласно определению, собственно конструкция цикла состоит из трех лексем: WHILE, DO, END. Между лексемами WHILE и DO записывается логическое выражение являющееся охраной цикла и между лексемами DO и END записывается операторная последовательность.

Примеры:

```
sum:=0; k:=1;
WHILE k<=10 DO
    sum:=sum+k;
    k:=k+1;
END;

sum:=0; k:=0; flag:=TRUE;
WHILE flag DO
    sum:=sum+a[k];
    IF sum>max THEN
    flag:=FALSE;
    END;
    k:=k+1;
END;
```

Следует заметить, что тело цикла с условием продолжения возможно не будет выполнено ни одного разу, если перед вхождением в цикл, условие окажется ложным. Охрана принципиально может не иметь никакого отношения к телу цикла, то есть величины участвующие в формировании охраны вполне возможно не будут вычисляться в теле цикла. Такая ситуация не является ошибкой с точки зрения компилятора, но необходимо понимать, что в этом случае нет причин для изменения значения охраны и следовательно нет причин и для прекращения выполнения цикла. Это означает, что цикл с условием продолжения может впасть в бесконечное выполнение (зависнуть). Поэтому отсутствие изменений величин формирующих охрану в операторной последовательности тела цикла следует считать семантической (смысловой) ошибкой.

Цикл с условием завершения (REPEAT UNTIL)

Цикл с условием завершения проверяет значение своей охраны после выполнения операторной последовательности и для этой формы охрана есть условие завершения цикла. В отличие от цикла WHILE, цикл с условием завершения завершает работу тогда, когда охрана дает значение истины. В сообщении о языке дано следующее определение:

RepeatStatement = REPEAT StatementSequence UNTIL Expression.

Циклическая конструкция состоит из двух лексем REPEAT и UNTIL между которыми записывается операторная последовательность являющаяся телом цикла. Данная структура не нуждается в лексеме END, телом цикла считается операторная последовательность записанная между лексемами REPEAT и UNTIL. Охрана цикла записывается после лексемы UNTIL.

Примеры:

```
sum:=0;k:=1;
REPEAT
    sum:=sum+k;
    k:=k+1;
UNTIL k>10;

sum:=0;k:=0;flag:=FALSE;
REPEAT
    sum:=sum+a[k];
    k:=k+1;
    IF sum>max THEN
    Flag:=TRUE;
    END;
UNTIL flag;
```

Так как цикл с условием завершения вычисляет охрану после исполнения тела, то очевидно тело цикла будет гарантированно выполнено хотя бы один раз. Отсюда видна полезность цикла. Если величины формирующие охрану выгодно вычислять в теле цикла, в силу чего тело должно быть обязательно выполнено, то цикл с условием завершения может дать небольшую экономию операторов.

Для цикла с условием завершения справедливо все что было сказано в отношении цикла с условием продолжения о возможности смысловых ошибок.

Цикл с шагом (FOR TO BY DO END)

Для цикла с шагом отсутствует понятие охраны. Его поведение определяет некоторая управляющая переменная именуемая шагом. Для шага описываются начальное и конечное значения и значение изменения шага. Цикл завершает свою работу, тогда когда управляющая переменная достигает своего конечного значения. В сообщении о языке дано следующее определение:

ForStatement =
FOR ident ":=" Expression TO Expression [BY ConstExpression]
DO StatementSequence END.

Из определения видно, что цикл с шагом наиболее сложная циклическая конструкция, это ясно хотя бы из количества лексем (пять лексем) описывающих цикл. Между лексемами FOR и TO описывается начальное значение управляющей переменной. После лексемы TO записывается конечное значение и после лексемы BY записывается шаг являющийся не нулевым константным выражением. Из определения видно, что шаг является необязательной составляющей, если шаг

не указан, то по умолчанию он считается равным единице. И между лексемами DO и END записывается исполняемая операторная последовательность.

Пример:

```
sum:=0;
FOR k:=1 TO 10 DO
    sum:=sum+k;
END;
```

Фактически роль управляющей переменной сводится к подсчету количества исполняемых циклом шагов. Она может участвовать в вычислениях операторной последовательности тела цикла. Нет ничего страшного если ее значение используется для вычисления других величин. Можно изменять и ее значение, но необходимо помнить, что в этом случае будет изменена логика работы всего цикла, поэтому к любым изменениям управляющей переменной в теле цикла необходимо подходить крайне осторожно.

Выражения, используемые для вычисления начального и конечного значения шага, вычисляются только один раз при вхождении в цикл. Если в расчете граничных значений шага участвуют какие-либо переменные, то эти переменные могут участвовать в операторной последовательности тела цикла в том числе изменять свое значение, однако такие изменения никак не повлияют на поведение цикла.

Пример:

```
sum:=0;a:=0;
FOR k:=1+a*a TO 10 DO
    sum:=sum+k;
    a:=a+1;
END;
```

Можно было бы ожидать, что величина «a» увеличиваясь уменьшит количество шагов выполняемых циклом, однако этого не произойдет. Данное свойство цикла дает возможность например использовать одну и ту же переменную в разных целях.

Пример:

```
sum:=0;k:=10;
FOR k:=1 TO k DO
    sum:=sum+k;
END;
```

В примере переменная k является управляющей переменной цикла и в то же время она задает свое конечное значение. Но тем не менее пример будет работать правильно, так как граничные значения вычисляются до начала работы тела цикла. Но в общем-то такой программный фрагмент производит плохое впечатление.

Безусловный цикл (LOOP)

Безусловный цикл, пожалуй наиболее простая форма цикла. Его поведение не определяется ничем. Прекращение выполнения самой формой цикла не предусмотрено. В сообщении о языке дано следующее определение:

LoopStatement = LOOP StatementSequence END.

Структура цикла описывается лишь двумя лексемами LOOP и END между которыми записывается операторная последовательность являющаяся телом цикла. В силу своей простоты оператор не имеет каких-либо заметных и интересных особенностей. Вся ответственность по прекращению его работы возлагается на программиста.

Примеры:

```
sum:=0;k:=1;
LOOP
    sum:=sum+k;
    k:=k+1;
    IF k>10 THEN
    EXIT;
    END;
END;

sum:=0;k:=0;
LOOP
    sum:=sum+a[k];
    k:=k+1;
    IF sum>max THEN
    EXIT;
    END;
END;
```

В заключение

В качестве заключения отметим, что все описанные выше формы циклических конструкций в значительно степени взаимозаменяемы. Взаимозаменяемость можно было бы считать полной, если бы не цикл с шагом. Форма цикла с шагом требует от программиста указать точное количество выполняемых действий, что не всегда возможно.

Оператор конкретизации типа WITH

Оператор устанавливает соответствие переменной некоторому типу, если соответствие имеет место, то выполняется операторная последовательность. WITH позволяет в одном операторе записать несколько проверок, вместе с соответствующими им операторными последовательностями. Для учета возможной ложности всех проверок, записывается операторная последовательность ELSE. В сообщении о языке дано следующее определение:

```
WithStatement   = WITH [ Guard DO StatementSequence ]
                  {"|" [ Guard DO StatementSequence} ]
                  [ELSE StatementSequence] END.
Guard           = Qualident ":" Qualident.
```

Среда программирования предполагает, что одна из записанных операторных последовательностей обязательно будет выполнена, иное считается ошибкой. Следовательно для WITH обязательно выполнение одного из двух условий:

- одна из проверок обязательно даст значение Истина;
- все проверки ложны, но есть возможность выполнить операторную последовательность после лексемы ELSE.

Пример использования:

```
PROCEDURE P1;
TYPE
    mas=RECORD
    END;
    mas1 = POINTER TO mas;
VAR
    a:POINTER TO mas;
BEGIN
WITH a:mas1 DO
END;
END P1;
```

Оператор возврата RETURN

Оператор возврата используется для прерывания работы процедуры и возврата значения если это процедура – функция. Оператор состоит из одной лексемы RETURN после которой указывается возвращаемое значение. Это может быть, как имя переменной, так и выражение.

Пример:

```
PROCEDURE Example(a:INTEGER):INTEGER;
BEGIN
RETURN a*a;
END Example;
```

В описании процедуры может быть несколько операторов RETURN. Завершение работы процедуры происходит по передаче выполнения на любой их них. Оператор возврата можно использовать, как в процедурах – функциях, так и обычных процедурах. В случае обычных процедур RETURN не должен содержать возвращаемое значение.

Ошибка:

```
PROCEDURE P1;
BEGIN
RETURN 1;
END P1;
```

Правильно:

```
PROCEDURE P1;
BEGIN
RETURN;
END P1;
```

Оператор прерывания EXIT

Оператор необходим для прерывания исполнения операторной последовательности в операторе LOOP. Обозначается одной лексемой EXIT. Его выполнение приводит к завершению охватывающего данную операторную последовательность цикла LOOP и передаче управления на оператор следующий за лексемой END завершающий данный LOOP. EXIT прерывает выполнение только того оператора LOOP в котором он записан. В случае вложенных LOOP объемлющие циклы не прерываются. Если EXIT находится внутри составного оператора, то это никак не мешает ему выполнить свою работу. Пример ниже:

```
LOOP
    IF 1=1 THEN
        EXIT;
    END;
END;
```

Синтаксически EXIT не связан с LOOP. Но его использование за пределами LOOP определяется компилятором, как ошибка. Поэтому прервать с помощью EXIT цикл или выполнение процедуры нельзя.

Процедуры. Описание

Главный смысл и цель любой процедуры – это выделить в отдельный логически замкнутый фрагмент последовательность действий. Это может быть полезно для:

- поддержки процесса декомпозиции задачи (разбиению ее на подзадачи);
- для минимизации текста программы (один раз пишем, много раз вызываем).

По функциональному назначению в КП выделяется три варианта процедур:

Обычная процедура. Обычная процедура может получать на вход формальные параметры, ее можно вызывать без параметров, завершение работы может быть выполнено по передаче управления на соответствующий END, а также и по выполнению оператора RETURN без возвращаемых значений. Вызов процедуры не может быть составной частью сложного выражения.

Процедура – функция. В отличии от обычной процедуры должна иметь список формальных параметров, но этот список может быть пустым. В случае пустого списка формальных параметров, список фактических также должен быть пустым. Общее правило гласит, что список формальных параметров должен соответствовать списку фактических. Сколько формальных в описании, столько и фактических в вызове. Типы соответствующих фактического и формального параметра должны быть совместимыми. Завершение процедуры – функции происходит только по выполнению оператора RETURN. Его отсутствие означает для компилятора ошибку.

Процедура – метод. Процедура связанная со структурами данным. Процедура – метод в КП предназначена для поддержки объектно-ориентированного программирования. «Свои» структуры данных отличаются от прочих тем, что процедура –

метод не нуждается в их получении, процедура имеет доступ к своим структурам без специальных усилий.

Сообщение о языке дает следующее общее определение описания процедуры:

```
ProcedureDeclaration    = ProcedureHeading [";" ProcedureBody ident ].
ProcedureHeading        = PROCEDURE [Receiver] IdentDef
                          [FormalParameters] MethAttributes.
ProcedureBody           = DeclarationSequence
                          [BEGIN StatementSequence] END.
DeclarationSequence     = {CONST {ConstantDeclaration ";"} |
                          TYPE {TypeDeclaration ";"} |
                           VAR {VariableDeclaration ";"} }
                          {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
ForwardDeclaration      = PROCEDURE " ^ " [Receiver] IdentDef
                          [FormalParameters] MethAttributes.
```

Из определения видно что описание процедуры содержит следующую общую информацию:

- имя процедуры (необходимо для вызова);
- информация о необходимых внешних данных;
- информация о возвращаемых значениях;
- список объявлений типов, величин, процедур;
- описание вида (для методов);
- тело процедуры – последовательность исполняемых операторов.

Определение ForwardDeclaration= PROCEDURE " ^ " [Receiver] IdentDef описывает важный языковый нюанс. Имя процедуры это идентификатор на который распространяются правила видимости идентификаторов. Отвлекитесь от данного пункта на минуту, прочитайте их еще раз внимательно. Из этих правил следует, что следующее описание компилятор воспримет как ошибочное.

```
PROCEDURE P1 ( x: INTEGER );
BEGIN
P2( x )
END P1;

PROCEDURE P2 ( y: INTEGER );
BEGIN
(* операторная последовательность *)
END P2;
```

Причина в том, что к моменту компиляции процедуры P1 идентификатор P2еще не известен, а для компиляции все идентификаторы должны быть объявлены, до того, как они понадобятся компилятору.

Данная проблема легко поправима. Компилятору нет необходимости иметь полную информацию о P2. достаточно сообщить только заголовочную информацию. Именно на такую возможность и указывает определение ForwardDeclaration.

Пример выше можно переписать следующим образом:

```
PROCEDURE^ P2 ( y: INTEGER );

PROCEDURE P1 ( x: INTEGER );
BEGIN
P2( x )
END P1;

PROCEDURE P2 ( y: INTEGER );
BEGIN
(* операторная последовательность *)
END P2;
```

Исправленный фрагмент уже не содержит ошибок с точки зрения компилятора.

Формальные параметры

Назначение формальных параметров заключается в передаче процедуре информации необходимой для ее успешной работы. Таковая информация представляет собой список величин являющихся либо параметрами – значениями, либо параметрами переменными. Параметры – значения являются локальными переменными для процедуры в заголовке которой они описаны. Свои значения они получают присвоением соответствующих фактических параметров. Параметры – переменные соответствуют фактическим, это означает, что имя фактического параметра и формального есть два имени одной области памяти и все изменения выполняемые над параметром переменным, происходят и с фактическим. Иначе говоря параметр – переменная и соответствующий ей фактический параметр это одна и та же переменная. Параметр – переменную можно пометить одним из трех описателей:

- IN – параметры предназначены только для ввода, они доступны только на чтение и могут использоваться только для массивов или записей;
- OUT – параметры используются только для вывода;
- VAR – параметры используются как для ввода, так и для вывода.

OUT и VAR – параметры реализуют собой механизм благодаря которому процедуры не являющиеся процедурами – функциями могут фактически возвращать значения в вызывающую процедуру. Пример:

```
TYPE
   mas=ARRAY 10 OF INTEGER;
PROCEDURE P1(OUT a:mas; IN b:mas);
BEGIN
a:=b;
END P1;
PROCEDURE P2;
VAR
   c,d:mas;
BEGIN
P1(c,d);
END P2;
```

Процедура P2 использует в качестве фактических параметров два массива с именами c и d. Массив c предназначен для получения результата работы процедуры P1. Массив d массив аргумент для P1. Процедура P1, выполнив присваивание возвращает массив a в точку вызова. Заметим, что какие-либо операции над массивом b в P1 запрещены. Если в P1 описатель IN заменить на VAR, то операции над b станут возможны.

Описатель VAR имеет существенное отличие от описателя IN. VAR допускает параметры простых типов. Но параметр – переменная все равно остается переменной. Например, следующий фрагмент содержит ошибку:

```
PROCEDURE P1(VAR a:INTEGER);
BEGIN
END P1;
PROCEDURE P2;
BEGIN
P1(2);
END P2;
```

Ошибка заключается в следующем. Описатель VAR создает возможность как для ввода данных, так и для вывода. Это предполагает возможность изменения величины – фактического параметра, что в данном примере невозможно, фактический параметр константа, изменить значение которой невозможно. Поэтому, вызов P1 должен содержать имя переменной.

Сообщение о языке дает следующее определение списку формальных параметров:

```
FormalParameters  = "(" [FPSection {";" FPSection}] ")" [":" Type].
FPSection         = [VAR | IN | OUT] ident {"," ident} ":" Type.
```

Структура списка формальных параметров наверное уже ясна из примеров. Определение добавляет только то, что список может состоять из нескольких секций, для каждой из которых возможен один из описателей параметров – переменных. Описатели, согласно определению могут отсутствовать, в этом случае речь идет о параметрах – значениях. Список секций заключен в квадратные скобки, это означает, что формальные параметры в заголовке не обязательны.

Процедуры. Вызов

Вызов процедуры выполняется по имени. Активизация процедуры может быть выполнена только в программном блоке представляющем область видимости для идентификатора процедуры (вспомните правила видимости идентификаторов). Если описание процедуры содержит список формальных параметров, то вызов обязан содержать список соответствующих фактических, замещающих формальные. Естественно при замещении параметров должны выполняться все правила соответствия типов.

Модули

Модуль это синтаксическая единица языка предназначенная для хранения описаний процедур и структур данных. По умолчанию все описания модуля доступны только внутри него, поэтому можно сказать, что модуль это средство охраны от несанкционированного доступа. Если разработчик модуля желает некоторые описания (как процедур, так и структур данных) сделать доступными для других модулей или среды программирования он должен эти описания экспортировать. Процедура экспорта в КП максимально проста – экспортируемые определения просто отмечаются звездочкой.

Модуль является единицей компиляции. Если разрабатываемая программа достаточно велика и состоит из нескольких модулей, то раздельная компиляция дает возможность переделывать отдельные модули, не затрагивая остальных. После переписывания текста модуля он компилируется и этот процесс никак не влияет на другие части создаваемой системы.

Сообщение о языке дает следующее определение модуля:

```
Module          = MODULE ident ";" [ImportList] DeclarationSequence
                  [BEGIN StatementSequence]
                  [CLOSE StatementSequence] END ident ".".
ImportList      = IMPORT Import {"," Import} ";".
Import          = [ident ":="] ident.
```

Описание модуля начинается ключевым словом MODULE и обязательно идентифицируется уникальным именем (лексема ident). Согласно определению модуль может иметь (но не обязательно) список импорта состоящий из идентификаторов импортируемых модулей. Модуль имеет два необязательных списка исполняемых операторных последовательностей:

- операторная последовательность после ключевого слова BEGIN исполняется после загрузки модуля, вне зависимости от того, будут или нет использоваться его процедуры. Эту операторную последовательность можно использовать для инициализации переменных модуля;
- операторная последовательность после ключевого слова CLOSE исполняется после удаления модуля из системы.

То, что в определении указано, как DeclarationSequence есть перечень объявлений модуля. Объявление – это все, возможные объявления языка КП. То есть объявления типов, переменных, констант, процедур. В неформальном введении приведено достаточно много примеров описания модулей. Единственно мы здесь проиллюстрируем примером модуль с использованием операторной последовательности исполняемой в момент загрузки:

```
MODULE Example;
IMPORT In, StdLog;
VAR
   N:INTEGER;
```

```
PROCEDURE Calc*;
BEGIN
StdLog.Int(N);
END Calc;
BEGIN
N:=10;
END Example.
```

Здесь загрузочная операторная последовательность используется для инициализации глобальной переменной N.

Полный список предопределенных процедур

Имя функции: ABS(x)
Аргумент: Число любого типа
Результат: Число. Для всех целых, кроме LONGINT тип результата есть INTEGER. Для аргументов LONGINT и REAL, тип результата совпадает с типом аргумента.
Назначение: Вычисление модуля величины

Имя функции: ASH(x, y)
Аргумент: оба аргумента – целые числа
Результат: Число. Для всех целых, кроме LONGINT тип результата есть INTEGER. Для аргументов LONGINT, тип результата LONGINT
Назначение: Выполнение арифметического сдвига в двоичной системе счисления
Пример: a:=ASH(11,3);
$11_{10} = 1011_2$ после сдвига на три разряда 1011000. Переведем обратно в десятичную систему $1011000_2 = 88_{10}$

Имя функции: CAP(x)
Аргумент: литера из набора Latin1
Результат: литера из набора Latin1
Назначение: Преобразование строчной буквы в заглавную

Имя функции: CHR(x)
Аргумент: целое число
Результат: литера
Назначение: получение литеры порядковый номер которой есть аргумент функции
Пример: Значение CHR(83) есть большая литера S

Имя функции: ENTIER(x)
Аргумента: Число вещественного типа
Результат: Число типа LONGINT
Назначение: вычисление наибольшего целого, не превосходящего x

---

Имя функции: LEN(v)
Аргумент: v – массив содержащий цепочку литер;
Результат: целое число
Назначение: длина литерной цепочки содержащейся в массиве.
Пример:

```
PROCEDURE P2;
VAR
    a: ARRAY 20 OF CHAR;
BEGIN
a:= "1234567890";
a[5] := 0X;
StdLog.Int(LEN(a$)); (*Длина литерной цепочки*)
StdLog.Int(LEN(a)); (*Длина массива*)
END P2;
```

———————————————————————————————————————

Имя функции: LONG(x)
Аргумент: Величина любого основного типа
Результат: Величина основного типа
Назначение: Изменение типа аргумента в сторону удлинения. Таблица ниже показывает пары замен типов

```
BYTE   SHORTINT
SHORTINT    INTEGER
INTEGER  LONGINT
SHORTREAL  REAL
SHORTCHAR CHAR
Shortstring   String
```

———————————————————————————————————————

Имя функции: MAX(T)
Аргумент: Имя основного типа
Результат: Если основной тип не множество, то максимальное значение для этого типа, для множества – максимальный элемент множества. А так как множество в КП это множество целых чисел, то тип результата для множества это обязательно INTEGER.
Назначение: Определение верхней границы значений для типа .

———————————————————————————————————————

Имя функции: MAX(x, y)
Аргументы: величины любого основного типа, за исключением типа SET
Результат: Если аргумент целый тип, но не LONGINT тип результата – INTEGER. При действительном типе аргумента тип результата совпадает с типом аргумента. При литерном типе аргумента тип результата также совпадает с типом аргумента.
Назначение: Определение большего значения из двух аргументов. В случае величин литерного типа, большей считается величина имеющая больший код.

———————————————————————————————————————

Имя функции: MIN(T)
Аргумент: Имя основного типа
Результат: Если основной тип не множество, то минимальное значение для этого типа, для множества – минимальный элемент множества. А так как множество

в КП это множество целых чисел, то тип результата для множества – обязательно INTEGER.
Назначение: Определение нижней границы значений для типа.

Имя функции: MIN(x, y)
Аргументы: величины любого основного типа, за исключением типа SET
Результат: Если аргумент целый тип, но не LONGINT тип результата – INTEGER. При действительном типе аргумента тип результата совпадает с типом аргумента. При литерном типе аргумента тип результата также совпадает с типом аргумента.
Назначение: Определение меньшего значения из двух аргументов. В случае величин литерного типа, меньшей считается величина имеющая меньший код.

Имя функции: ODD(x)
Аргумент: величина любого целого типа
Результат: величина логического типа
Назначение: Сравнение по модулю 2 с единицей
Примеры:
ODD(2) имеет значение FALSE; ODD(3) имеет значение TRUE

Имя функции: ORD(x)
Аргумент: величина символьного типа или типа множество
Результат: Если аргумент типа CHAR, то тип результата INTEGER. Если тип аргумента SHORTCHAR то тип результата SHORTINT. Для аргумента типа SET (множество) тип результата INTEGER
Назначение: порядковый номер литеры

Имя функции: SHORT(x)
Аргумент: величина любого основного типа
Результат: величина основного типа
Назначение: преобразование типа величины в сторону уменьшения. В таблице ниже показано соответствие пар типов:

LONGINT	INTEGER
INTEGER	SHORTINT
SHORTINT	BYTE
REAL	SHORTREAL
CHAR	SHORTCHAR
String	Shortstring

Имя функции: SIZE(T)
Аргумент: переменная любого типа
Результат: величина типа INTEGER
Назначение: Количество байт требуемое для переменной данного типа
Примечание: SIZE не может применяться в константных выражениях, т.к. его значение зависит от фактической реализации компилятора.

Имя процедуры: ASSERT(x)
Аргумент: логическое выражение
Назначение: остановка работы программы, если аргумент ложен

Имя процедуры: ASSERT(x, n)
Аргумент: x – логическое выражение; n – целое
Назначение: остановка работы программы, если аргумент ложен
Примечание: значение n – определяется реализацией.

Имя процедуры: DEC(v)
Аргумент: Переменная целого типа
Назначение: уменьшение значения аргумента на единицу

Имя процедуры: DEC(v, n)
Аргумент: оба аргумента целые числа
Назначение: уменьшение значения аргумента v на значение аргумента n

---

Имя процедуры: EXCL(v, x)
Аргументы: v – множество, x – целое число
Назначение: удалить целое число x из множества v
Примечание: интервал допустимых значений для x: 0 <= x <= MAX(SET)

Имя процедуры: HALT(n)
Аргумент: целая константа
Назначение: остановить программу
Примечание: значение константы n определяется реализацией.

Имя процедуры: INC(v)
Аргумент: величина целого типа
Назначение: увеличение значения аргумента на единицу

Имя процедуры: INC(v, n)
Аргумент: оба аргумента величины целого типа
Назначение: увеличение значения аргумента v на значение аргумента n

---

Имя процедуры: INCL(v, x)
Аргументы: v – множество, x – целое число
Назначение: добавить целое число x ко множеству v
Примечание: интервал допустимых значений для x: 0 <= x <= MAX(SET)

Имя процедуры: NEW(v)
Аргумент: указатель на запись или фиксированный массив
Назначение: Размещение указанного типа величины в памяти

Имя процедуры: NEW(v, x0, ..., xn)
Аргумент: v – указатель на открытый массив, остальные аргументы величины целого типа
Назначение: Размещение открытого массива с длинами x0.. xn

Глава 3

Практикум

Данный раздел преследует две цели: практика в разработке и реализации алгоритмов и приобретение представления о методах и приемах алгоритмизации. Вторая цель не ставится во главу угла, такая цель достойна отдельного учебника, поэтому выбор задач не подчинен задаче хорошего усвоения того или иного метода. Подбор задач несколько хаотичен, но все задачи достаточно интересны и не вполне тривиальны в реализации.

Ни одну задачу вам не придется решать с нуля. Для каждой описана идея решения, но описаны идеи с разной степенью детальности. Иногда достаточно подумать над написанием программы, иногда будет необходимо немного доработать идею, иногда над идеей придется поработать более тщательно.

Здесь нет ни одной до конца решенной задачи, более того, почти нет строгого описания алгоритмов, хотя бы на псевдокоде. Это сделано специально, для тренировки перехода от неполного описания к строгим алгоритмическим формулировкам. Единственное исключение в отношении описания сделано для некоторых задач в разделе «Графы». Эти несколько задач представляют собой требование реализации стандартных и очень важных алгоритмов, для них приведена реализация на псевдокоде. Но это не слишком облегчит вашу работу, так как описание алгоритмов приведено в терминологии достаточно далекой о того, что может представить в распоряжение программиста язык программирования. Для этих же задач приведены очень детально проработанные примеры, что делает их описание очень объемным.

Начинается практикум разделом «Разные задачи». Это задачи, которые сложно как-то однозначно классифицировать их решение с пустого места может потребовать самых различных знаний, а самое главное хорошей способности к исследованию решений.

Задачи, которые можно отнести к следующему разделу «Сортировки» уже решались в неформальном введении. Различие заключается в том, что методы сортировки используемые здесь существенно сложнее и соответственно эффективнее методов упоминаемых в неформальном введении.

Два следующих раздела «Задачи перебора» и «Графы» в значительной степени пересекаются. Например, задачи о расстановке ферзей и поиске путей для коня можно отнести и к разделу «Графы», так как их решение можно рассматривать, как процесс построения некоторого дерева. Но они все же включены в раздел переборных задач. В целом задача раздела показать природу переборных задач и некоторые возможности оптимизации алгоритмов полного перебора.

И наконец последний раздел «Графы» по большей части состоит из задач представляющих собой формулировки уже известных алгоритмов, но есть несколько задач решаемых с использованием графов, их можно назвать прикладными по отношению к теории графов. И последнее, так как за термином «граф» стоит очень серьезная и развитая теория, то рекомендуется прежде чем решать задачи раздела познакомится как минимум с основами этой теории по специальному пособию или хотя бы по приложению к данной книге.

Если вы тщательно проработаете все задачи практикума, то во-первых, получите неплохой навык решения логически сложных задач, а во-вторых, получите представление о математических методах используемых в программировании и это сможет стать основой для дальнейшего уже систематического изучения математических методов.

Раздел А. Разные задачи

Задача 1. Задача Дейкстры.
Вычислить функцию заданную следующими условиями:

- F(1) = 1;
- F(2N) = F(N);
- F(2N + 1) = F(N) + F(N + 1);
- запрещается использовать массивы и рекурсию.

Идея решения:
Возможность использования массивов или рекурсии значительно упростили бы решение задачи (листинг 90 неформального введения содержит решение с использованием рекурсии), но они запрещены. Это означает необходимость поиска хорошей математической закономерности. Условия задачи говорят о том, что:

- непосредственно вычислить функцию можно только от единицы;
- если у функции четный аргумент то функция выражается через функцию с вдвое меньшим аргументом;
- если у функции нечетный аргумент, то функция распадается на сумму двух функций, таких что их аргументы различаются на 1 и в сумме дают исходный.

Начнем поиск закономерности с расчетного примера. Вычислим F(113). Промежуточные результаты запишем в виде последовательности чисел:

F(113) = F(56) + F(57) = F(28) + (F(28) + F(29)) и далее

1. 113
2. 56, 57
3. 28, 28, 29
4. 14, 14, 15
5. 7, 7, 7, 7, 8
6. 3, 4, 3, 4, 3, 4, 3, 4, 4
7. 2, 1, 2, 2, 1, 2, 2, 1, 2, 2, 1, 2, 2
8. 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1

Итого F(113) = 13

Заметим, что на каждом шаге расчетов получается только два числа, одно четное и одно нечетное. Поэтому есть смысл поискать закономерность между количествами четных и нечетных чисел. Запишем эти же результаты в виде таблицы:

Таблица 3.1. Промежуточные данные

Шаг	Четных	Нечетных
1	1	1
2	2	1
3	3	1
4	1	4
5	5	4
6	9	4
7	0	13

Видно что начиная со второго шага одно из чисел равно сумме количеств четных и нечетных предыдущего шага, а второе равно одному из количеств предыдущего шага. Остается ответить на вопрос, в каком случае количество четных равно сумме, а в каком сумме равно количество нечетных.

Задача 2. Построение графика соревнований.

Требуется составить график соревнований по виду спорта предполагающего парные состязания. Вид спорта роли не играет, необходимо только выполнить следующие условия:

- в каждом бою участвуют два спортсмена;
- проигравший спортсмен выбывает из соревнования;
- в любом поединке встречаются спортсмены, проведшие или равное количество боев или у одного из них на один бой меньше.

Идея решения:

Предложим следующий способ определения пар бойцов: выстроим спортсменов в ряд. Пусть пары образуют спортсмены являющиеся соседями. Пары можно создавать проходя ряд слева на право (назовем это направление прямым) и наоборот проходя ряд справа на лево (назовем это направление обратным). Прямое и обратное направления не равнозначны, в том смысле что при прямом направлении боец без пары может оказаться на правом краю, а при обратном на левом.

Начнем построение пар, например с прямого прохода. Рассмотрим некоторый очередной шаг. Построение пар на очередном шаге может оставить без пары ровно одного бойца. Если на следующем шаге направление построения пар оставить тем же, то боец оставшийся без пары на предыдущем шаге, может опять остаться без соперника. Дальнейшие рассуждения очевидны.

Задача 3. Выборка из миллиарда.

Из числового интервала от единицы до миллиарда, случайным образом выбирается миллион, неповторяющихся чисел и записывается в файл. Необходимо за приемлемое время выяснить, наименьшее, отсутствующее в файле число. Использовать массивы или иные структуры данных могущих их заменить запрещается.

Пояснение. Пусть, например мы выбрали из интервала 1..5 два числа 1 и 4. Тогда наименьшее не выбранное это число 2.

Идея решения:

Задача имеет простое, но скажем сразу негодное решение. А именно, можно упорядочить файл в порядке возрастания, после чего пройти файл еще раз и найти первое число (назовем его Числом) отличающееся от своего соседа справа более чем на единицу. Тогда Искомое значение = Число + 1. Однако это плохое решение. Сортировка очень трудоемкая операция, даже если речь идет о сортировке массива. В задаче же сказано, что числа хранятся в файле, поэтому проблема усугубляется трудоемкостью операций файлового доступа. Хорошее решение должно избавить от необходимости многократного прохода файла.

А теперь хорошая идея. Поделим миллиардный интервал на два по 500 миллионов. Где может находиться искомое число? Очевидно в первом интервале, так как 500 миллионов миллионом чисел не заполнить. Далее, поделим первый 500-миллионный интервал на два и т.д. Рано или поздно заключение о том, что интервал, содержащий искомое число будет первым, может оказаться несправедливым.

Из сказанного выше ясно, что нужно найти интервал, длина которого окажется больше количества чисел в нем действительно содержащихся и этот интервал должен быть первым из обладающих таким свойством. Пусть, например, рассматриваемый интервал [3000, 3999]. Чтобы быть полностью заполненным, он должен содержать 1000 чисел (напомним, что числа в файле не повторяются, иначе чисел в интервале может оказаться больше 1000 и интервал при этом будет не заполнен). Известно, что все эти числа не меньше 3000 и не больше 3999. Прочитаем файл один раз и посчитаем, сколько в нем чисел от 3000 и до 3999. Если их окажется меньше 1000, то в этом интервале есть дырка.

Остался вопрос о том, на сколько интервалов делить исходный и промежуточные интервалы и когда завершать процесс деления. На этот вопрос вы найдете ответ самостоятельно.

Задача 4. Обратная польская запись.

Дано арифметическое выражение, состоящее из натуральных чисел, знаков арифметических операций, скобок. Преобразовать это выражение в форму обратной польской записи.

Идея решения:

Обратная польская запись это форма бесскобочной записи арифметического выражения. Примеры:

Традиционная запись: 2 + 2
Обратная : 2 2 +
Традиционная: 4 * (2 + 3)
Обратная : 4 2 3 + *

Первым в списке действий стоит знак плюс, поэтому выполняется сложение 2 + 3. Результат: 4 6 *. Следующее действие в списке – умножение, результат уже окончательный = 24.

Заметим, что в случае элементарного арифметического выражения, состоящего из одной арифметической операции, обратная запись получается переносом знака операции за правый аргумент. Эту операцию принципиально можно использовать для любых выражений, если учесть что аргумент справа, соответствующий знаку операции может быть двух видов:

- правый аргумент может оказаться выражением записанным в форме обратной польской записи;
- правый аргумент может оказаться еще не обработанным традиционным выражением.

В первом случае, если справа стоит обратная польская запись, то проблема переноса знака сводится к проблеме различения уже обработанной части выражения от необработанной. Представьте себе ситуацию, вы берете знак, переносите его через число, стоящее справа от него и обнаруживаете еще одно число. Вывод один. Это следующее число есть часть выражения, уже записанного в форме обратной польской записи, так как в традиционной форме два числа обязательно разделяются знаком.

Иной случай. Вы переносите знак через число и обнаруживаете знак. Такая ситуация неоднозначна. Этот знак может быть как частью обработанного, так и необработанного выражения. При этом сам знак не несет в себе никаких дополнительных признаков позволяющих определить, что он есть такое. А если признаков нет, то их необходимо создать. Например, перенесенный знак (являющийся частью обратной польской записи) можно выделять точками. Тогда проблема решается легко. Если переносимый знак наталкивается на еще один знак, то надо поискать точки выделения. Если они есть, то это новая форма записи, иначе это необработанная часть выражения.

Следующая проблема – баланс скобок. Если на пути переносимого знака встречаются скобки, то необходимо пройти столько закрывающих скобок, сколько и открывающих.

Алгоритм можно оформить как процесс последовательного переноса знаков за правый аргумент с учетом ситуаций описанных выше. Каждый перенесенный знак выделяется точками, для того чтобы его можно было отличить от еще не перенесенного. По завершении процесса достаточно удалить все точки и оставшиеся скобки.

Задача 5. Движение в поле сил тяготения.

Несколько тел известной массы с известными векторами начальных скоростей находятся в пустом трехмерном пространстве. Необходимо построить их траектории движения.

Идея решения:

Закон всемирного тяготения позволяет в каждый момент времени вычислить суммарную силу действующую на тело со стороны других тел. Второй закон Ньютона позволяет рассчитать ускорение полученное от этой силы. Зная ускорение, можно построить закон равноускоренного движение тела.

Проблема заключается в том, что сила и соответственно ускорение определяются взаимным расположением тел. А это взаимное расположение в следующий момент будет уже другим. Следовательно, мы вынуждены рассчитывать ускорение для каждого момента времени и следовательно движение тел нельзя считать равноускоренным.

Движение такой системы тел описывается системой дифференциальных уравнений, решение которой представляет собой значительную проблему. Но можно подойти к задаче с другой стороны. Допустим, что силы взаимного тяготения изменяются не непрерывно, а дискретно, с некоторым временным интервалом. В течении выделенного временного интервала сила считается постоянной величиной не зависящей от взаимного расположения тел. По завершении интервала силы действия на каждое тело опять пересчитываются с учетом уже нового положения тел.

Такая модель позволяет свести движение по сложной криволинейной траектории к ломаной траектории состоящей из множества прямых отрезков, вдоль которых все тела движутся равноускоренно. Схема расчетов такова:

Разобьем временную ось на равные интервалы. Точки разбиения – это точки пересчета сил тяготения. В каждой точке выполняются следующие расчеты (для каждого тела):

- рассчитываются вектора силы и соответственно вектора ускорений созданные воздействиями на данное тело других тел системы;
- рассчитывается суммарное ускорение равное векторной сумме ускорений.

Далее в течении следующего интервала времени, сила считается постоянной и тело движется равноускоренно.

Ясно, что такая модель даст траекторию с некоторой, может быть даже значительной погрешностью, но увеличивая количество точек пересчета сил (уменьшая интервал равноускоренного движения), можно сколь угодно уменьшать погрешность расчетов.

Важное замечание. Обдумайте, как будет вести себя модель в случае удара двух или более тел.

Задача 6. Одинокий путник с плохой памятью.

На квадратном поле установлены препятствия произвольной формы и два пункта А и В. Перед путником поставлена задача – найти путь из А в В. Известно, что путник не располагает картой местности, у него очень плохое зрение, он практически видит на одну точку впереди себя и очень плохая память, настолько плохая, что он может запомнить только одно число. Из средств ориентации путник располагает естественным чувством правого и левого, он может сказать где у него правая рука, а где левая. Также у него есть прибор всегда показывающий на пункт В.

Идея решения:

Для упрощения анализа договоримся, что препятствия состоят из горизонтальных и вертикальных линий. То есть существует какая-то координатная сетка и препятствия по ней ориентированы. Такую координатную сетку можно

например привязать к экрану компьютерного монитора. Эта договоренность никак не упрощает задачу и не ограничивает ее общности.

Разобьем движение путника на две части:

- движение по пустому пространству;
- обход обнаруженного препятствия.

Движение по пустому пространству не представляет интереса. Согласно условия у путника есть компас, показывающий на цель. Следовательно, путнику на пустом пространстве достаточно идти по компасу.

Второй случай существенно интереснее. Путник, встретив перед собой препятствие должен обойти его и в некоторой точке опять начать движение по компасу (оторваться от препятствия).

Обойти препятствие путник может держась за него правой или левой рукой. Вопрос завершения обхода сложнее. Образно говоря, путник должен оторваться от препятствия тогда когда он окажется на «другой стороне». А вот, что такое «другая сторона» не вполне ясно. Для того, чтобы прояснить суть проблемы рассмотрим ошибочное решение.

Пусть путник, двигаясь вдоль препятствия, на каждом шагу проверяет, нельзя ли начать движение по компасу. То есть он полагает, что как только по компасу впереди него образовалось пустое пространство, то значит он вышел на другую сторону. В ряде случаев это действительно так. На следующем рисунке подтверждающий пример:

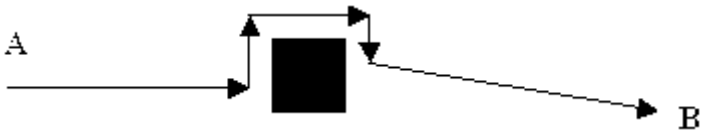

Рис. 3.1. Удачный обход препятствия

В этой ситуации путник пройдя две стенки обнаружит перед собой пустое пространство до пункта назначения. Но так будет далеко не всегда. На следующем рисунке плохая ситуация.

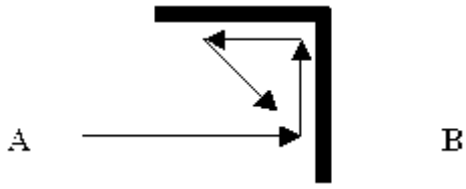

Рис. 3.2. Неудачный обход препятствия

Сделав один поворот путник обнаружит свободное пространство между собой и целью, но движение по компасу приведет его к той же стенке, вдоль которой он уже двигался. Далее, движение станет бесконечным по замкнутому контуру. Видимо просто факт поворота не должен быть причиной попытки отрыва от препятствия.

Заметим, что в распоряжении путника только два типа поворотов. Назовем повороты которыми путник обходит квадрат из рисунка 3.1 правыми, а повороты того же типа как поворот из рисунка 3.2 левыми. Очевидно, что можно дать описание пути вдоль препятствия последовательностью левых и правых поворотов. Очевидно также, что расстояние от одной точки поворота до следующей не играет никакой роли, существенное значение имеет только факт поворота. Можно также сформулировать следующую гипотезу: *Для решение вопроса о «той стороне препятствия» достаточно информации о разнице между количествами левых и правых поворотов.*

Обдумайте эту гипотезу, она дает ключ к решению задачи.

Задача 7. Закраска односвязного контура

На плоскости задан произвольный односвязный контур, требуется его закрасить. Из всех графических возможностей разрешается использовать только процедуру рисования точки.

Идея решения:

Односвязность контура означает следующее: пусть о точках A и B известно, что они находятся внутри контура. Тогда существует ломаная линия такая что:

- начало ломаной находится в точке A;
- конец ломаной находится в точке B;
- все точки принадлежащие ломаной, принадлежат внутренней области контура.

Представим процесс закраски образно. Есть некоторая точка (будем называть ее источник) о которой достоверно известно, что она находится внутри контура. А контур это что-то вроде низкого заборчика через который краска не может перелиться. Если лить краску в точку-источник, то краска начнет разливаться от точки к точке пока не зальет весь контур. Это образ. А сейчас перейдем к более строгому представлению.

Будем считать каждую точку внутри контура, источником если она только что закрашена. Каждая такая точка является источником для четырех соседей. После того как соседи закрашиваются, их источник перестает быть источником, а вновь закрашенные точки становятся новыми источниками.

Создадим массив источников, в начале процесса содержащий одну точку. Процесс же закраски можно описать следующим алгоритмом:

Пока в массиве источников есть хотя бы один источник выполнять:

- взять из массива очередной источник;
- определить незакрашенных соседей очередного источника;
- закрасить соседей и поместить их в массив источников;
- удалить из массива отработанный источник.

Важная проблема.

Рассмотрим следующий рисунок:

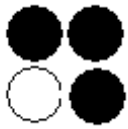

Рис. 3.3. Дублирование источника

Здесь изображены четыре точки. Пусть три закрашенные являются источниками, одна незакрашенная – сосед для всех трех. Следовательно, все три источника в процессе обработки занесут своего общего соседа в массив источников. Так как такие ситуации не редкость, в массиве источников появится много копий. Идея – проверки массива на наличии копии, перед занесением источника не очень удачна, такая проверка выполняясь многократно, над большим массивом потребует много ресурсов. Интереснее обдумать порядок занесения источников и соответственно их выборки.

Задача 8. Живая группа ГО.

В игре ГО есть понятие живой группы. Требуется, имея координаты одного камня, установить, является ли этот камень представителем живой группы.

Идея решения:

Для понимания сути задачи, нет необходимости детально изучать правила игры ГО, тем более что они достаточно сложны. Рассмотрим только понятие группы и живой группы. Игра ГО ведется на доске 19×19 клеток. Каждый игрок в свою очередь хода ставит одну фишку, называемую камнем на перекрестие линий. Два камня называются соседями, если они соседи по вертикали или горизонтали. Множество камней называется группой если от любого камня группы можно пройти до любого камня группы передвигаясь по соседям. Множество из одного камня также считается группой.

Группа считается живой, если хотя бы одному камню из этой группы можно добавить соседа.

На рисунке 3.4 две живых группы. Из белых камней и из черных.

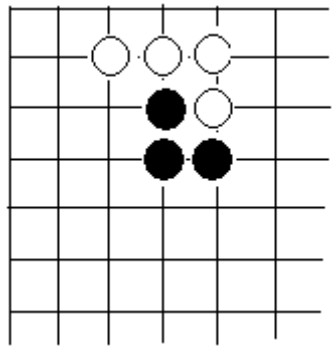

Рис. 3.4. Пример живой группы

На рисунке 3.5 группа черных камней полностью окружена тремя группами белых.

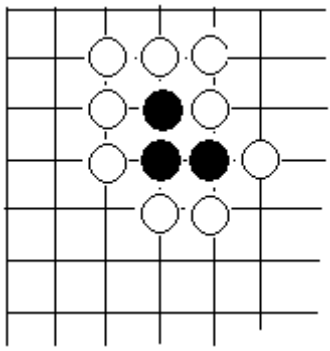

Рис. 3.5. Пример неживой группы

Первый шаг в построении рекурсивного решения – это получение рекуррентного определения процесса. В нашей задаче рекуррентная природа процесса налицо, процесс заключается в переходе от камня к камню в поиске хотя бы одного свободного «соседа». Если такой «сосед» найден, то группа живая, иначе нет. Схема рекурсии может быть такова:

- если рассматриваемое поле пусто, то поиск прекращается и возвращается сообщение о удачном завершении;
- если рассматриваемое поле занято камнем противника, то поиск прекращается и возвращается сообщение о неудаче данной ветки поиска. Именно ветки, так как частная неудача не означает полной неудачи, другая ветка может обнаружить свободное поле;
- если рассматриваемое поле занято своим камнем, то выполняется новый вызов рекурсивной процедуры.

Второй вариант решения:

Задачу о живой группе ГО можно свести к седьмой задаче . Для этого достаточно заметить лишь, что поля доски ГО как и точки в закрашиваемом контуре имеют 4 соседей. Далее, представим группу камней, как группу точек источников. Дальнейшие рассуждения для самостоятельной проработки.

Задача 9. Поиск пути с наибольшим весом.

Дана матрица, заполненная положительными, целыми, случайными числами. Необходимо найти путь из левого верхнего угла матрицы в правый нижний с наибольшим возможным весом. Весом пути будем называть сумму чисел всех элементов матрицы, через которые проходит путь. Путь можно строить только двумя типами смещений: вправо на один шаг и вниз на один шаг.

Идея решения:

Задача выглядит как задача полного перебора. Действительно, ее можно решить организовав полный перебор всех возможных путей с выбором пути наибольшего

веса, но количество путей с увеличением размера матрицы растет настолько быстро, что реально даже для относительно небольшой матрицы потребуется слишком много времени.

Более эффективное решение основано на небольшом вспомогательном построении: пройдя всю матрицу снизу вверх и справа на лево определим для каждого элемента новый вес, равный сумме собственного веса и весов двух элементов: справа от данного и снизу от данного. Такая обработка не займет много времени. После чего искомый путь от левого верхнего элемента, до правого нижнего пройдет по элементам с наибольшим весом. Ниже пример.

Таблица 3.2. Пример построения

1	2	8	3	5
3	2	2	1	45
5	1	1	0	4
2	3	4	3	8
1	1	3	4	8

1480	**816**	**419**	190	70
663	395	**221**	**117**	**65**
265	172	102	51	**20**
88	69	50	31	**16**
17	16	15	12	**8**

Левая таблица содержит исходную матрицу, правая вспомогательную. Жирным шрифтом отмечен искомый путь.

Задача 10. Поиск потенциально рекурсивной процедуры.

Дан текст программы, состоящий из некоторого количества процедур. Процедуры могут содержать вызовы других процедур. Необходимо выяснить, нет ли среди них потенциально рекурсивных. Потенциально рекурсивной процедурой назовем процедуру, порождающую цепочку вызовов такую, что в этой цепочке встречается вызов процедуры породившей цепочку.

Идея решения:

Так как целью анализа является только лишь выявление факта рекурсии, можно считать, что каждая процедура состоит только лишь из вызовов процедур. Названия процедур также роли не играют, их можно пронумеровать в любом порядке и имя заменить на порядковый номер. Составим для множества номеров процедур таблицу. Верхний ряд и левый столбец в таблице заполним номерами процедур, а остальные ячейки таблицы нулями и единицами. Единица на пересечении строки и столбца означает, что процедура чей номер указан слева вызывает процедуру чей номер указан вверху. Пример

Таблица 3.3. Таблица вызовов

	1	2	3
1	0	0	1
2	1	0	1
3	0	1	0

Из таблицы видно, что первая процедура вызывает третью, вторая первую и третью, третья вторую. Здесь все три процедуры потенциально рекурсивны. Это следует из того, что процедура 1 вызывает процедуру 3. Она в свою очередь вызывает процедуру 2, а процедура 2 вызывает две процедуры, в том числе и процедуру 1. То есть все три процедуры вызывают друг друга по кругу.

Примечание. Почему речь идет не о рекурсивности, а о потенциальной рекурсивности? Дело в том, что мы исключаем из анализа все операторы программы, кроме вызовов процедур, это означает, что нет анализа реального процесса передачи управления в программе, а в реальном процессе, зависящем от состояния данных управление на вызов процедуры может и не быть передано.

Будем считать, что вызывающие процедуры перечислены в левом столбце, а в верхней строке вызываемые. Определим одномерный массив для хранения номеров вызываемых процедур. Пусть на рекурсивность анализируется процедура с номером k. Внесем в массив число k (в первый элемент массива). Далее шаг процесса можно описать следующим образом:

1. Если массив пуст, то прекращаем работу. В этом случае процедура не является потенциально рекурсивной.
2. Если массив не пуст, извлекаем очередное число (удаляя его из массива). Взятое число есть номер *очередной* строки таблицы.
3. Просматриваем *очередную* строку таблицы и если в j-столбце стоит 1 (процедура вызывается), то выполняем проверку, есть ли в массиве число j. Если таковое есть, то прекращаем работу, в этом случае процедура является потенциально рекурсивной, иначе заносим j в массив.
4. По завершению анализа k – ой строки, если работа алгоритма не была прервана, возвращаемся к пункту 1 и переходим к анализу следующей строки таблицы.

Примечание. Для построения алгоритма вам необходимо определить, что означает фраза «массив пуст»

Задача 11. Построение модели стоячей волны.

Среда представляет собой множество равноотстоящих точек, находящихся на одной прямой. Левая крайняя точка порождает поперечную волну. Волна определяется двумя характеристиками: высотой (амплитуда) и скоростью распространения. Правая крайняя точка является отражающей стенкой. Волна доходя до стены отражается от нее без потерь энергии, то есть начинает двигаться в противоположную сторону, с той же амплитудой накладываясь на прямую волну.

Идея решения:

Волна распространяется в линейной среде, состоящей из равноотстоящих точек. Волновое движение в исходной точке (крайней левой) определяется законом Y=A*sin(wt) где w – частота колебаний, t – время, A – амплитуда. Колебательный закон в любой точке находящейся правее источника определяется тем же самым законом, но начало колебательного процесса смещено по времени на величину (N-1)*Dt где N – номер точки, Dt – время прохождения волны между точками.

Когда волна доходит до правой крайней точки, правая крайняя точка становится новым источником волны с той же амплитудой, но противоположным направлением.

Задача 12. Умножение многочленов.

Два многочлена различной степени заданы массивами коэффициентов. Для небольшого упрощения можно положить все коэффициенты целыми числами. Вычислить массив коэффициентов многочлена являющегося их произведением.

Идея решения:

Многочлен можно представить множеством коэффициентов. Запишем два многочлена – множителя в следующем виде (a_0, a_1, ..., a_n) и (b_0, b_1, ..., b_m), $n \neq m$ перемножаемые многочлены могут иметь разные степени. Многочлен – произведение также можно представить в виде (c_0, c_1, ..., c_L). Возьмем какой-либо элемент многочлена – произведения, c_k. Очевидно что c_k равен сумме всех произведений элементов множителей $a_i * b_j$ таких что k=i + j. Это основная идея. Вам осталось додумать, чему равна длина массива произведения или что то же самое чему равно L.

Задача 13. Подсчет черных пятен на белой шкуре.

На белой шкуре имеется некоторое количество черных пятен. Найти самое большое пятно.

Идея решения:

Естественно шкуру представить в виде двумерного числового массива, в котором черные точки это единицы, а белые – нули.

Предположим, мы имеем некую процедуру, умеющую по одной точке пятна найти все пятно. Тогда будет достаточно просмотреть все точки массива и если очередная точка содержит число 1, выполнить построение пятна от очередной точки.

Построение пятна может заключаться в замене единиц на двойки. Такая замена позволит избежать повторного обнаружения одного и того же пятна, так как все точки обнаруженного пятна станут двойками, а запуск процедуры осуществляется только при обнаружении единицы.

Если описанную выше процедуру научить возвращать размер пятна, то решение задачи сведется к поиску наибольшего из возвращаемых чисел.

Метод определения точек принадлежащих пятну может быть разным. Мы эту проблему рассматривали уже дважды: в задаче 7 о закраске контура и в задаче 8 о живой группе ГО.

Задача 14. Квадратные суммы.

Дана квадратная матрица a[0..n][0..n] и число m<=n Для каждого квадрата размера m на m в этой таблице вычислить сумму стоящих в нем чисел.

Идея решения:

Для простого, но довольно грубого решения достаточно организовать два вложенных цикла для обхода почти (почему почти?) всех элементов матрицы и расчета суммы квадрата для которого текущий элемент будет левой верхней вершиной. Серьезный недостаток этого решения в том, что каждый элемент исходной матрицы поучаствовует в расчетах многократно.

Более эффективную идею вы можете доработать сами с небольшой подсказкой. Вычислим сумму элементов только одного квадрата с вершиной в левой верхней точке исходной матрицы. Если теперь этот квадрат сдвинуть вправо, то сумму элементов нового квадрата можно получить из суммы предыдущего, если вычесть из предыдущей суммы m – элементов левого столбца (ушедшего из квадрата) и добавить m – элементов правого столбца (включенного в квадрат на этом шаге).

Задача 15. Быстрое вычисление степени.
Дано два целых положительных числа N и M. Вычислить N^M
Идея решения:
Алгоритм строится на следующих очевидных утверждениях:

- если в выражении a^n n=2m (то есть четно) то $a^n=(a*a)^m$ и таким образом показатель степени уменьшается вдвое;
- если в выражении a^n n=2m+1 (то есть нечетно) то $a^n=(a^{n-1})*a$ таким образом показатель степени уменьшается на 1 и приводится к четному числу.

Задача 16. Целочисленная решетка.
На плоскости задана целочисленная решетка (координаты точек решетки – целые числа). На плоскости также задан многоугольник, координаты которого рациональные числа. Определить количество целых точек лежащих в вершинах и на сторонах данного многоугольника.
Идея решения:
Только небольшой намек. Задача сводится к определению точек, которые лежат на прямой заданной двумя точками. Это во-первых. Во-вторых, задание прямой через две фиксированные точки сводится к составлению пропорции.

Задача 17. Удаление комментариев.
Некоторый входной файл содержит текст программы. В тексте могут находится комментарии. Каждый комментарий открывается последовательностью символов (* и закрывается последовательностью *). Комментарии могут быть вложенными. Требуется удалить все комментарии и создать файл с чистым текстом.
Идея решения:
Введем для каждого символа понятие разности скобок. В нашем случае открывающей скобкой называется последовательность символов (* и закрывающей скобкой последовательность *). Разностью символа назовем разность открывающих и закрывающих скобок стоящих в тексте до данного символа. Разность может быть трех видов:

- открывающих скобок больше чем закрывающих. Это означает, что данный символ находится внутри комментария;
- открывающих скобок меньше чем закрывающих. Это означает, что комментарии расставлены неверно;
- количество открывающих скобок равно количеству закрывающих. Это означает, что данный символ относится к чистому тексту.

Задача 18. Разложение многочлена.

Дан многочлен вида $a_0 + a_1x + a_2x^2 + ... + a_nx^n$, коэффициенты которого – целые числа от 0 до L – 1 . Выяснить, можно ли его разложить на произведение двух многочленов с коэффициентами из того же интервала от 0 до L – 1 .

Идея решения:

Многочлен можно записать в виде $a_0a_1a_2 ...a_n$ а это ничто иное как запись числа в L – ичной системе счисления. Таким образом задача разложения многочлена сводится к разложению числа записанного в L – ичной системе счисления на два множителя.

Задача 19. Построение последовательностей.

Построить все возможные последовательности состоящие ровно из N чисел из интервала 1..k так чтобы каждая последовательность отличалась от предыдущей только в одном числе и ровно на 1.

Идея решения:

Представим себе доску, разлинованную на квадраты длиной в N – квадратов и шириной в k квадратов. Расставим на первой горизонтали фишки с нарисованными на них стрелками и пусть стрелки в данном начальном положении на всех фишках ориентированы одинаково в сторону последней горизонтали. Если поля доски пронумеровать (по ширине) числами от 1 до k то любая расстановка фишек представляет некоторую последовательность. Будем считать, что описанная расстановка дает первую последовательность. Любую другую последовательность можно получить из уже имеющейся по следующему правилу:

- найдем самую правую фишку, которую можно сдвинуть в направлении стрелки на ней нарисованной;
- все шашки правее нее повернем стрелками в противоположную сторону (после этого они опять могут двигаться).

После реализации алгоритма докажите, что таким способом действительно будут получены все возможные последовательности.

Задача 20. Выпуклая оболочка.

На плоскости дано множество из N точек. Построить на этом множестве выпуклую оболочку. Выпуклой оболочкой называется выпуклая фигура, содержащая все точки.

Идея решения:

Найдем четыре точки: самую верхнюю, самую нижнюю, самую левую и самую правую. Проведем через эти точки четыре линии. Через нижнюю и верхнюю – горизонтальные, через правую и левую вертикальные. В результате получим прямоугольник являющийся выпуклой оболочкой данного множества, но не являющийся решением поставленной задачи, так как в его углах может не быть точек множества. Обрежем углы полученного прямоугольника, соединив соседние точки. В результате опять получим выпуклую оболочку, в углах которой гарантированно

стоят точки множества, но сейчас возможно не все точки находятся внутри полученной выпуклой фигуры. Далее начнем расширять выпуклую фигуру до тех пор пока это возможно следующим способом:

1. Выберем пару соседних вершин выпуклой фигуры. Отрезок их соединяющий назовем Данным отрезком.
2. Среди точек внешнего множества (точки не попавшие в выпуклую фигуру) найдем точку отстоящую на наибольшее расстояние от Данного отрезка. Впрочем, такой точки может и не быть. И на самом деле поиск такой точки необходимо осуществлять не среди всех точек внешнего множества. Вам необходимо уточнить множество поиска.
3. Найденная точка соединяется отрезками с выбранной парой вершин и становится новой вершиной выпуклой фигуры.
4. Для каждой точки внешнего множества проверяем, не попала ли она внутрь новой выпуклой фигуры

Процесс завершается, когда внешнее множество окажется пусто. Еще один важный, но не рассмотренный вопрос – это критерий позволяющий определять попадает ли точка во внутреннюю область выпуклой фигуры. Этот вопрос остается на самостоятельный анализ.

Задача 21. Лишние скобки.

Дано правильное арифметическое выражение, состоящее из букв (латинский алфавит A..Z, буквы используются без индексов), цифр, знаков арифметических операций и круглых скобок, записанное в общепринятой форме. Составить программу, удаляющую из выражения лишние пары скобок, не влияющие на порядок выполнения операций.

Идея решения:

Необходимо рассмотреть все возможные ситуации, в которых удаление скобок не влияет на ход выполнения вычислений. Таких ситуаций не очень много:

- + (выражение) + скобочное выражение слева и справа участвует в сложении;
- + (выражение) – скобочное выражение справа участвует в сложении, слева в вычитании.

Во всех остальных случаях скобки существенно влияют на ход расчетов. Например, в следующем варианте – (выражение) + для того, чтобы убрать скобки, необходимо внутри поменять знаки всех чисел и переменных. И существует несколько особых случаев в которых знаки перед скобками не играют никакой роли. Это следующие ситуации:

- никакого знака (выражение) никакого знака. В этом случае скобки просто закрывают все выражение как единое целое;
- знак ((выражение)) знак. Скобки дублируются. Одна пара лишняя;
- знак (пусто) знак. Впрочем такую ситуацию можно считать ошибочной записью выражения.

Задача 22. Поиск палиндрома.

Дан одномерный символьный массив. Найти в нем нечетный палиндром наибольшей длины. Нечетный палиндром – это палиндром, состоящий из нечетного количества элементов.

Идея решения:

Любой элемент массива может быть центром палиндрома. Максимально возможный палиндром это весь массив, поэтому центром максимально возможного палиндрома будет срединный элемент массива.

Задача разбивается на две подзадачи: первая обход всех возможных центров, фактически это означает перебор всех элементов массива и поиск наибольшего палиндрома для каждого из центров. Но для оптимизации поиска перебор элементов массива лучше выполнять от центра к краям.

Задача 23. Построение последовательности чисел.

Построить числовую последовательность, удовлетворяющую условиям: первое число последовательности равно нулю. Каждое очередное удовлетворяет следующим требованиям:

- оно больше предыдущего;
- оно не содержит цифр имеющихся в десятичной записи предыдущего;
- оно наименьшее из множества чисел удовлетворяющих двум первым условиям.

Идея решения:

Представим число массивом цифр. Предположим, что некоторое число последовательности уже получено. Назовем его очередным. Следующее число может быть получено из очередного по следующим правилам:

- так как новое число минимальное из возможных, оно должно быть получено либо заменой цифры в как можно более младшем разряде, либо увеличением количества разрядов на 1;
- замена цифры в разряде выполняется на цифру, такую, что ее нет в старших разрядах текущего числа и она больше цифры текущего разряда (на 1, на 2 и т.д.);
- замены цифры необходимо начинать с младшего разряда;
- если цифру в текущем разряде заменить не удалось, следует перейти к следующему по старшинству разряду;
- если не удалась замена цифры в старшем разряде, то следует увеличить количество разрядов.

Искомая последовательность: 0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 22, 30, 41, 50, 61, 70, 81, 90, 111, ...

Задача 24. Минимальное количество заправок.

Автомобиль движется из пункта A в пункт B, начав движение с полным баком. Расход бензина на один километр пути известен. Между пунктами A и B находится некоторое количество заправок. Определить, на каких бензоколонках необходимо заправляться, чтобы количество заправок было минимальным.

Идея решения:

После каждой заправки автомобиль имеет полный бак. С полным баком автомобиль может пройти вполне определенный путь. Обозначим путь с полным баком за отрезок длины L. Тогда от каждой заправки, включая исходную точку можно отложить отрезок длины L. Если множество отрезков не перекрывают весь путь, то решения не существует. Если перекрывают, то возможно на некоторых участках перекрытие множественное, то есть участок пути перекрывается несколькими отрезками L. Если есть отрезок L такой что любой участок дороги, перекрывающийся данным отрезком, перекрывается еще хотя бы одним, то данный отрезок (соответственно заправку) можно исключить.

Для определения лишнего отрезка рассчитаем для каждого километра индекс перекрытия равный количеству отрезков L перекрывающих данный километр. Затем для каждого отрезка пройдем все километровые точки и просмотрим их индекс перекрытия, если для данного отрезка нет ни одного километра с индексом перекрытия равным единице, то отрезок (заправка) лишний.

Примечание. Для упрощения полагается, что число L – целое. Но это упрощение никак не ограничивает общности задачи.

Задача 25. Очистка памяти.

Память компьютера состоит из пронумерованных ячеек памяти. Некоторое количество блоков памяти уже занято данными. Операционной системе требуется еще один блок памяти определенной длины. Желательно выделить его в свободной области и если это невозможно, то освободить некоторое количество занятых блоков. Требуется найти вариант, при котором освобождался бы минимальный объем занятой памяти.

Идея решения:

Пронумеруем границы блоков памяти, как пустых, так и занятых. Точка являющаяся границей между пустым и занятым блоком отмечается один раз. Можно доказать, что самый выгодный вариант размещения блока имеет своей левой границей одну из отмеченных точек.

Далее, остается лишь попытаться выделить нужный блок от каждой из отмеченных точек и выяснить в каком случае новым блоком перекрывается меньше блоков уже занятой памяти.

Необходимо учесть, что если свободный блок перекрывает только часть занятого, то высвободить необходимо все равно весь занятый блок.

Задача 26. Правильное арифметическое выражение.

Дана последовательность из левых и правых круглых и квадратных скобок. Выяснить, можно ли добавкой в эту последовательность чисел и знаков арифметических операций получить правильное арифметическое выражение.

Идея решения:

Фактически необходимо выяснить две вещи:

- имеет ли место баланс скобок, отдельно круглых и отдельно квадратных. Если нет баланса, то правильное выражение составить нельзя;

- нет ли пересечения пары круглых и квадратных скобок. Возможны два типа пересечений: ([)] и [(]). Если есть хотя бы одно пересечение, правильное выражение составить нельзя;

Как проверить баланс скобок

Пусть проверяется баланс круглых скобок. Пройдем все выражение, считая разность открывающих и закрывающих скобок. Если появляется открывающая скобка, то к разности прибавляем единицу, если закрывающая, то вычитаем единицу. Если разность хотя бы раз станет меньше нуля, то это будет означать нарушение баланса. Конечно же оба баланса, и для круглых и для квадратных скобок можно считать одновременно.

Как проверить пересечение

Проблема пересечения также сводится к проверке баланса скобок. А именно внутри любой пары открывающих и закрывающих круглых скобок должен иметь место баланс квадратных. То же самое можно утверждать и про квадратные скобки. Завершите разработку идеи самостоятельно.

Раздел B. Сортировки

Задача 27. Сортировка слиянием.

Сортировка слиянием представляет собой процесс на каждом шагу которого, массив переписывается в другой массив, немного при этом упорядочиваясь. На первом шаге массив разбивается на пары и упорядочение выполняется внутри пар. На каждом последующем шаге в новые пары объединяются группы уже отсортированные на предыдущем шаге. При объединении двух групп в общую, элементы сливаемых групп переписываются в порядке возрастания. Приведем пример.

Исходный массив (3; 7; 1; 8; 2; 1; 6; 1; 2)

1. (3; 7), (1; 8), (1; 2), (1; 6), (2) массив разбит на пары и пары упорядочены. Заметим, что одному числу пары не хватило.
2. [(3; 7), (1; 8)], [(1; 2), (1; 6), (2)] выделяем группы
3. [(3; 7), (1; 8) результат (1; 3; 7; 8)], [(1; 2), (1; 6), (2) результат (1; 1; 2; 2; 6)]
4. (1; 3; 7; 8), (1; 1; 2; 2; 6) результат (1; 1; 1; 2; 2; 3; 6; 7; 8)

Задача 28. Быстрая сортировка.

Разобьем исходный массив на два подмассива, относительно центрального элемента (назовем его барьером). Для всех элементов обоих подмассивов выполним следующую операцию: если элемент левого подмассива больше барьера, то перенесем его вправо, если элемент правого подмассива меньше барьера, то перенесем его влево.

По завершении этого процесса массив окажется немного более упорядоченным. Продолжим упорядочение, разбив каждый из уже обработанных подмассивов, еще на два подмассива и проведем над ними ту же самую операцию. Полностью

процесс сортировки будет завершенным, когда длина подмассивов окажется единичной. Быстрая сортировка имеет один недостаток. Например, если исходный массив окажется изначально полностью или почти полностью упорядоченным, это не уменьшит количество выполняемых операций. Быстрая сортировка все равно проведет дробление исходного массива до победного конца.

Приведем пример. Исходный массив (3; 7; 1; 8; 2; 1; 6; 1; 2). Число – барьер будем выделять жирным шрифтом. В качестве барьера можно брать любой элемент массив. В нашем примере барьером послужат центральные элементы:

1. (3; 7; 1; 8; **2**; 1; 6; 1; 2) = (1; 1; 1; 2), **2,** (3; 7; 8; 6)
2. (1; 1; **1**; 2), **2**, (3; **7**; 8; 6) = (1; 1), **1,** (2), **2,** (3; 6), **7,** (8)
3. (1; **1**), **1,** 2, **2,** (**3**; 6), **7,** 8 = 1, 1, 1, 2, 2, 3, 6, 7, 8

Задача 29. Сортировка Шелла.

Алгоритм Шелла, можно рассматривать, как модификацию алгоритма вставки. Цель алгоритма – заставить каждый сортируемый элемент за шаг вставки перемещаться большим скачком. Для этого исходный массив разбивается на группы специальным образом. Предположим в массиве 16 чисел (число равное степени двойки взято только для удобства иллюстрации). Выполним разбиение массива на пары следующим образом: (a_1, a_9), (a_2, a_{10}).... (a_8, a_{16}). Внутри каждой группы выполняем сортировку способом вставок. Затем переходим к следующему разбиению, в котором элементы массива объединяются уже по 4 следующим образом (a_1, a_5, a_9, a_{13}), (a_4, a_8, a_{12}, a_{16}) и т.д. То есть элементы отстоят друг от друга на 4 позиции. На следующем шаге соответственно на 2 позиции и на последнем на 1. Таким образом, последовательность расстояний внутри групп в нашем примере такова 8, 4, 2, 1. Ясно, что такая последовательность невозможна для любого массива, но сортировка Шелла этого и не требует. Существенно важно только одно – последнее смещение должно быть равно 1, последовательность же может быть любой, в любом случае сортировка сделает свою работу, но выбор последовательности существенно влияет на скорость обработки. В качестве примера возьмем то же массив (3; 7; 1; 8; 2; 1; 6; 1; 2) принадлежность элемента массива отметим нижним индексом:

1. (3_1; 7_2; 1_3; 8_4; 2_1; 1_2; 6_3; 1_4; 2_4) четвертой группе на один элемент больше из-за нечетности исходного множества. Результат = (2_1; 3_1; 1_2; 7_2; 1_3; 1_4; 2_4; 8_4; 6_3)
2. (2_1; 3_2; 1_1; 7_2; 1_1; 1_2; 2_1; 8_2; 6_1) Результат = (1_1; 1_1; 2_1; 1_2; 3_2; 7_2; 2_1; 8_2; 6_1)
3. (1_1; 1_1; 2_1; 1_1; 3_1; 7_1; 2_1; 8_1; 6_1) Результат = (1_1; 1_1; 1_1; 2_1; 2_1; 3_1; 6_1; 7_1; 8_1;)

Задача 30. Двоичная сортировка.

Предположим элементы массива расположены в вершинах двоичного дерева со следующим свойством: с каждой вершиной дерева (предком) связаны две вершины – потомка, такие что левый потомок всегда меньше предка, а правый больше либо равен предку.

Имея такое представление легко собрать массив, в порядке возрастания обходя дерево по правилу: Левый потомок – Предок – Правый потомок. Указанное

требование к сожалению не дает информации для единственного представления дерева. Поэтому приведем пример. Пусть дан следующий массив: 7, 1, 3, 1, 8, 1, 9, 5, 6, 13, 0. Соответствующее ему дерево выглядит следующим образом (рис. 3.6):

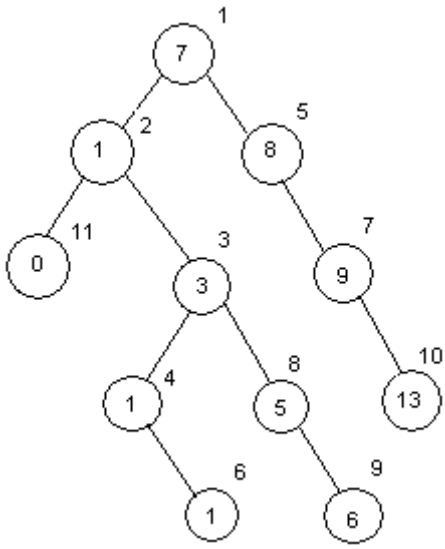

Рис 3.6. Двоичное дерево

В кружочках отмечены числа массива, над кружками порядок построения дерева. Таким образом сортировка двоичным деревом заключается в выполнении двух операций:

- пошаговом построении дерева, на каждом шагу которого определяется вершина подходящая для подклеивания очередной вершины;
- обход построенного дерева по указанному выше правилу.

Примечание: Правило: Левый потомок – Предок – Правый потомок также не совсем полно. Над его детализацией вам придется подумать самостоятельно. Для подсказки укажем порядок обхода дерева – примера (указываются номера вершин): 6, 4, 11, 2, 3, 8, 9, 1, 5, 7, 10

Раздел С. Задачи перебора

Задача 31. Построение выборок.
Дан массив символов. Построить все возможные выборки.
Идея решения:
Для этой задачи рассмотрим два варианта решения.

Вариант 1. Представление выборки двоичным числом.
Пронумеруем множество (будем далее называть его базовым) из элементов которого должны быть составлены выборки. Нумерация выполняется естественным образом, если множество представлено массивом. Составим еще один массив (назовем его двоичным числом) целых чисел, значениями которого могут быть только : 0 и 1.

Правило построения выборки таково: если k-ая цифра двоичного числа равна единице, то k-ый элемент базового множества входит в выборку, иначе нет.

Таким образом, задача получения всех выборок, сводится к задаче получения всех k-разрядных двоичных чисел. Получить же все двоичные числа можно так:

- в начале процесса проведем инициализацию двоичного числа нулями;
- очередное двоичное число получается прибавлением к предыдущему единицы по правилам двоичной арифметики.

Примечание. В общем-то здесь речь идет не совсем о двоичном числе. У числа не может быть ведущих нулей, здесь же мы учитываем все разряды «двоичного числа». Точно такая ситуация имеет место в некоторых последующих задачах раздела.

Вариант 2. Рекурсивный.
Заметим, что каждый элемент базового множества может либо входить в выборку, либо не входить. Если для выборки выделить отдельный массив, с длиной равной длине массива базового множества, то для каждого элемента массива выборки возможно два состояния: либо этот элемент пуст, либо он содержит соответствующий элемент базового множества.

Для первого элемента массива выборки возможны два состояния (содержит, не содержит), для каждого состояния первого элемента возможны два состояния второго элемента и т.д. Введем понятие последовательности выборки. Последовательностью выборки длины K, будем называть часть массива выборки длины K, начиная с первого элемента. Множество последовательностей длины K, возможно определить через множество последовательностей длины K-1 следующим образом: каждая последовательность длины K получается из некоторой последовательности K – 1 добавлением в позицию K либо пробела, либо K-того элемента базового множества.

Тогда множество выборок – это множество последовательностей длины N. Полученное определение имеет рекуррентный вид, из чего следует возможность получения рекурсивного решения. Строится рекурсивное решение на следующих очевидных соображениях:

- цель искомой процедуры за N вызовов построить очередную выборку, следовательно, на N – ом вызове можно завершить построение очередной выборки и выполнить распечатку;
- каждый вызов удлиняет уже построенную последовательность на один элемент;
- очередной элемент последовательности можно определить двумя способами (пустой или элемент базового множества), следовательно каждый вызов процедуры, содержит еще два вызова.

Полученное решение, содержит существенный недостаток, исправить который предоставляется вам. А именно: выборка, даже короткая сопоставляется с максимально длинной последовательностью длины N. Это означает, что часть последовательности (хвост) состоящий из одних пробелов не несет в себе информации о выборке, но тем не менее этот хвост будет добросовестно построен.

Задача 32. О гвоздях и деревянной рейке.

В длинную деревянную рейку забили N гвоздей, можно считать, что гвозди расположены на одной линии. Гвозди объединяются в пары веревочками так, чтобы выполнились следующие условия:

- к каждому гвоздю привязана хотя бы одна веревочка;
- сумма длин веревочек минимальна.

Идея решения:

Для решения задачи используем метод перебора всех возможных вариантов. Очевидно, что если в рейку вбито некоторое количество гвоздей, то количество промежутков между гвоздями (а следовательно и веревочек) будет на единицу меньше. Кроме длины веревочки для решения важно знать, есть ли в данном промежутке веревочка или нет.

Пусть в рейку вбито M гвоздей, тогда максимально возможное количество привязанных веревочек = M-1. Рейку удобно представить в виде двумерной таблицы размерностью 2 × (M-1). Элементы первой строки этой таблицы содержат длины промежутков между гвоздями, а элементы второй строки показывают наличие или отсутствие веревочки в промежутке с соответствующим номером. Если веревочка в промежутке есть, то соответствующий этому промежутку элемент второй строки равен 1, если нет то 0.

Необходимо перебрать все варианты расположения веревочек. Для этого можно представить вторую строку как двоичное число (1 – веревочка есть, 0 – веревочки нет), тогда задача сведется к перебору всех таких чисел длины (M-1). Перебор в нашем случае это процесс сложения двоичного числа с единицей (по правилам двоичной арифметики) работающий до тех пор, пока все разряды двоичного числа не окажутся заполненными единицами. Прием уже использованный в задаче о получении выборок.

А алгоритм можно построить как цикл перебора двоичных чисел, начиная с нулевого, на каждом шаге которого будет проверяться, подходит ли данное расположение веревочек к условию задачи. Для чего достаточно убедиться, что первая и последняя цифры двоичного числа не равны 0 (т.е. к первому и последнему

гвоздям привязаны веревочки) и что в записи числа не содержатся два идущих подряд нуля (т.е. нет гвоздей, к которым ничего не привязано). Если расположение веревочек удовлетворяет условию задачи, то остается только посчитать сумму длин тех промежутков, где есть веревочки и сравнить эту сумму с уже найденным наименьшим. В качестве первого значения наименьшего можно использовать сумму длин всех промежутков между гвоздями. Если длина веревочек при рассматриваемой расстановке меньше MIN, то необходимо сохранить этот вариант положения веревочек в дополнительный массив, а значение MIN заменить на новое. Таким образом, в конце перебора MIN будет равно минимальной удовлетворяющей условию сумме длин веревочек, а в дополнительном массиве будет сохранено расположение веревочек, соответствующее этой сумме.

Задача 33. Построение арифметического выражения.

Дано целое число M. Вставить между некоторыми цифрами 1 2 3 4 5 6 7 8 9 расположенными именно в таком порядке знаки «+» или «-» так чтобы результатом получившегося выражения было число M. Если это невозможно, то выдать соответствующее сообщение.

Идея решения:

Решение задачи осуществляется методом перебора вариантов расстановки знаков между цифрами. Для организации такого перебора условимся, что 0 – это отсутствие знака между цифрами, 1 – знак «+», 2 – знак «-».

В выражении может быть максимум 8 знаков, схему их расстановки представим как троичное число длиной в 8 цифр, каждая из которых обозначает знак (или его отсутствие) между цифрами исходного выражения. Самая левая цифра этого числа обозначает знак между первыми двумя цифрами (1 и 2) и так далее слева направо.

Чтобы перебрать все троичные числа заданной длины, достаточно организовать циклический процесс прибавления к данному числу единицы по правилам троичной арифметики до тех пор пока не наступит ситуация переполнения (больше нельзя прибавить единицу без увеличения количества разрядов).

На каждом шаге этого циклического процесса требуется проверить, подходит ли данная схема расстановки знаков для выполнения условия задания или нет. Проверку можно осуществить следующим образом:

Запишем исходную расстановку в виде числа L = 123456789. В процессе перебора цифр троичного числа увеличиваем некий Счетчик на 1, если троичная цифра равна 0 (т.е. знака нет). Если цифра троичного числа равна 1 или 2, то выполняются следующие операции:

1. Вычисляем остаток от деления L на 10 в степени, равной Счетчику. Таким образом, из L выделяется число, которое будет стоять после проверяемого знака в итоговом арифметическом выражении.
2. Выделенное число прибавляется (или вычитается, в зависимости от знака) к общей сумме (первоначально она равна 0). Кроме того, полученное число необходимо вычеркнуть из L. Для этого достаточно целочисленно поделить L на вышеозначенную степень десятки.

Также нельзя забывать, что после последнего шага цикла останется одно неучтенное слагаемое (т.к. количество цифр больше количества знаков), поэтому в конце его необходимо прибавить к сумме. Ясно, что это будет результат последнего целочисленного деления L (см. выше).

После прохождения цикла необходимо проверить, равна ли полученная сумма введенному числу. Если да, то остается только вывести текущий вариант расстановки знаков.

Задача 34. Расстановка ферзей.

На шахматной доске расставить восемь ферзей так, чтобы ни один из них не оказался под боем.

Идея решения

Решение задачи осуществляется методом перебора всех возможных вариантов расстановки ферзей и поиска искомого варианта. Для этого условимся, что 0 – это свободная и не битая каким-либо ферзем клетка поля, 1 – клетка, в которой стоит ферзь, 2 – клетка, битая хотя бы одним ферзем. Шахматную доску можно представить двумерным массивом размерностью 8x8.

В общем виде идея решения такова: необходимо поместить ферзя в первой клетке первой горизонтали (т.е данному элементу массива присвоить значение 1), затем отметить все битые этим ферзем клетки (т.е. клетки расположенные на одной горизонтали, вертикали и диагонали с ферзем отметить в массиве двойками). Далее переходим к следующей горизонтали, и в первой (с левого края горизонтали) пустой и небитой клетке ставим очередного ферзя. Для него также нужно осуществить процедуру отметки битых им полей. И так далее для всех горизонталей. Таким образом, наступит один из двух вариантов развития событий: либо будет установлен восьмой ферзь в последнюю горизонталь (в таком случае, данный вариант является одним из искомых решений), либо будет установлено меньшее количество ферзей, но на поле не останется небитых клеток.

В случае неудачи необходимо убрать последнего ферзя, вернуться на предыдущую горизонталь и найти следующую свободную клетку на этой горизонтали. Если таковая присутствует, то следует поставить туда ферзя и отметить битые им поля, перейти на следующую горизонталь и снова попытаться расставить ферзей в свободные клетки вплоть до последней горизонтали. Если же свободных клеток на данной горизонтали больше нет, то необходимо возвращаться на предыдущие горизонтали и убирать с них ферзей до тех пор, пока на какой либо из них не обнаружится свободная непроверенная клетка.

Конечно же убирая ферзя, необходимо пересчитывать битые поля. Просто убрать с поля двойку нельзя, так как поле может быть битым несколькими ферзями. Вести полный пересчет всех полей для каждого уже установленного ферзя нерационально. Это может потребовать очень большого объема перерасчета. В качестве идеи решения этой проблемы можем посоветовать вам ввести для каждого поля доски числовой коэффициент запоминающий сколькими ферзями данное поле бито.

Задача становится сложнее, если потребовать получение не одного решения, а всех возможных. В таком случае она становится задачей полного перебора.

И с учетом количества возможных расстановок ферзей, полным перебором задача практически не решаема. Однако очень часто специальной организацией перебора от полного перебора можно уйти. Существует два общих подхода к решению этой проблемы. Во-первых, это отсечение вариантов, метод нами в решении уже использованный, отметка битых полей, ведет к отсечению всех вариантов использующих битые поля. Второй подход основан на сравнивании дерева вариантов и отсечении одинаковых ветвей. Этот подход называется склеиванием ветвей и опирается на возможные симметрии в позициях. Например для задачи о ферзях можно утверждать, что зеркальное отражение и поворот на 90 градусов дадут новые варианты. Возможны и другие виды симметрии. Поищите их самостоятельно. Еще одна серьезная техническая проблема – это проблема учета симметрий внутри общего процесса перебора.

Задача 35. Разложение натурального числа.

Дано множество натуральных чисел и число N. Определить, возможно ли представить N в виде суммы элементов данного множества, при условии, что каждый элемент может входить в сумму не более одного раза.

Идея решения:

Вариант 1: Простой.

Простое лобовое решение заключается в:

- переборе всех последовательных чисел от 1 до тех пор пока не будет найдено искомое;
- для каждого очередного числа строятся все возможные выборки из заданного множества. Для каждой выборки считается сумма ее элементов, и выясняется равна эта сумма заданному числу или нет. Задача построения всех выборок уже была рассмотрена – это задача 2.

Решение задачи посредством построения выборок можно существенно усилить. Предположим, что есть некоторая выборка, сумма элементов которой уже больше заданного числа. Это означает, что строить выборки, содержащие данную выборку, в качестве составной части нет смысла.

Вариант 2: Более эффективный.

Проблема предыдущего варианта в том, что одни и те же выборки составляются многократно, для каждого числа из натурального ряда, что очень не эффективно. Единственно, что может спасти ситуацию от большого количества повторных выборок, это ранее обнаружение необходимого числа. Предлагаемое усиление несколько облегчает проблему, но не радикально, кроме того это улучшение существенно усложняет логику.

Более интересный вариант заключается в одноразовом построении выборок. Если построить все выборки, и их суммами заполнить некоторый массив, предварительно инициализированный нулями, то после, достаточно найти первый ноль в массиве, индекс найденного нуля и будет искомым числом.

Ясно, что нет необходимости в большом массиве для выборок, так как суммы очевидно будут часто повторяться.

Такую стратегию можно даже существенно улучшить. Упорядочим массив исходного множества в порядке возрастания. Если выборки строить так как описано в задаче 2, то очевидно суммы также будут получены в некотором порядке. Каков этот порядок и что он дает для решения задачи исследуйте самостоятельно.

Задача 36. О арифметической прогрессии.

Заполнить целочисленный массив длины N числами интервала 1..N, так чтобы никакие три элемента идущие в порядке возрастания индексов не образовывали бы арифметическую прогрессию.

Идея решения:

Задача имеет простое переборное решение. Заполним массив любым образом, затем запустим процесс получения перестановок. Алгоритм получения перестановок рассмотрен в неформальном введении (задача 40, листинг 95). И для каждой перестановки проверим выполнимость условия задачи. Если задача решаема, то правильная перестановка обязательно будет обнаружена и более того, метод перестановок позволит обнаружить все существующие решения. Проблема заключается в очень быстром росте числа перестановок. Для N – элементов исходного множества, количество перестановок выражается числом N! которое, с ростом N очень быстро становится астрономическим. Поэтому даже для не очень больших значений N, формально решаемая задача становится фактически неразрешимой (нельзя получить решение за приемлемое время).

Перебор иногда можно ограничить, но зачастую справиться с огромным количеством вариантов перебора все же не удается. В таких случаях можно попытаться построить решение с нужными свойствами. Общий подход таков:

- построим решение при ограниченном количестве элементов исходного множества;
- определим, как зная решение для N элементов перейти к решению для N + 1 элемента.

В нашем случае элементарной ситуацией будет множество из трех чисел 1..3. Для данного интервала возможно следующее решение: 1 3 2. Оно не единственное, подумайте что означает для решения задачи множественность решения для простого множества.

Переход к следующему множеству это расширение интервала на один элемент. В нашем случае это означает добавление числа 4. Перестройка решения будет заключаться в добавлении числа 4 в некоторую позицию ряда без изменения взаимного положения уже расставленных чисел, так как только так можно гарантировать не возникновение новых троек составляющих прогрессию.

Задача 37. Путешествие коня по доске.

Построить путь обхода конем шахматной доски по следующим правилам:

- конь начинает свой путь из левого нижнего угла;
- в каждой клетке конь бывает ровно по одному разу;
- путь заканчивается в той же клетке откуда и начался.

Идея решения:

Все множество путей коня по доске можно представить в виде дерева с ветвями разной длины. Таким образом, задача сводится к задаче обхода дерева и поиску ветки с определенным свойством (завершение ветки в левом нижнем углу при полном обходе доски). Специфика задачи только лишь в особенном способе его построения. Следовательно, решение задачи сводится к описанию способа построения дерева ходов.

- запишем в каждой клетке доски ноль. Ноль означает, что поле пока не рассматривалось;
- запишем в исходном поле единицу. Назовем текущим полем, поле в котором находится конь после очередного хода. До первого хода текущее поле это левое нижнее;
- далее после каждого хода, выбираем любое поле на которое может попасть конь из текущего и имеющее значение ноль или единица. Если выбранное поле имеет значение единица, значит конь вернулся на исходное поле;
- переводим коня на выбранное поле, оно становится текущим и его нулевое значение заменяется на значение предыдущего поля увеличенного на единицу;
- если с текущего поля нет возможности сделать ход то выполняется процедура возврата, смысл которой в поиске поля на уже пройденном пути с которого можно продолжить построение пути.

Процедура возврата:

Процедура возврата заключается в следующих действиях:

- ищется поле, чье значение на единицу меньше значения поля текущего. Это и есть поле с которого конь попал на текущее;
- в текущее поле записывается -1, для того, чтобы полностью убрать поле из дальнейшего рассмотрения;
- найденное поле становится текущим.

Примечание. Уже было сказано, что решением будет путь, заканчивающийся на исходном поле. Но конь может придти в исходное поле и при этом не пройти всю доску. Критерий завершения можно уточнить двумя способами:

- путь завершен на исходном поле и ни одно поле доски не имеет нулевого значения;
- путь завершен на исходном поле и длина пути равна 64. Длина уже пройденного пути в нашем алгоритме это значение текущего поля.

Описанные действия выполняются до тех пор, пока не будет найдено поле, с которого можно выполнить ход на поле с нулевым значением. Если в процессе возврата конь попадет на поле со значением 1 и при этом не будет возможности выполнить ход на нулевое поле и на доске еще остаются нулевые поля, то это будет означать отсутствие решения (точнее это будет означать ошибку, так как известно, что решение существует).

Заметим, что как и в задаче о ферзях, мы использовали идею отсечения вариантов, выбрасывая из рассмотрения поля уже отмеченные, как тупиковые.

Принципиально, как и задача о ферзях, данная задача является переборной, и как и задача о ферзях, полным перебором, практически не решаемая.

Возможно, еще одно улучшение называемое правилом Варнсдорфа. Заключается оно в том, что при каждом выборе очередного хода, выбирается поле, с которого можно выполнить наименьшее количество ходов на следующем шаге. Использование правила ведет к раннему отсечению тупиков и увеличению вероятности быстрого обнаружения требуемого пути.

Задача 38. Тройки точек.

На плоскости задано некоторое количество точек парой координат. Найти количество троек точек лежащих на одной прямой.

Идея решения:

Для не очень большого множества точек задача решается перебором и сводится к построению всех возможных выборок по 3 точки. Количество таких выборок для сотни точек равно 161700. Для каждой полученной выборки достаточно проверить выполнимость условия задачи. Количество выборок можно резко сократить, если перебирать только пары точек. Таких выборок, для 100 точек, согласно той же формуле всего лишь 4950. Далее, для каждой выборки необходимо найти все точки лежащие с двумя данными на одной прямой и из них уже ничего не проверяя составить все возможные тройки.

Напомним, что координаты точек лежащих на одной прямой представляют собой простую пропорцию.

Количество выборок можно существенно сократить. Для этого в полученной схеме есть по крайней мере две возможности:

- предположим, что для некоторой пары точек было построено множество точек лежащих с данной парой на одной прямой. Это означает, что все пары точек принадлежащих данному множеству из дальнейшего рассмотрения можно исключить. Но точки данного множества вполне могут участвовать в парах с точками ему не принадлежащими;
- пусть опять, для некоторой пары было построено множество точек (назовем его очередным) лежащих с ними на одной прямой. И пусть теперь для анализа из исходного множества взяты две точки, не принадлежащие очередному множеству. Назовем множество точек лежащих на одной прямой с новой парой точек новым множеством. Очевидно, что **очередное** множество и **новое** множество если и пересекаются, то только в одной точке.

Задача 39. Последовательность возрастающих чисел.

Дана последовательность целых чисел, среди которых нет двух одинаковых. Найти минимальное количество чисел, таких что после их вычеркивания последовательность превратится в строго возрастающую.

Идея решения:

Предположим, что искомые числа уже найдены, вычеркнуты и строго возрастающая последовательность получена. Ясно, что во-первых, эта последовательность

уже существовала в исходной и во-вторых, найденная последовательность является наибольшей возрастающей подпоследовательностью. Поэтому задача может быть сведена к поиску наибольшей возрастающей подпоследовательности.

Каждая подпоследовательность имеет своим началом одну из точек исходной последовательности. Можно сказать, что числа исходной последовательности разбивают множество подпоследовательностей на классы. Класс – это множество подпоследовательностей имеющих общее начало. Таким образом необходимо организовать перебор по всем элементам всех классов подпоследовательностей.

Фактически, мы имеем задачу полного перебора, которую опять таки можно немного оптимизировать. Способ оптимизации следующий. Пусть на некоторый момент найдена возрастающая подпоследовательность длины Max. Тогда из рассмотрения можно выбросить все точки последовательности, справа от которых элементов исходной последовательности меньше чем Max.

Раздел Д. Графы

Задача 40. Самый длинный путь рубки.

На стандартном поле 8×8 расставлено, некоторое количество черных шашек и одна белая. Найти самый длинный путь рубки для белой шашки.

Идея решения:

Путь рубки можно представить как дерево, в котором узлами являются срубленные шашки и из каждого узла выходит не более трех веток. Если удастся создать такое дерево, то решение задачи сведется к вычислению глубины дерева, для чего необходимо организовать процедуру обхода (можно рекурсивную) возвращающую длину при достижении тупикового узла (тупик – это узел из которого не выходит ни одной ветки).

Задача 41. Волновой алгоритм.

Формулировка задачи. Дан непустой, связный граф. Необходимо найти путь между двумя вершинами содержащий наименьшее количество ребер.

Идея алгоритма:

Перед началом работы алгоритма каждой вершине присваивается число называемое волновой меткой и имеющее минимальное значение (например 0). Работа алгоритма заключается в распространении фронта волны от исходной вершины по всему графу, либо до полного затухания волны либо до достижения вершины назначения. В процессе движения волны значения волновых меток изменяются по определенному правилу позволяющему определять длину пути до каждой вершины, через которую прошла волна. Правило заключается в увеличении значения волновой метки вершины в момент прохождения волны через нее. При этом значение приращения волновых меток зависит от длины пути пройденного волной. Таким образом, можно утверждать, что чем больше значение волновой метки тем больший путь был пройден волной до данной вершины.

Для полного понимания идеи внимательно прочитайте примечание данное после текста алгоритма.

Текст алгоритма:

Метка исходной точки = 0
Смещение = 0
Фронт волны состоит только из исходной точки
Новый фронт пуст
Повторять
 Для всех ВЕРШИН Фронта
  Для всех вершин смежных ВЕРШИНЕ
   Если метка = -1 ТО метка = Смещение + 1 и вершина включается в список вершин Нового фронта
  Фронт = Новый фронт
  Смещение = Смещение +1
Если Новый фронт пуст то решение получено.

Примечания:

- заметим, что алгоритм обеспечивает проход всего графа, это немного больше чем требовалось в условии, алгоритм фактически находит кратчайшие пути от исходной точки, до всех точек графа. Граф при этом предполагается связным. Для несвязного графа алгоритм то же будет работать, но точки не связанные никаким путем с исходной вершиной выпадут из рассмотрения, на процесс обработки остальных вершин, несвязные компоненты не окажут никакого влияния;
- важно заметить, что значение метки меняется только в том случае если волна в нее пришла первый раз. Это дает возможность сохранять пути к вершине назначения в чистом виде. Но конечно эта особенность делает механизм распространения волны совершенно не похожим на распространение физической волны.

После завершения работы алгоритма искомый путь восстанавливается следующим алгоритмом:

Очередная вершина есть вершина назначения.
Пока текущая вершина не есть исходная делать
 Новой Очередной вершиной становится вершина чье значение метки на единицу меньше значения метки Очередной вершины.

Обоснование алгоритма:

Возьмем любую вершину графа. Предположим волна в эту вершину может придти из двух вершин, назовем их A и B. Предположим, что путь от вершины B проходит через две вершины, а путь от вершины A через одну. Из этого следует, что волна от вершины A придет за один шаг работы алгоритма, а от вершины B через два. И следовательно в данную вершину волна придет из вершины A. Это утверждение легко обобщить на путь произвольной длины. А это и означает, что волна идет по кратчайшему пути. Кроме того из того, что высота волны с каждым шагом растет на 1 следует корректность правила восстановления пути.

Пример:

Исходное состояние. Ниже (рис. 3.7) показано исходное состояние графа, который подлежит исследованию волновым алгоритмом. Источник волны самая левая вершина. Наша задача построить все кратчайшие пути от нее до всех остальных вершин. В качестве теста попробуем определить кратчайший путь до крайней правой вершины. В начале движения волны ее высота равна нулю, поэтому исходная вершина помечена нулем. Метки остальных вершин также будем считать нулями.

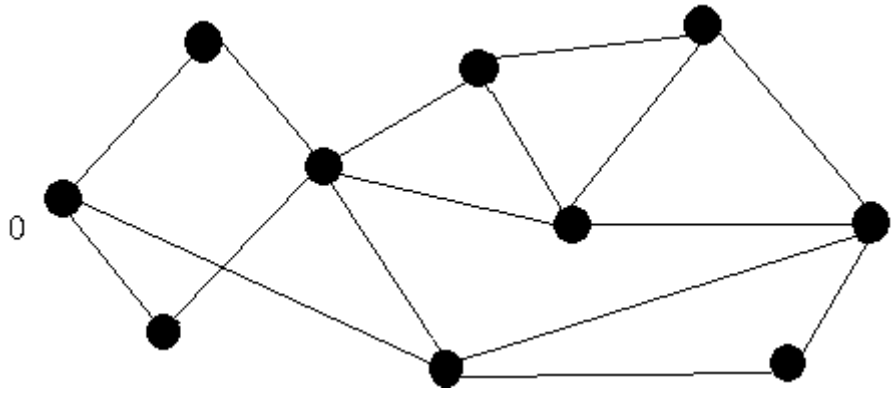

Рис. 3.7. Исходный граф

Шаг 1. Исходная вершина соединена ребрами с тремя другими вершинами графа, поэтому на данном шаге три вершины помечаем единицами (Рис 3.8).

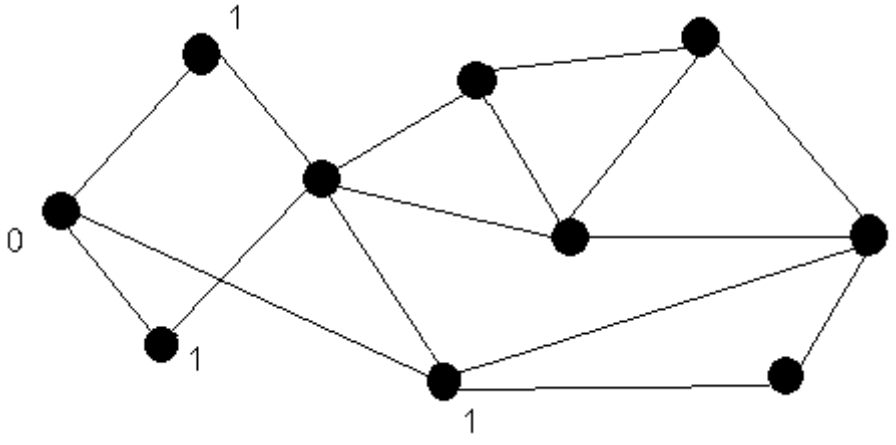

Рис. 3.8. Шаг 1

Шаг 2. На данном шаге три вершины отправляют волну с высотой два в те вершины которые соединены с ними ребрами и никак пока не отмечены, то есть нули (рис. 3.9).

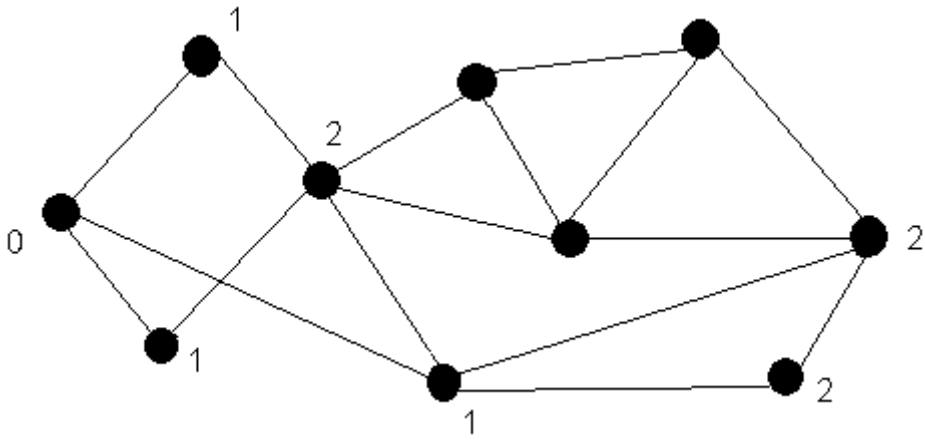

Рис 3.9. Шаг 2

Шаг 3. Это последний шаг. Вершины, содержащие двойки посылают волну с высотой 3 в неотмечененные, после чего неотмеченных вершин уже не остается. Теперь легко восстановить путь из любой вершины до исходной. Для нашей тестовой, крайней правой вершины до исходной путь отмечен жирной линией. Что короче путь невозможен, очевидно.

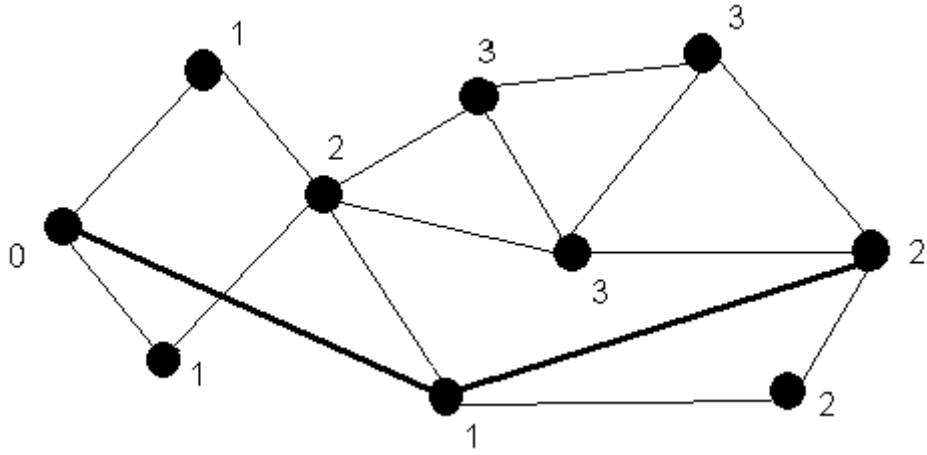

Рис. 3.10. Шаг 3

Еще одно важное примечание. Кратчайший путь не обязательно единственный. Если кратчайших окажется несколько, волновой алгоритм обнаружит их все.

Задача 42. Алгоритм поиска компонент связности.

Формулировка задачи:

Дан произвольный граф возможно состоящий из нескольких компонент связности. И дана произвольная вершина графа. Требуется выделить компоненту связности, содержащую данную вершину.

Идея алгоритма:

Предположим, про некоторое количество вершин уже известно, что они принадлежат искомой компоненте связности. Присвоим этим вершинам некоторое число, будем называть его статусом и договоримся, что для вершин принадлежащих компоненте статус имеет вполне определенное значение. Тогда расширить множество вершин принадлежащих компоненте можно простой процедурой:

- возьмем любую вершину, чей статус говорит о том, что вершина принадлежит компоненте связности;
- передадим ее статус всем вершинам, соединенным с ней ребром.

Алгоритм:

Присвоим всем вершинам статус 1.
Присвоим ИСХОДНОЙ вершине статус 2.
Пока есть вершины имеющие статус = 2
 Выберем любую вершину имеющую статус 2 назовем ее ТЕКУЩЕЙ
 Для всех вершин смежных ТЕКУЩЕЙ
  Если вершина имеет статус 1 то ее статус = 2
 Статус ТЕКУЩЕЙ вершины = 3

Совершенно очевидно, что по завершению работы данного алгоритма все вершины имеющие статус 3 будут принадлежать к той же компоненте связности что и исходная вершина, точно также можно утверждать, что вершины имеющие статус = 3 полностью исчерпывают множество вершин данной компоненты связности.

Пример:

На рис. 3.11 ниже показан пример графа на котором мы продемонстрируем работу описанного алгоритма.

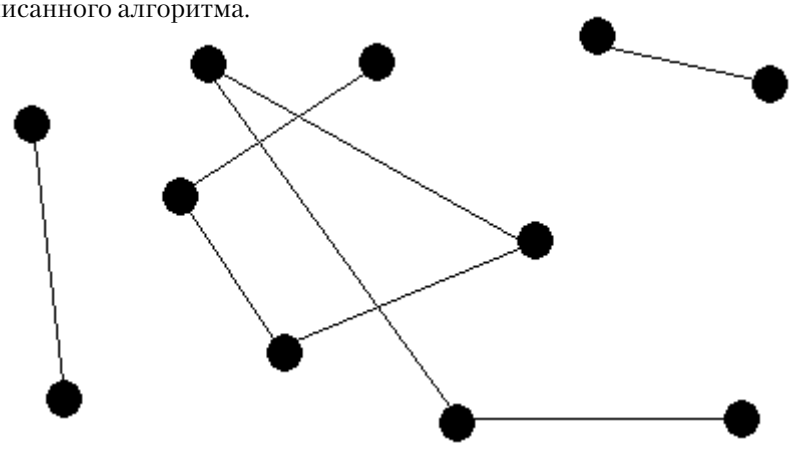

Рис. 3.11. Исходное состояние графа

Результат первого шага работы (Рис. 3.12) это присвоение статуса 1 всем вершинам графа, так как на этом этапе работы ни про одну из них не известно, о принадлежности искомой компоненте связности.

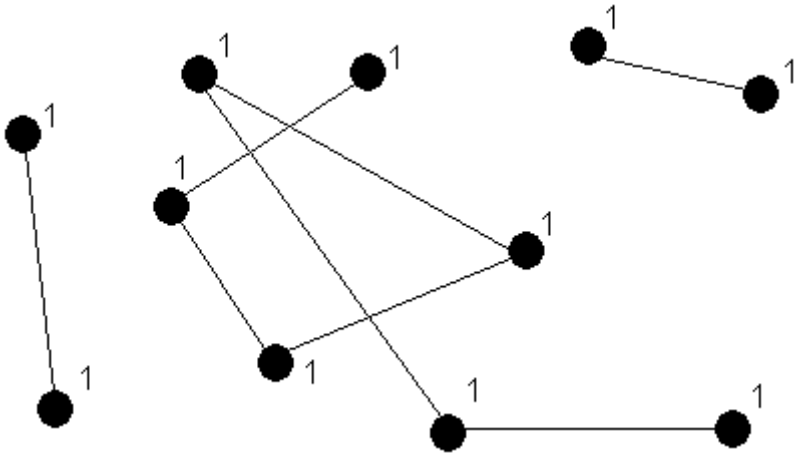

Рис. 3.12. Первый шаг

Второй шаг (Рис. 3.13) отражает тот факт, что про одну из вершин (ИСХОДНУЮ) известно о ее принадлежности к компоненте, поэтому ей присвоен статус 2.

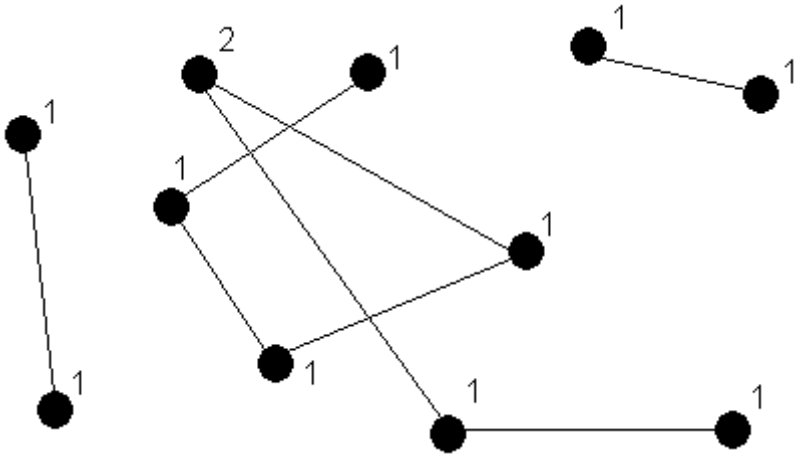

Рис. 3.13. Второй шаг

Начиная с рис. 3.14 иллюстрируется работа цикла расширяющего множество вершин включенных в компоненту связности. На первом шаге работы ИСХОДНАЯ вершина получает статус 3 и выбывает из обработки, а две ей смежные приобретают статус 2.

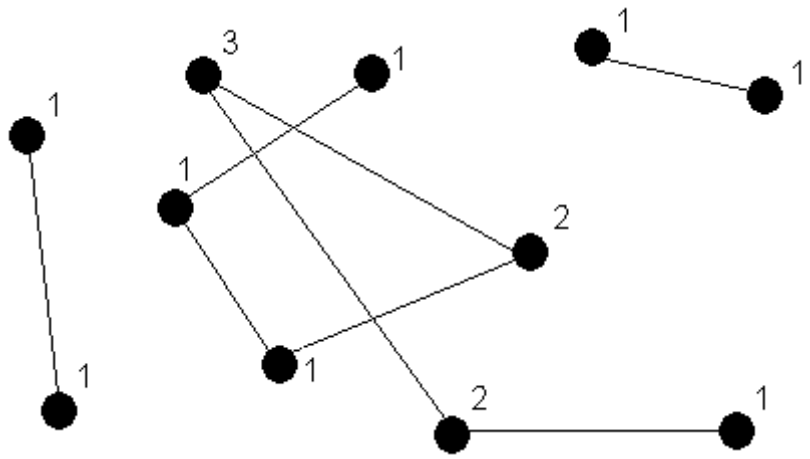

Рис. 3.14. Третий шаг

Цикл обработки продолжает свою деятельность и на очередном шаге еще одна вершина получает статус 3, соответственно ее соседи получают статус 2 (Рис. 3.15).

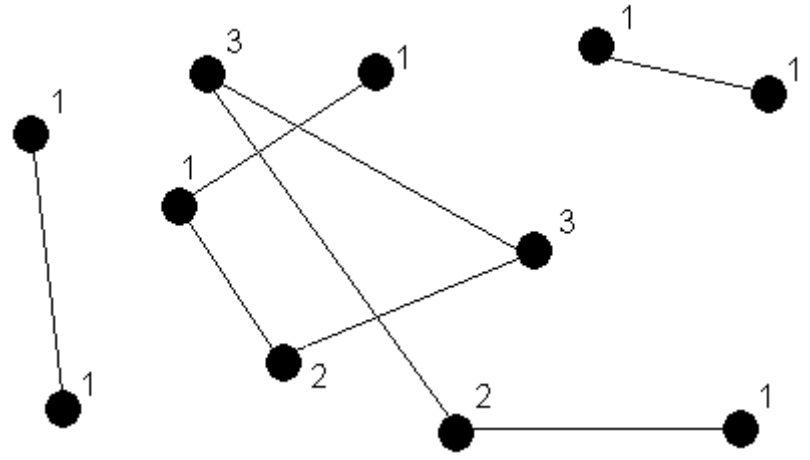

Рис. 3.15. Четвертый шаг

Думается процесс распределения статусов вполне понятен, поэтому оставшиеся два рисунка оставим без комментариев.

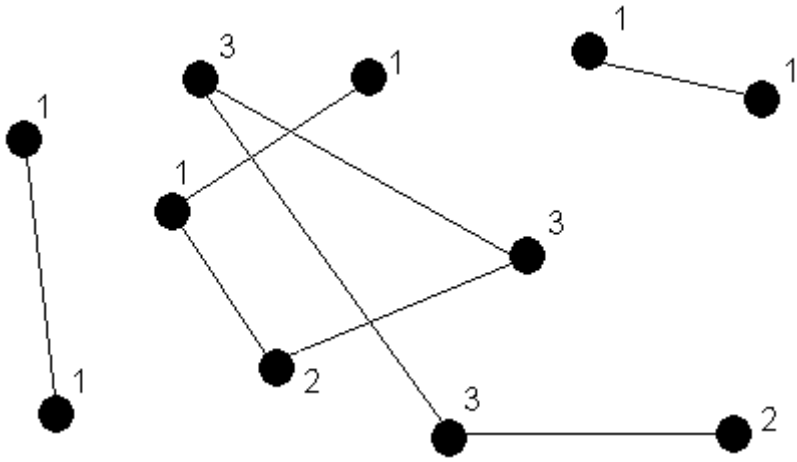

Рис. 3.16. Пятый шаг

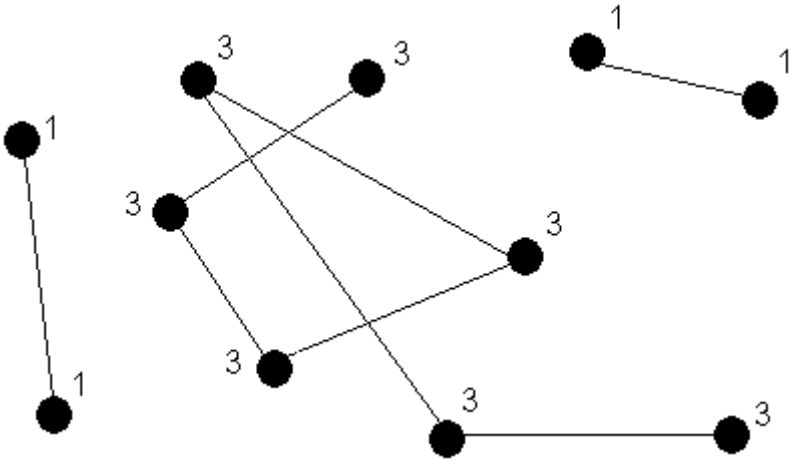

Рис. 3.17. Результат

Примечание. По завершению алгоритма мы гарантированно получили одну компоненту связности, все вершины которой отмечены числом 3. Про вершины отмеченные единицами по прежнему нельзя сказать ничего определенного.

Задача 43. Кратчайший путь в лабиринте.

Задан лабиринт с одним входом и одним выходом. Найти кратчайший путь.

Идея решения:

Представим себе лабиринт в виде решетки, некоторые из вершин которой являются доступными, а некоторые нет. Тогда путь по лабиринту, это переход от вершины к соседней вершине, по горизонтали или вертикали. Отбросим все недоступные вершины, тогда множество доступных вершин с отрезками их соединяющими будет представлять собой неориентированный граф, кратчайший путь по которому можно найти волновым алгоритмом.

Задача 44. Система шестеренок.

На плоскости задана система из N одинаковых сцепленных шестеренок. Схема сцепления задается. Система приводится во вращение одной шестеренкой, которую будем для упрощения называть первой. Необходимо для заданной системы шестеренок определить будет ли она вращаться или ее заклинит.

Идея решения:

Шестеренка может крутиться только в двух направлениях. Будем одно из них называть первым (обозначать 1), а другое вторым (обозначать 2). Какое из них считать первым, а какое вторым, конечно же не более чем условность. Существенно важно лишь то, что при передаче движения от шестеренки к шестеренке направление движения меняется на противоположное.

Пусть направление, в котором крутится первая шестеренка обозначено, как первое. Далее необходимо построить модель передачи движения. Для этого построим граф, вершины которого – шестеренки, ребра – соединения между ними, а направление кручения характеристика принимающая значения 1, 2, 0 (0 – шестеренка пока не крутится).

Процесс передачи кручения в системе шестеренок тогда можно рассматривать, как процесс передачи характеристики между вершинами графа.

- начинается процесс передачи характеристики с начального положения в котором вершина принятая за первую имеет характеристику 1, а характеристики остальных вершин графа есть ноль;
- все вершины графа разбиваются на три класса: A – получившие характеристику и передавшие ее дальше; B – получившие характеристику, но не передавшие ее; C – все остальные вершины. На первом шаге анализа класс A пуст, класс B состоит из первой вершины, класс C – все остальные вершины графа;
- на каждом шаге процесса каждая вершина класса B передает характеристику всем вершинам из класса C связанным с нею;
- при этом вершина передавшая характеристику перемещается в класс A, а вершина получившая характеристику перемещается в класс B;
- если передаваемая характеристика 1 то получаемая 2 и наоборот;
- система считается заклинившей если вершина передающая характеристику является смежной с вершиной имеющей такую же характеристику.

Задача 45. Экономный обход графа.

Задано двоичное дерево. Необходимо совершить полный обход, используя минимальное количество дополнительных структур данных.

Идея решения:

Принципиально обход двоичного дерева не представляет собой проблемы. В неформальном введении мы рассмотрели рекурсивное решение этой задачи. Но рекурсия для своей реализации требует немалых ресурсов памяти, поэтому данная постановка задачи требует отказа от рекурсивных решений. Рассмотрим нерекурсивный алгоритм позволяющий выполнить обход с использованием только одного дополнительного битового поля (алгоритм Дойча).

Обозначим ветви входящие в вершину, как ИСТОЧНИК, ЛЕВАЯ и ПРАВАЯ (ветвь это указатель на вершину, соответственно ЛЕВАЯ указывает на дочернюю левую вершину, и ПРАВАЯ на дочернюю правую вершину). Ветвь ИСТОЧНИК соединяет данную вершину с вершиной предком, а ветви ЛЕВАЯ и ПРАВАЯ это соответственно ветви идущие вглубь дерева. Проблема организации обхода, очевидно, заключается в моментах возврата. Вернувшись в вершину, мы можем встретиться с двумя принципиально разными ситуациями. Во-первых, может оказаться, что из данной вершины возможен путь вглубь по другой ветке и во-вторых, может оказаться, что путь вглубь и по ветви ЛЕВАЯ и по ветви ПРАВАЯ невозможен (уже пройдены) и необходимо выполнить процедуру возврата. То есть, процедура возврата может либо породить еще одну процедуру возврата либо процедуру движения вглубь.

Для того, чтобы не ошибиться в принятии решения нужна дополнительная информация, о том, что было сделано в момент предыдущего вхождения в вершину. Запоминанием такой информации и занимается механизм рекурсии.

Отправной точкой алгоритма Дойча являются три очевидных утверждения. Во-первых, каждая вершина хранит информацию о трех указанных ветвях и во-вторых, возврат в каждую вершину двоичного дерева возможен только дважды и в третьих, по каждой ветви дерева движение будет выполняться только дважды: при движении вглубь и в момент возврата.

А теперь собственно идея алгоритма Дойча

Введем для каждой вершины одно дополнительное битовое поле (назовем его ФЛАГ) инициализированное нулем. При первом входе в вершину мы обнаруживаем ФЛАГ равный нулю. Это является сигналом, что возможно движение по ветви ЛЕВАЯ. Уходим по ветви ЛЕВАЯ и так как движение по ней из данной вершины уже невозможно, то используем ветвь ЛЕВАЯ для запоминания ветви ИСТОЧНИК.

Если мы обнаружили ФЛАГ = 0 выполняя возврат, то это означает, что ветвь ЛЕВАЯ уже пройдена и сейчас она фактически хранит информацию о ветви ИСТОЧНИК. Тогда уходим вглубь по ветви ПРАВАЯ, значение ФЛАГА меняем на 1. И сейчас ветвь ИСТОЧНИК можно запомнить в ветви ПРАВАЯ.

Если же выполняя возврат, мы обнаруживаем значение ФЛАГА = 1 это означает, что обе ветви ведущие вглубь уже пройдены и необходимо выполнить процедуру возврата еще раз, а информация для этого возврата хранится в ветви ПРАВАЯ.

Описанная идея содержит один существенный недостаток. Запоминая в ЛЕВОЙ (ПРАВОЙ) ветви информацию о ветви ИСТОЧНИК, информацию о ЛЕВОЙ (ПРАВОЙ) ветви мы теряем. Это приводит к тому, что с обходом дерева оно практически уничтожается.

Указанный недостаток легко исправить, если заметить, что ЛЕВАЯ (ПРАВАЯ) ветвь очередного узла, есть ИСТОЧНИК его дочернего, то есть того, в который будет осуществлен переход из текущего. Иначе говоря информация о ЛЕВОЙ (ПРАВОЙ) ветви сохраняется в дочерних узлах и следовательно ее вполне можно восстановить.

Примечание. В тексте часто говорится о ветвях и о том, что одна ветвь запоминается в другой. В терминах ветвей это выглядит достаточно неясно. Эта терминология станет ясной при переходе к указателям и связным спискам. Запомнить ветвь это всего лишь присвоение указателей.

Задача 46. Алгоритм Краскала.

Формулировка задачи:

Дан взвешенный граф, в котором веса присвоены ребрам. Необходимо найти минимальное остовное дерево имеющее своим корнем одну из вершин графа.

Идея алгоритма:

Искомые ребра соединяют вершины. Поэтому возможны две стратегии построения. Можно идти от вершин и для каждой из них искать минимальное ребро (это сделано в алгоритме Прима см. следующую задачу) а можно для каждого ребра выяснять можно ли его включить в строящееся дерево. Алгоритм Краскала ведет построение от ребер и предлагает делать это следующим образом. Во-первых, ребра графа нумеруем в порядке возрастания весов. Затем для каждого ребра начиная с первого проверяем соединяет или нет оно две несвязные компоненты графа, если да, то его можно включить в остовное дерево. Ясно, что если мы имеем V вершин, то работа алгоритма начинается с V несвязных компонент графа (фактически вершин графа). Для того, чтобы их связать необходимо найти V-1 ребро.

Другими словами, алгоритм организует процесс роста компонент связности в ходе которого они объединяются друг с другом до тех пор пока не останется одна являющаяся конечным результатом.

Алгоритм:

```
Создаем список ребер по возрастанию.
Создаем множество компонент связности каждая из которых
содержит ровно одну вершину.
Пока компонент связности больше чем одна:
    Взять очередное ребро из списка ребер.
    Если ребро соединяет две разных компоненты связности
    то компоненты связности объединить в одну.
```

Ниже рассмотрен процесс работы алгоритма. Каждому ребру поставлено в соответствие два числа: вес и в скобках номер. Ребро, включенное в остовное дерево (соединяющее две компоненты связности) выделено жирным, вершины входящие в дерево выделены белыми квадратиками. На рис. 3.18 показана исходная ситуация.

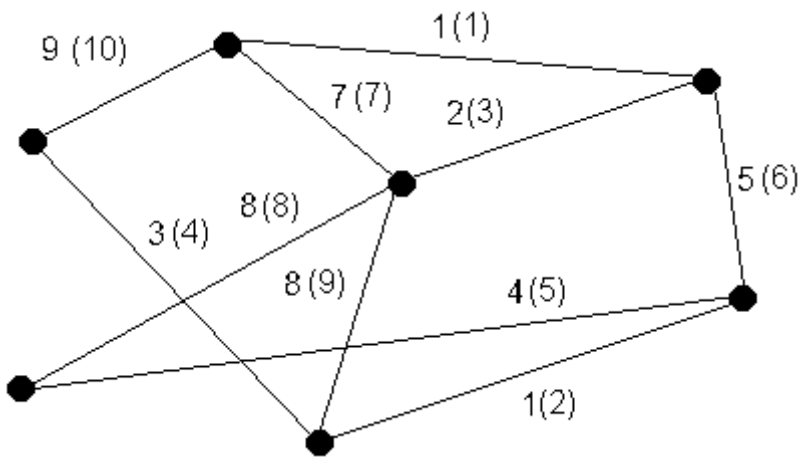

Рис. 3.18. Исходный граф

Шаг 1. Первое в списке ребро соединяет две вершины. Объединяем их в одну компоненту связности (рис. 3.19).

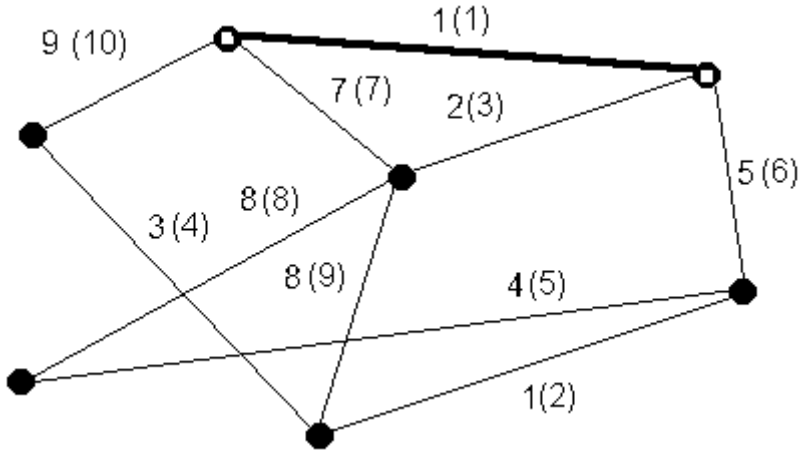

Рис. 3.19. Первый шаг

Шаг 2. Ребро помеченное как второе и также имеющее вес 1 соединяет две вершины. Объединяем их в одну компоненту и получаем две компоненты, включающие в себя по две вершины (рис. 3.20).

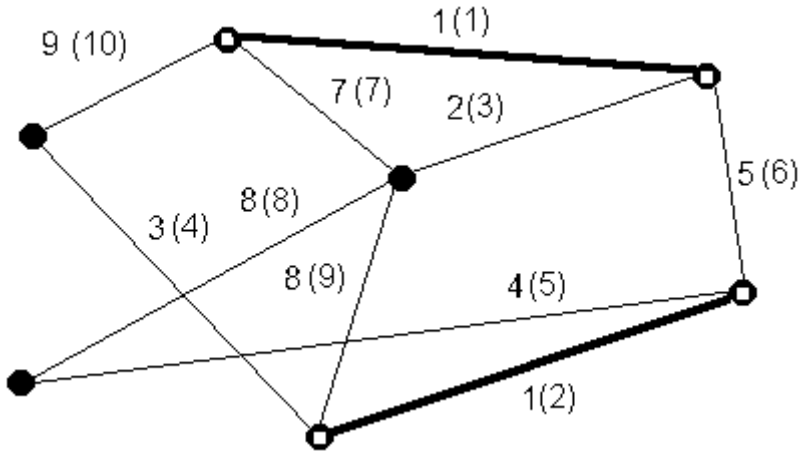

Рис. 3.20. Второй шаг

Шаг 3. Третье ребро имеющее вес 2, соединяет отдельную вершину с компонентой имеющей две вершины и наша первая компонента увеличивается в размере (рис. 3.21).

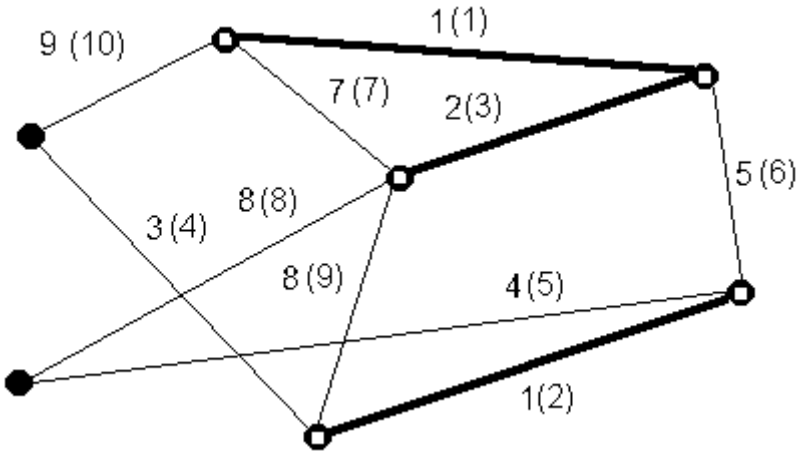

Рис. 3.21. Третий шаг

Шаг 4. Ребро идущее в списке четвертым добавляет вершину второй компоненте связности и теперь мы имеем две компоненты в каждой из которых по три вершины (рис. 3.22).

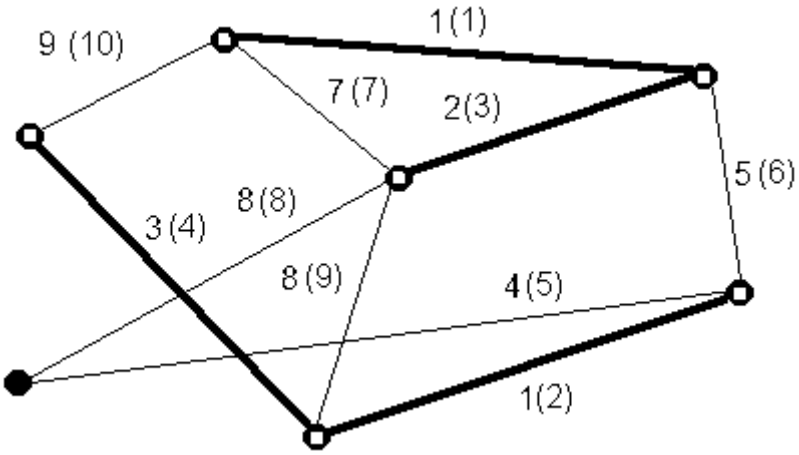

Рис. 3.22. Четвертый шаг

Шаг 5. Следующее ребро добавляет во вторую компоненту еще одну вершину и теперь все вершины распределены между двумя компонентами связности (рис. 3.23).

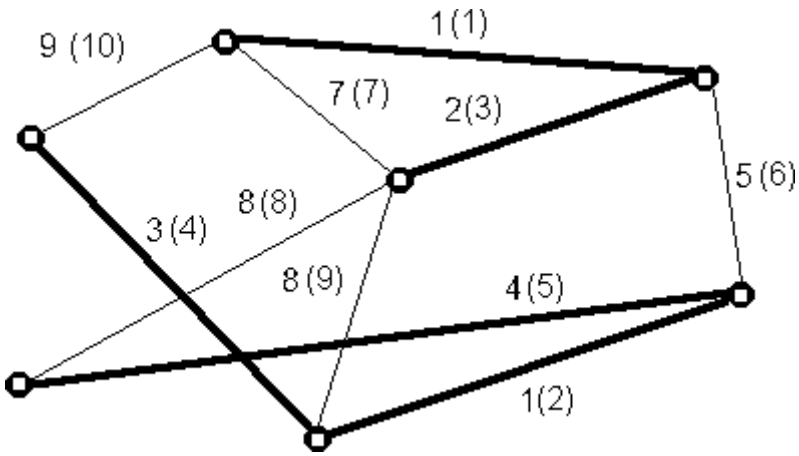

Рис. 3.23. Пятый шаг

Шаг 6 заключительный. Завершает наш процесс ребро с номером 5. Оно объединяет две построенные на предыдущих шагах компоненты связности в одну, которая и есть минимальное остовное дерево (рис. 3.24).

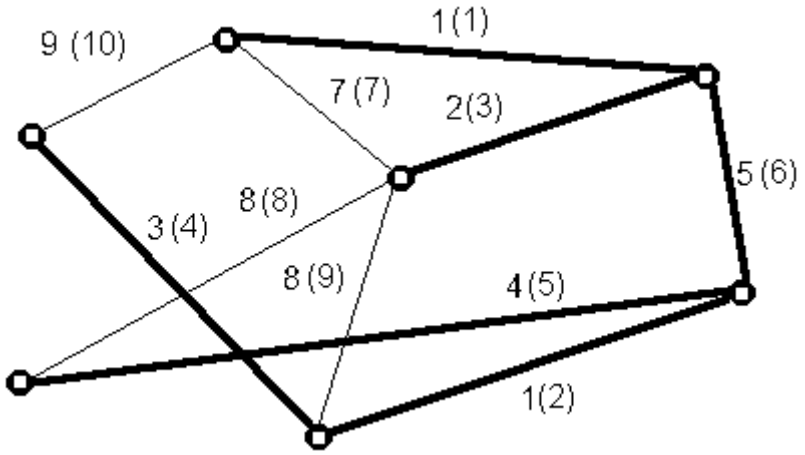

Рис. 3.24. Результат

Задача 47. Алгоритм Прима.

Формулировка задачи:

Дан взвешенный граф, в котором веса присвоены ребрам. Необходимо найти минимальное остовное дерево имеющую своим корнем одну из вершин графа.

Идея алгоритма:

Пусть часть остовного дерева уже построена. Это утверждение всегда верно, так как в начале процесса вершина с которой начинается построение уже входит в дерево. Итак, если часть остовного дерева уже есть, то множество вершин графа можно разделить на два подмножества: подмножество состоящее из вершин уже построенного остовного дерева и оставшихся вершин графа.

Очевидно, что среди ребер соединяющихся эти два множества существует ребро наименьшего веса. Можно доказать, (но мы здесь этого делать не будем) что минимальное дерево проходит через это ребро.

Алгоритм:

Множество остовных вершин – это исходная вершина
Множество оставшихся – все вершины за исключением исходной.
Пока множество оставшихся не пусто
    Ищем ребро соединяющее множество остовных и множество
    оставшихся и имеющее наименьший вес.
    Для найденного ребра, вершину принадлежащую множеству оставшихся:
        Вычеркиваем из множества оставшихся.
        Добавляем к множеству остовных.

Ниже показан процесс работы алгоритма Прима. Исходная точка для построения остовного дерева обозначена как точка A. На каждом шаге работы алгоритм добавляет к остовному дереву одну вершину и соответственно одно ребро. На рис. 3.25 изображен граф до начала работы алгоритма. Ребра остовного дерева прорисованы жирными линиями.

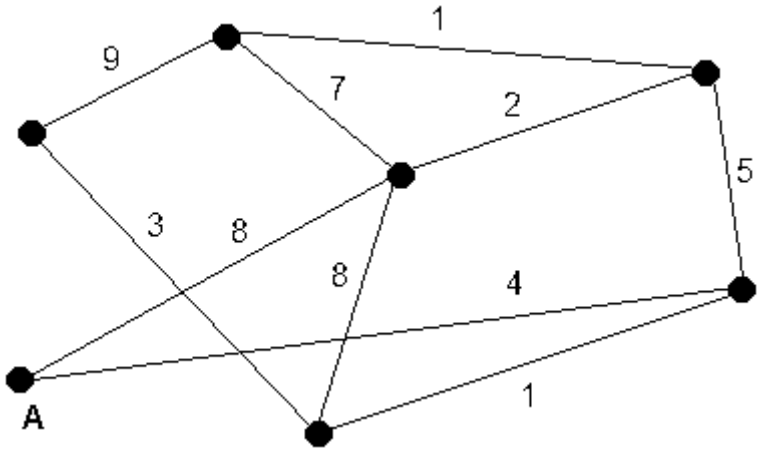

Рис. 3.25. Исходное состояние графа

Шаг 1. Из вершины A выходит два ребра которые соединяют ее с вершинами множества «оставшихся». Одно из них имеет вес = 4 и второе вес = 8. Выбираем наименьший, и отмечаем жирной линией выбранное ребро (рис. 3.26).

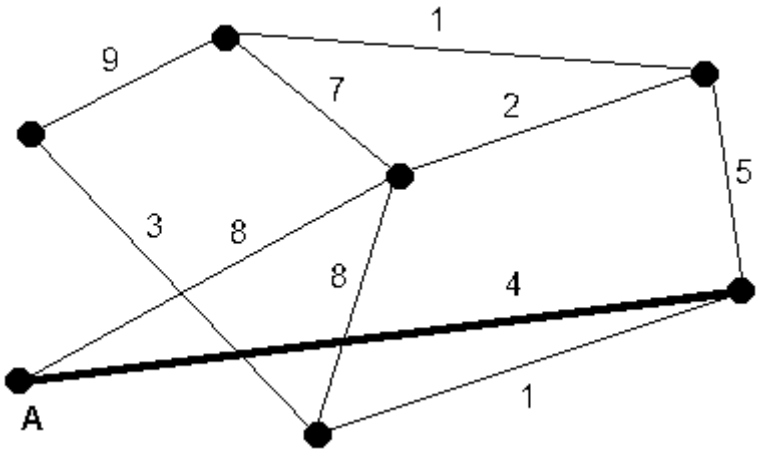

Рис. 3.26. Шаг 1

Шаг 2. Сейчас остовное дерево состоит из одного ребра и двух вершин. Рассмотрим все ребра соединяющие каждую из двух вершин остовного дерева с «оставшимися вершинами». Таких ребер три и их веса 1, 5, 8. Конечно мы выбираем для включения в строящееся дерево ребро с весом 1. и получаем следующее дерево (рис. 3.27).

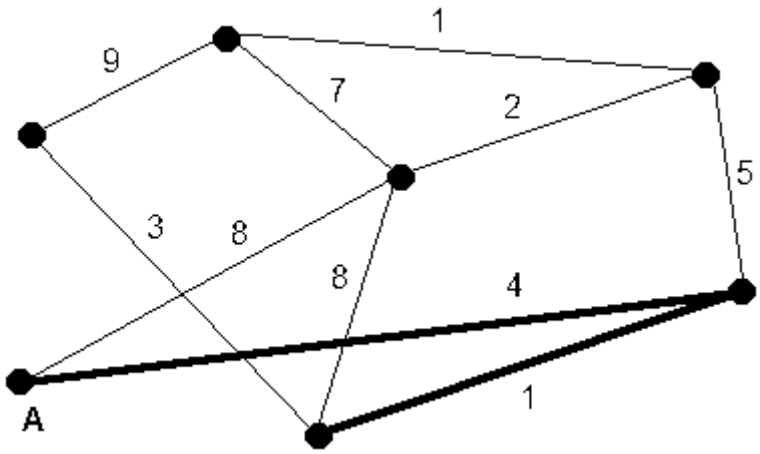

Рис. 3.27. Шаг 2

Шаг 3. Теперь остовное дерево состоит из трех вершин. Эти вершины соединены с остальным деревом тремя ребрами чей вес соответственно равен: 3, 5, 8. Следовательно следующее ребро включаемое в дерево это ребро с весом три (рис. 3.28).

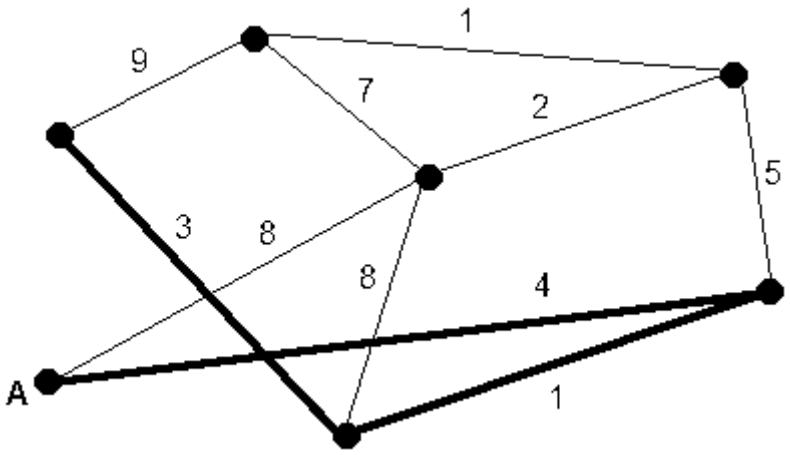

Рис. 3.28. Шаг 3

Шаг 4. В новой ситуации есть четыре ребра из которых можно выбирать минимальное и веса этих ребер: 5, 8, 8, 9. Таким образом новое ребро, это ребро с весом – 5 (рис. 3.29).

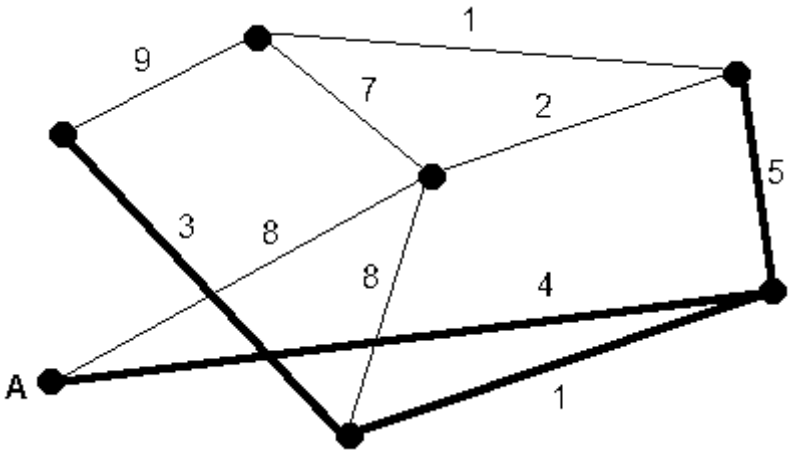

Рис. 3.29. Шаг 4

Шаг 5. Сейчас множество ребер – кандидатов на минимальные состоит из 5 ребер и их веса таковы: 1, 2, 8, 8, 9. Выбор очевиден (рис. 3.30).

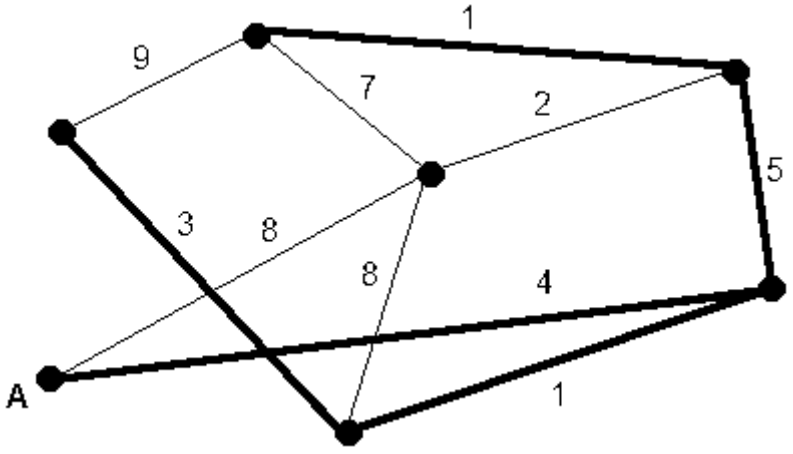

Рис. 3.30. Шаг 5

Шаг 6. И наконец последний шаг. Во множестве «оставшихся» только одна вершина. Эта вершина соединена с остовным деревом четырьмя ребрами. Их вес: 2, 7, 8, 8. Выбираем ребро с весом 2 (рис. 3.31).

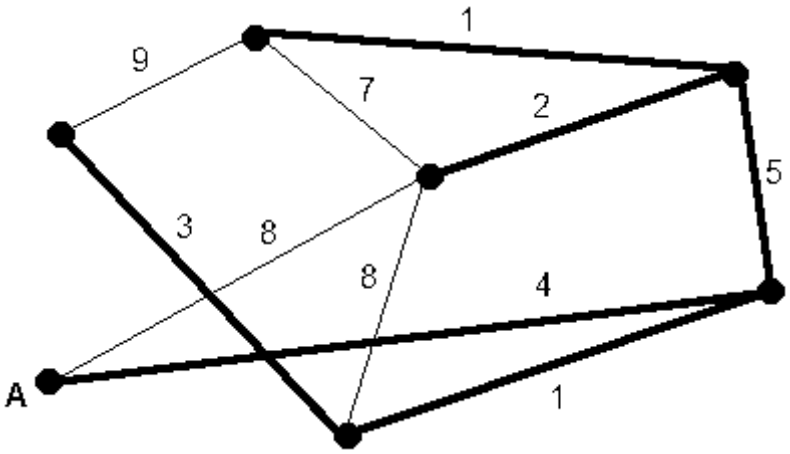

Рис. 3.31. Результат

Важное примечание:

На самом деле остовное дерево строится не для конкретной точки. В нашей версии алгоритма исходная точка существует и может показаться, что такое допущение способно исказить работу алгоритма. На самом деле существование исходной точки ровным счетом ни на что ни влияет. Это ясно из двух простых соображений:

- все вершины должны быть включены в остовное дерево;
- алгоритм строит продолжение дерева от вершины не считаясь с тем каким путем дерево было достроено до этой вершины.

Задача 48. Алгоритм Дейкстры.

Формулировка задачи:

Имеется взвешенный, неориентированный граф (веса сопоставлены ребрам). Некоторая его вершина обозначена как вершина 1. Необходимо найти минимальные пути от вершины 1 до каждой из вершин графа. Минимальным путем будем называть путь с минимальной суммой цен вдоль пути. Ценой назовем неотрицательное число являющееся весом ребра.

Идея алгоритма:

Идея основывается на следующем очевидном утверждении: Пусть построен минимальный путь из вершины A в вершину B. И пусть вершина B связана с некоторым количеством вершин i. Обозначим через C_i – цену пути из вершины B в вершину i. Выберем из всех C_i минимальную величину. Тогда минимальное продолжение пути из точки B пойдет через выбранную вершину.

Из этого утверждения вытекает очень серьезное следствие. Пусть есть множество вершин через которые уже проходят минимальные пути. Такое множество гарантированно есть, это вершина 1. Утверждение сформулированное выше дает возможность добавлять к уже существующему множеству вершин (будем далее называть их выделенными) еще одну вершину, а так как в графе количество вершин конечно, то за конечное количество шагов все вершины графа окажутся выделенными, а это и будет решением.

Сейчас перейдем немного к другой терминологии. Разобьем множество вершин на три группы:

- вершины для которых установлена окончательная цена;
- вершины без цены;
- вершины для которых установлена предварительная цена.

В начале процесса существует одна вершина, для которой известна цена пути до нее. Это первая вершина. Так как путь с нее и начинается, то ее окончательная цена равна нулю. На основании этого факта можно составить и множество вершин с известными предварительными ценами. Первая вершина соединена с некоторым количеством неоцененных вершин. Этим вершинам можно поставить в соответствие предварительные цены равные весам ребер соединяющих их с первой вершиной. Получив таким образом множество предварительных цен, далее можно получить еще одну окончательную цену, – это наименьшая из цен предварительного множества. И далее процесс идет следующим образом:

1. Строится множество вершин, не имеющих окончательной цены и соединенных с вершиной (будем называть ее ВЕРШИНА) для которой на предыдущем шаге была установлена окончательная цена.
2. Для не оцененных вершин этого множества устанавливается предварительная цена равная сумме окончательной цены ВЕРШИНЫ и цены ребра соединяющего вершину с ВЕРШИНОЙ.
3. Для вершин имеющих предварительную цену новая предварительная цена равна наименьшему значению из старой предварительной цены и посчитанной в предыдущем пункте суммы.
4. Наименьшая из предварительных цен объявляется окончательной
5. Если остались вершины без окончательных цен, то выполняется переход к первому пункту.

Обратите внимание, что алгоритм работает с двумя типами цен: ценой ребра и ценой вершины. Цены ребер являются постоянной величиной. Цены же вершин постоянно пересчитываются. Смысл этих цен различен. Цена ребра это цена перехода из вершины в вершину соединенную этим ребром. А цена вершины это цена минимального пути до этой вершины.

Прежде чем записать текст алгоритма рассмотрим пример для лучшего понимания. На рис. 3.32, построен граф и указаны веса ребер. Вершина помеченная нулем это исходная точка, относительно которой необходимо искать минимальные пути.

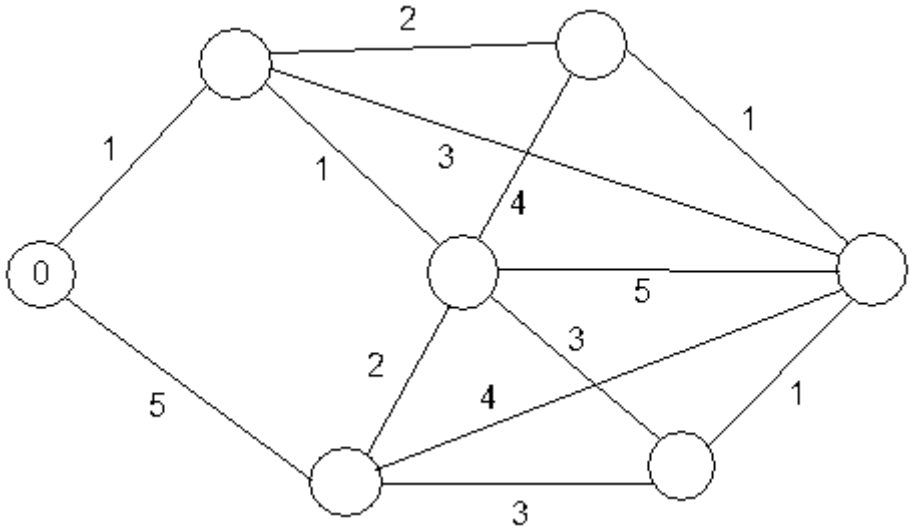

Рис. 3.32. Исходное состояние

Шаг1. Выделенные вершины будем помечать ценами внутри кружка. На первом шаге известна цена первой вершины. Эта цена равна нулю. Множество смежных вершин состоит из двух вершин их предварительная цена фактически равна цене ребер соединяющих их с исходной вершиной. Предварительную цену будем проставлять сверху от кружочка (рис. 3.33).

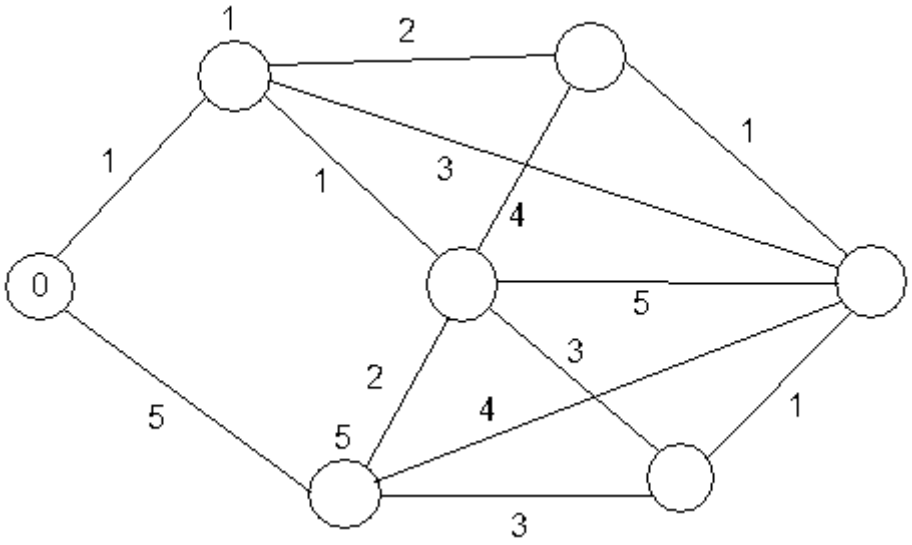

Рис. 3.33. Шаг 1

Шаг 2. На следующем шаге необходимо в множество выделенных вершин добавить одну вершину и произвести пересчет предварительных цен. Наименьшая из имеющихся цен = 1. Эту цену и объявим ценой соответствующей вершины. Новая выделенная вершина связана с тремя для которых рассчитаем предварительные цены (рис. 3.34).

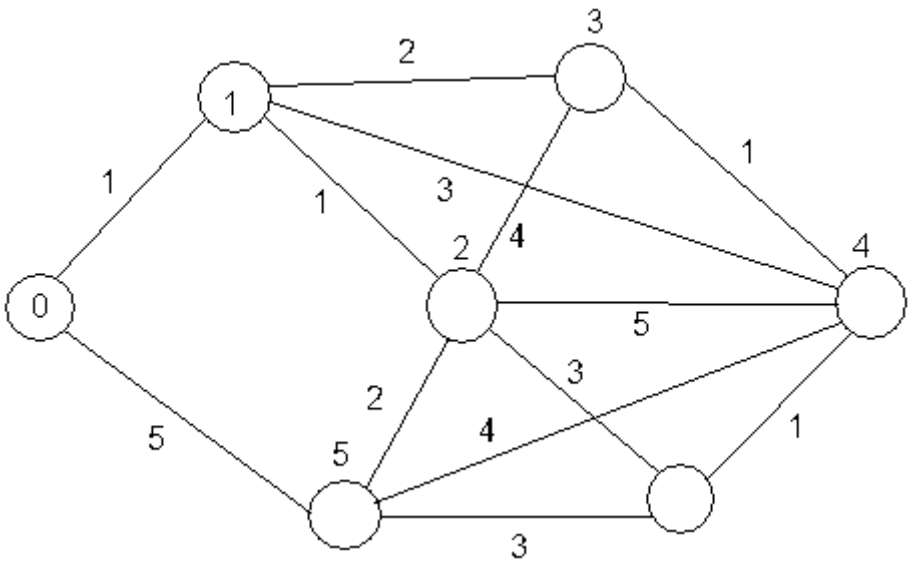

Рис. 3.34. Шаг 2

Шаг 3. На данном шаге (рис. 3.35) имеем следующие предварительные цены: 3, 2, 5, 4. Наименьшая из них – 2. Соответствующая вершина становится выделенной с ценой 2. Новая выделенная вершина связана с четырьмя невыделенными. Эта ситуация уже сложная. Пронумеруем вершины – кандидаты на пересчет начиная с верхней по часовой стрелке. Тогда:

- для первой вершины наша новая выделенная предлагает предварительную цену – 6, в то время как первая вершина уже имеет цену 3. Оставляем меньшую, то есть 3;
- для второй вершины предлагается цена 7, в то время как она уже оценена на 4. Оставляем меньшую.
- Третья вершина никак не оценена, поэтому для нее предварительная цена равна предлагаемой то есть 5;
- четвертая вершина имеет предварительную цену 5, а выделенная вершина предлагает цену 4. В этом случае предварительная цена изменяется на новую.

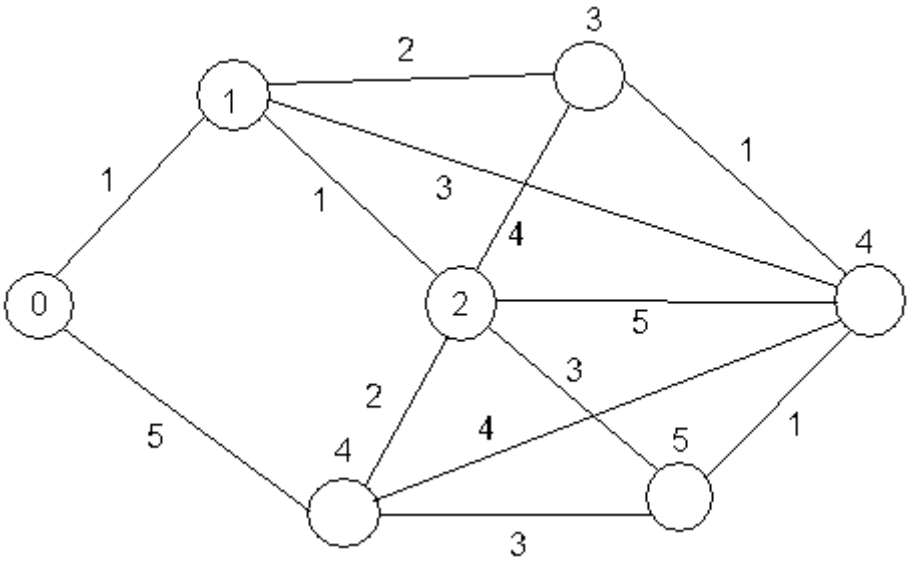

Рис. 3.35. Шаг 3

Шаг 4. На следующем шаге имеем такой набор предварительных цен: 3, 4, 5, 4. Минимальная цена равна 3. Соответствующая вершина смежна только одной невыделенной и здесь предлагаемая предварительная цена равна уже имеющейся, следовательно множество предварительных цен не меняется (рис. 3.36).

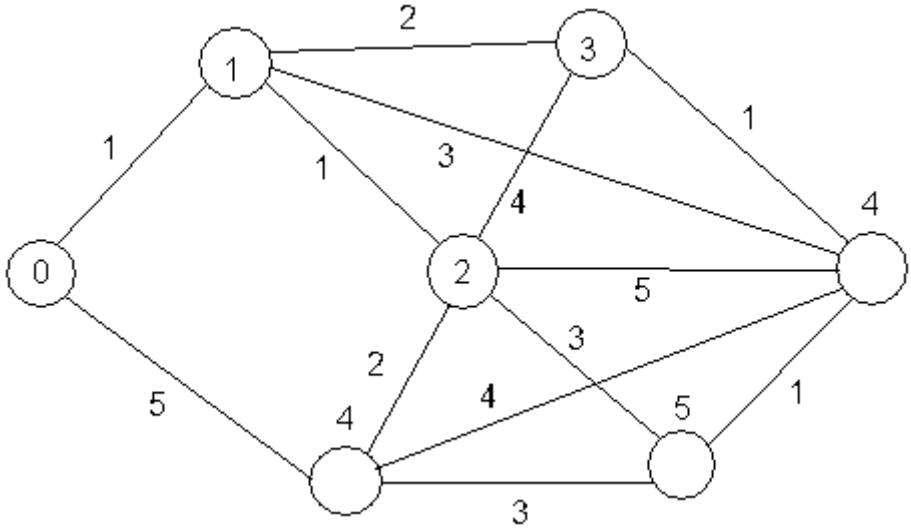

Рис. 3.36. Шаг 4

Шаг 5. Новое множество предварительных цен такое 4, 5, 4. Две вершины имеют одинаковую цену, поэтому на текущем шаге выделить можно любую из них. Мы выделим правую крайнюю, но это дело вкуса (рис. 3.37).

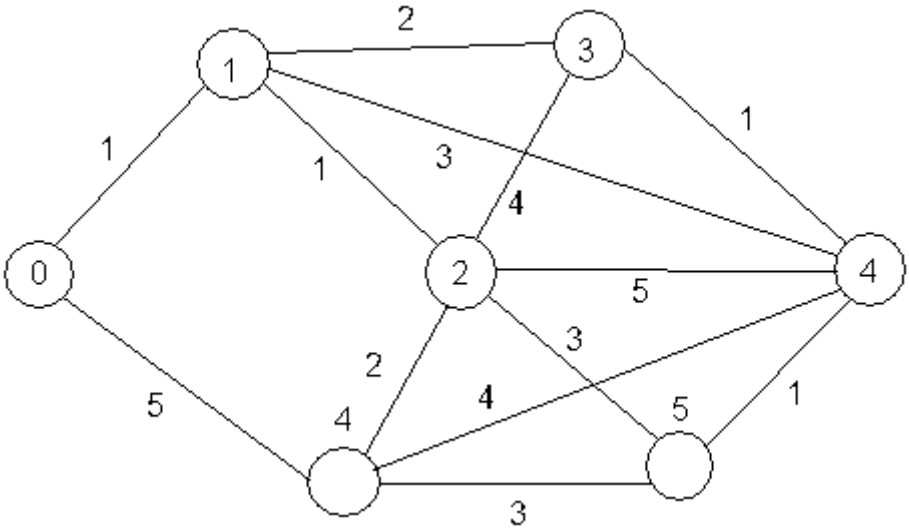

Рис 3.37. Шаг 5

Шаг 6. Осталось две вершины, обе они имеют предварительные цены. это 4 и 5. 4 минимальная. Включаем ее во множество выделенных. Эта вершина смежна оставшейся и предлагает ей предварительную цену 7. Это больше чем уже имеющаяся, поэтому цена вершины остается 5 и эта цена становится окончательной ценой последней вершины на последнем шаге (рис. 3.38).

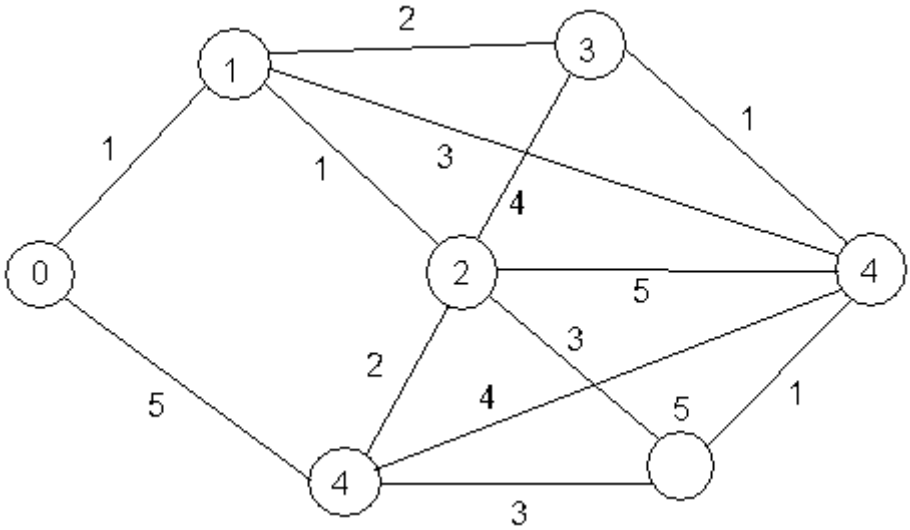

Рис. 3.38. Шаг 6

Следующий рисунок. показывает окончательную расстановку цен для минимальных путей в каждую из вершин графа. Напомним, что все минимальные пути начинаются с вершины обозначенной нулем (рис. 3.39).

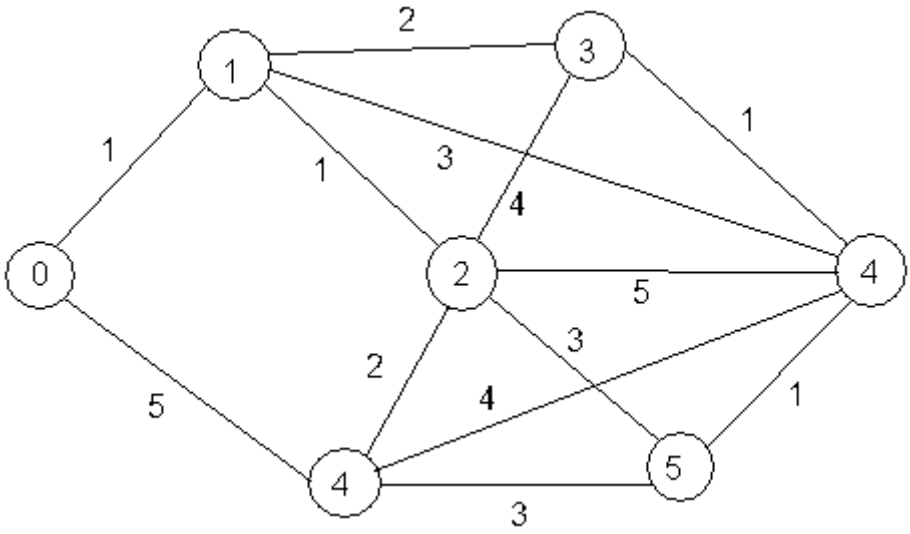

Рис. 3.39. Результат

Алгоритм:

Множество выделенных вершин = исходная вершина
Пока есть невыделенные вершины делать
    Для каждой вершины смежной ПОСЛЕДНЕЙ ВЫДЕЛЕННОЙ рассчитать предварительную цену как минимальную из уже имеющейся и цены полученной с учетом пути от ПОСЛЕДНЕЙ ВЫДЕЛЕННОЙ до данной.
    Среди множества вершин для которых определена предварительная цена найти вершину с минимальным значением предварительной цены.
    Найденную вершину занести во множество выделенных вершин.

Обоснование алгоритма:

Предположим до некоторой вершины существует более короткий (дешевый) путь нежели найденный алгоритмом. Это означает, что на некотором этапе работы мы можем присоединить к множеству выделенных вершин не минимальную, а какую-то другую вершину. Но любая другая вершина имеет цену большую нежели минимальная, а это и означает, что путь проходящий через ней минимальным не будет.

Задача 49. Алгоритм Флойда.

Дан непустой взвешенный граф с произвольными весами ребер. Требуется найти кратчайшие длины путей между всеми парами вершин графа, если в графе нет циклов отрицательной длины или обнаружить наличие таких циклов.

Пояснения. Циклы отрицательной длины могут появится при наличии отрицательных весов. В этом случае сумма весов вдоль цикла действительно может оказаться отрицательной. А под кратчайшей длиной пути естественно понимается путь сумма весов вдоль которого минимальна.

Описание алгоритма:

Инициализация структур данных

Построим матрицу D^0 размерности $|V| \times |V|$, элементы которой (обозначим из v) определяются по правилу:

1. $d_{ii}^0 = 0$;
2. $d_{ij}^0 =$ Вес(v_i, v_j), где $i<>j$, если в графе существует ребро (дуга) (v_i, v_j);
3. $d_{ij}^0 =$ бесконечность , где $i<>j$, если нет ребра (дуги) (v_i, v_j).

Основная часть алгоритма:

Выполнять цикл, завершение которого наступает по выполнению одного из двух условий: либо количество шагов цикла равно V, либо был обнаружен цикл отрицательной длины. Шаги цикла нумеруются с нуля. Шаг цикла будем обозначать переменной m. Ниже действия цикла:

Строится матрица с индексом равным номеру шага, обозначим его через m, в которой элементы определяются через элементы матрицы предыдущего шага по следующим формулам:
$d_{ij}^{m+1} = \min\{d_{ij}^m, d_{i(m+1)}^m + d_{(m+1)j}^m\}$, где $i<>j$; $d_{ii}^{m+1} = 0$.
Если $d_{im}^m + d_{mi}^m < 0$ для какого-то i, то в графе существует цикл (контур) отрицательной длины, проходящий через вершину v_i;

По завершению работы данного алгоритма, элементы матрицы равны длинам кратчайших путей между соответствующими вершинами.

Поиск путей

Если требует найти сами пути, то перед началом работы алгоритма построим матрицу P с начальными значениями элементов $p_{ij}=i$. Каждый раз, когда на шаге (1) значение d_{ij}^{m+1} будет уменьшаться (т.е. когда $d_{i(m+1)}^m + d_{(m+1)j}^m < d_{ij}^m$), выполним присваивание $p_{ij} := p_{(m+1)j}$. В конце работы алгоритма матрица P будет определять кратчайшие пути между всеми парами вершин: значение p_{ij} будет равно номеру предпоследней вершины в пути между i и j (либо $p_{ij}=i$, если путь не существует).

Примечание: если граф – неориентированный, то все матрицы D^m являются симметричными, поэтому достаточно вычислять элементы, находящиеся только выше (либо только ниже) главной диагонали.

Пример:

Граф построенный на рисунке послужит примером применения рассмотренного алгоритма. Числа проставленные в кружках это номера вершин (рис. 3.40).

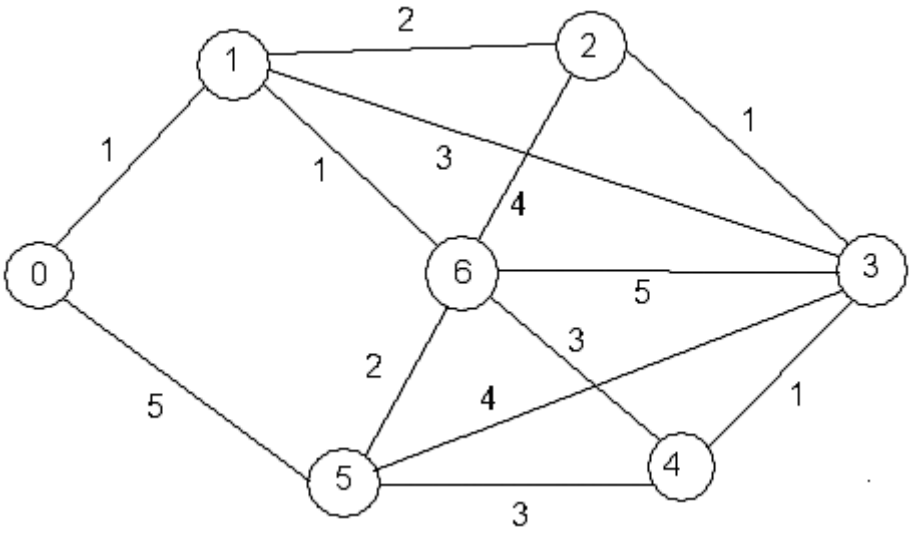

Рис. 3.40. Исходное состояние

Первым шагом алгоритма строится матрица весов.

Таблица 3.4. Матрица исходного состояния

	0	1	2	3	4	5	6
0	0	1	∞	∞	∞	5	∞
1	1	0	2	3	∞	∞	1
2	∞	2	0	1	∞	∞	4
3	∞	3	1	0	1	4	5
4	∞	∞	∞	1	0	3	3
5	5	∞	∞	4	3	0	2
6	∞	1	4	5	3	2	0

Шаг первый: m=0

Таблица 3.5. Матрица первого шага

	0	1	2	3	4	5	6
0	0	1	3	4	∞	5	2
1	1	0	2	3	∞	∞	1
2	3	2	0	1	∞	∞	3
3	4	3	1	0	1	4	4
4	∞	∞	∞	1	0	3	3
5	5	∞	∞	4	3	0	2
6	2	1	3	4	3	2	0

Шаг второй: m=1

Таблица 3.6. Матрица второго шага

	0	1	2	3	4	5	6
0	0	1	3	4	∞	5	2
1	1	0	2	3	∞	∞	1
2	3	2	0	1	∞	∞	3
3	4	3	1	0	1	4	4
4	∞	∞	∞	1	0	3	3
5	5	∞	∞	4	3	0	2
6	2	1	3	4	3	2	0

Шаг третий: m=2

Таблица 3.7. Матрица третьего шага

	0	1	2	3	4	5	6
0	0	1	3	4	5	5	2
1	1	0	2	3	4	7	1
2	3	2	0	1	2	5	3
3	4	3	1	0	1	4	4
4	5	4	2	1	0	3	3
5	5	7	5	4	3	0	2
6	2	1	3	4	3	2	0

Шаг четвертый: m=3

Таблица 3.8. Матрица четвертого шага

	0	1	2	3	4	5	6
0	0	1	3	4	5	5	2
1	1	0	2	3	4	7	1
2	3	2	0	1	2	5	3
3	4	3	1	0	1	4	4
4	5	4	2	1	0	3	3
5	5	7	5	4	3	0	2
6	2	1	3	4	3	2	0

Шаг пятый: m=4

Таблица 3.9. Матрица пятого шага

	0	1	2	3	4	5	6
0	0	1	3	4	5	5	2
1	1	0	2	3	4	7	1
2	3	2	0	1	2	5	3
3	4	3	1	0	1	4	4
4	5	2	2	1	0	3	3
5	5	7	5	4	3	0	2
6	2	1	3	4	3	2	0

Шаг шестой: m=5

Таблица 3.10. Матрица результат

	0	1	2	3	4	5	6
0	0	1	3	4	5	4	2
1	1	0	2	3	4	3	1
2	3	2	0	1	2	5	3
3	4	3	1	0	1	4	4
4	5	2	2	1	0	3	3
5	4	3	5	4	3	0	2
6	2	1	3	4	3	2	0

Полученная на последнем шаге таблица содержит минимальные стоимости за которые можно добраться от одной вершины до любой другой. Пользоваться этой таблицей можно так. Предположим, нас интересует стоимость пути от второй вершины до пятой. На пересечении второй строки и пятого столбца стоит число 5.

Это означает, что существует путь стоимостью в пять единиц и более дешевого нет. Действительно видно, что такой путь есть. Это путь через вершины: 2 – 3 – 4 – 5. Так как наш граф достаточно прост, то легко видеть, что более короткого пути действительно нет.

Обоснование алгоритма:

Исходная структура содержит цены (веса) для ребер непосредственно соединяющих две вершины. Иначе можно сказать, что исходная матрица содержит цены уже построенных путей состоящих из одного ребра. Бесконечные значения появляются в силу того, что есть пары вершин между которыми нет пути длиной в одно ребро.

Следующая итерация строит пути длиной в два ребра, следующая длиной в три ребра и т.д. Выбор минимального значения из двух необходим для того, чтобы определить, какой путь имеет более низкую стоимость, тот который уже построен или новый, более длинный (на одно ребро).

Количество итераций равно количеству вершин – 1. Это следует из очевидного факта, что максимально длинный путь без циклов (а только такой имеет смысл) максимально может содержать каждую вершину только один раз, а количество ребер составляющих такой максимальный путь равно количеству вершин – 1.

Если все цены неотрицательны, то алгоритм Флойда можно рассматривать как более сильный вариант алгоритма Дейкстры. Однако если появляются отрицательные величины, то в случае появления цикла с отрицательной стоимостью, алгоритм Флойда уже не справится со своей задачей, но в качестве компенсации он позволяет обнаружить такой плохой цикл.

Задача 50. Поиск максимального потока.

Дан связный ориентированный, взвешенный граф. Веса сопоставлены дугам. Одна из вершин графа является источником потока и одна стоком для потока. Определить максимальный поток, который можно пропустить по графу от истока до стока.

Идея решения:

Прежде чем описывать метод решения задачи, необходимо видимо пояснить ее сущность. Описанный в условии граф это модель транспортных систем передающих от некоторого пункта к некоторому пункту что-либо. В качестве примера такой системы можно привести систему труб, по которым от источника (производителя, со склада и т.д.) передается например вода, или газ. В этом случае вершины графа это узлы соединения труб, дуги – это трубы, а их пропускная способность, – это площадь сечения труб.

Еще одним примером может быть электрическая сеть источником которой будет какой-либо источник тока, дуги графа – это система проводников, вершины графа – проводники тока, вершины – точки соединения, а веса ребер – электропропускная способность проводников.

В общем случае, источник не обязательно один. Стоков также может быть несколько. Кроме того, дуги пропускающие поток только в одном направлении это также некоторое упрощение.

Но мы рассмотрим все же именно такую упрощенную задачу, с одним истоком, одним стоком и дугами поток по которым может идти только в одну сторону. Именно поэтому в исходной формулировке говорится о ориентированном графе.

Еще два заметных упрощения: во-первых, пусть исходный граф не имеет циклов и во-вторых граф таков, что из поток из каждой вершины графа может дойти до стока (граф без тупиков).

Мы опишем метод решения поставленной упрощенной задачи, вы при желании можете рассмотреть ситуацию без заданных упрощений.

Метод Форда – Фалкерсона

Метод основан на идее дополнения уже существующего потока. Предположим, что некоторый поток по сети (графу) уже идет. Тогда необходимо найти путь, по которому поток можно еще немного увеличить. Такова главная идея. Рассмотрим технику построения дополнительного пути.

Сопоставим каждой дуге графа еще одно число – остаточную пропускную способность. Это величина равна разности собственной пропускной способности и величины потока уже идущего по дуге. Путь строится так:

- найдем дугу с наименьшей остаточной пропускной способностью;
- пустим по найденной дуге дополнительный поток равный остаточной пропускной способности и выясним, можно ли такой поток подвести к дуге и можно ли такой поток пропустить от данной дуги к стоку.

Две задачи: получения дополнительного потока от источника и отвода дополнительного потока до стока аналогичны. Поэтому рассмотрим только одну из них.

Как отвести дополнительный поток от выбранной дуги до стока

Введем понятие «остаточной пропускной способности вершины». Эта величина равна сумме остаточных пропускных способностей дуг исходящих из вершины. Поток от исходной дуги распространяется по сети и при этом, для каждого ребра возможны две ситуации:

- дополнительный поток по дуге меньше либо равен остаточной пропускной способности вершины. В этом случае поток от дуги входит в вершину и перераспределяется между исходящими дугами.
- дополнительный поток больше остаточной пропускной способности вершины. В этом случае избыток потока «возвращается» в вершину источник данной дуги и в ней выполняется перераспределение. Если перераспределение в источнике дуги не решило проблему (остался еще избыточный поток), то остаток возвращается дальше. Если некоторый остаток вернулся к исходной дуге, то дополнительный поток по ней уменьшается.

Примечание. Последняя задача самая сложная в разделе и быть может самая сложная во всем практикуме. Но тем не менее, вам предоставляется не только разработка программы, но и существенная доработка алгоритма. То что сказано выше о методе можно считать только первым приближением. Но если вы дошли до этой задачи, то вы должны справиться со всеми проблемами.

Приложение. Кратко о теории графов

Введение

Любой раздел математики занимается обработкой данных. Данные эти могут быть самыми разнообразными. Чаще всего это конечно числа. Но не всегда. Иногда объектом изучения становятся сложные объекты построенные из чисел или даже других сложных объектов. Например, планиметрия изучает плоские геометрические фигуры, описываемые числами, но числами не являющимися.

Но независимо от того, какое множество объектов изучает раздел математики этих объектов много, или как было только что сказано, математика изучает множество объектов.

Немного уточним. Вообще то не любое множество. Множества исследуемые математикой состоят из похожих элементов. Та же планиметрия не изучает системы уравнений (но может ими пользоваться), а алгебра не изучает треугольники (но может использоваться для их изучения). Для того, чтобы объекты объединить в одно множество они должны обладать чем-то общим, иначе говоря все элементы множества можно назвать одним именем.

Примеры: множество линейных уравнений, множество натуральных чисел, множество геометрических фигур и т.д. дело же математики исследовать свойства элементов множества и отношения (связи) между ними.

Что такое отношения. Вспомним теорему Пифагора: *Квадрат гипотенузы равен сумме квадратов катетов.* Эта теорема описывает отношение между катетами прямоугольного треугольника и гипотенузой. Еще один пример: *Если дискриминант квадратного уравнения равен нулю, то уравнение имеет один действительный корень.* Это утверждение описывает отношение между корнями уравнения и значением дискриминанта.

Достаточно часто отношения между элементами множества можно представить в виде рисунка. Рассмотрим несколько примеров.

Очень простой пример. На плоской карте расположено несколько городов. Некоторые из них соединены дорогами, а некоторые нет. Длина каждой дороги известна. Некий путешественник начиная путь из города А стремится попасть в город В. Необходимо найти кратчайший путь, если известно, что путешествовать он будет только по дорогам.

Эта задача проиллюстрирована на рис. 4.1.

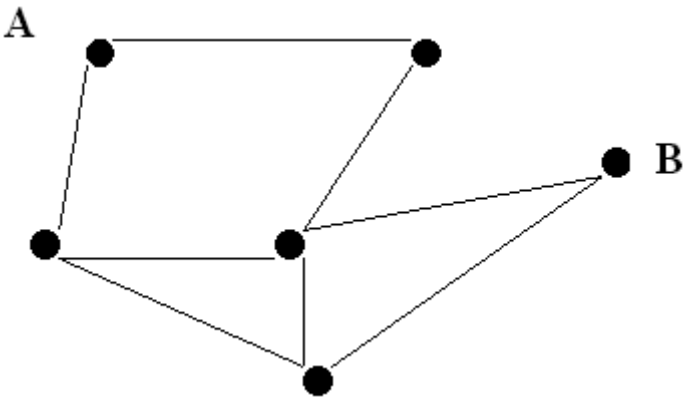

Рис. 4.1. Простая карта

Здесь дороги – это отношения между элементами множества населенных пунктов, а кружочки изображают сами населенные пункты.

Еще один пример. Дана куча камней разного веса. Требуется разбросать эту кучу на две, так чтобы их вес (этих двух куч) отличался как можно меньше. Ясно, что для решения этой задачи необходимо перебрать все возможные варианты распределения камней по двум кучам, для каждого варианта вычислить разность весов и в процессе работы определить вариант с наименьшей разностью. Здесь также можно построить графическое представление.

Заметим, что из каждого варианта распределения можно получить еще несколько очень простым действием. А именно взяв камень из одной кучи и положив в другую. Будем говорить, что два распределения находятся в отношении друг с другом, если одно из них можно получить из другого перекладыванием одного камня из одной кучи в другую. Тогда все множество распределений представимо похожей картинкой, рисовать ее еще раз мы уже не будем, заметим только, что черные кружки на этот раз будут изображать собой распределения, а отрезки между ними будут показывать какие из них находятся в описанном выше отношении.

Еще один пример. А сейчас представим в точно таком же графическом виде арифметический пример. Возьмем следующее выражение: 7*(6-2)+5/(9-3) (рис. 4.2).

Здесь стрелками показано перемещение аргументов между операциями и соответственно порядок выполнения операций. Есть два небольших отличия от картинок нарисованных раньше. Сейчас вместо кружков мы используем квадратики и вместо простых отрезков стрелки. Второе отличие существенно, но о нем разговор будет позже.

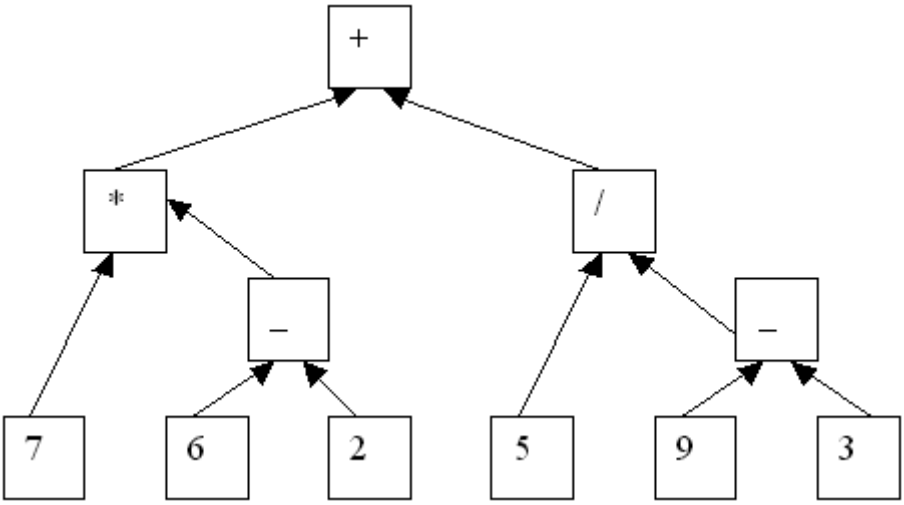

Рис. 4.2. Дерево операций

Основные определения

Картинки, которые мы рисовали выше – называются графами, с их помощью можно не только более понятно представлять условия задач, но и разрабатывать методы решения. Но прежде чем заняться методами решения необходимо дать точные определения. В примерах приведенных выше нам встречались три важных особенности. На рисунке 4.1 кружки (далее будем называть их вершинами) соединены простыми отрезками, а на рисунке 4.2 вершины соединены стрелками.

Первый случай графа называется неориентированным, а во втором случае граф называется ориентированным. Кроме того, обратите внимание, что на рисунке 4.2 нет вершин, для которых возможен путь, начинающийся в этой вершине и в ней же заканчивающийся. Эта особенность также порождает особый вид графа. Вот эти виды графов мы и определим

Неориентированным графом или просто графом G = (X, U) называется упорядоченная пара (X, U), где X есть непустое множество вершин графа, а U есть множество неупорядоченных пар элементов из X, называемых ребрами графа.

Примечание. Какой смысл имеют вершины и ребра графа, теория графов не изучает, поэтому смысл эти понятия могут иметь любой, теорию же интересует только структура графа. В наших примерах вершинами являются кружки и квадраты, а ребрами – отрезки и стрелки их соединяющие.

Очень важно также запомнить, что граф по определению это множество, то есть набор объектов такой структуры какая описана в определении, а наши примеры с кружочками и отрезками это наглядная графическая иллюстрация, Хотя, обычно изображая картинку о ней говорят, как о графе.

Ориентированным графом (Орграфом) G=(X, Y) называется упорядоченная пара (X, Y), где X есть непустое множество вершин орграфа(ориентированного графа), а Y есть множество упорядоченных пар элементов из X называемых дугами орграфа. Рисунок 4.2 как раз пример ориентированного графа.

Примечание. Ориентированный граф от неориентированного отличается тем, что его ребро имеет жестко заданное направление. На рисунке это изображается стрелкой. В ориентированном графе между двумя соседями соединенными дугой перемещение возможно только в одном направлении. В неориентированном графе если вершина A соединена с вершиной B ребрами, то из A можно дойти в B. В ориентированном это может оказаться не так.

Очень много задач теории графов исследуют именно возможность построения пути из одного пункта в другой, поэтому дадим еще несколько важных определений, что такое путь на графе, какие виды путей существуют.

Пусть u – дуга орграфа вида (x, y), u=(x, y), Вершина x называется началом дуги u, а вершина y – концом дуги u. При этом говорят, что дуга выходит из x и входит в y. Если дуга или входит в вершину или выходит из нее говорят, что она инцидентна вершине. Дуга, у которой начало и конец находятся в одной и той же вершине называется петлей. Вершины, соединенные дугой называются смежными.

Путем из вершины a в вершину b называется последовательность вершин и дуг вида $a(a, x_1), x_1(x_1, x_2)....x_{n-1}(x_{n-1}, b)b$. Путь называется простым, если ни одна вершина в нем не встречается дважды.

Если в орграфе две вершины (a b) связаны путем u(a, b), то говорят, что b достижима из вершины a. Орграф называется ***односторонне связанным*** если одна вершина достижима из другой. ***Орграф называется сильно связанным*** если для любой пары вершин каждая из них достижима из другой.

Путь называется ***эйлеровым*** если он содержит все дуги графа ровно по одному разу, и он называется ***гамильтоновым***, если он содержит все вершины ровно по одному разу.

Примечание. Обратите внимание на понятие связности. Это одно из важнейших понятий. Оно говорит о том, насколько граф представляет собой единое целое. Это понятие не характеризует количество ребер через которые можно дойти до вершины, только сам факт, можно дойти или нельзя. Ниже примеры связанных и несвязанных графов. Первые два примера это неориентированный связанный и несвязный граф (рис 4.3).

А теперь примеры на связность ориентированного графа (рис 4.4).

Выше уже было сказано, что многие задачи теории связаны с возможностью обхода графа или поиском пути из одной вершины в другую. Поэтому дадим еще несколько определений описывающих особые вершины графа или их особые свойства.

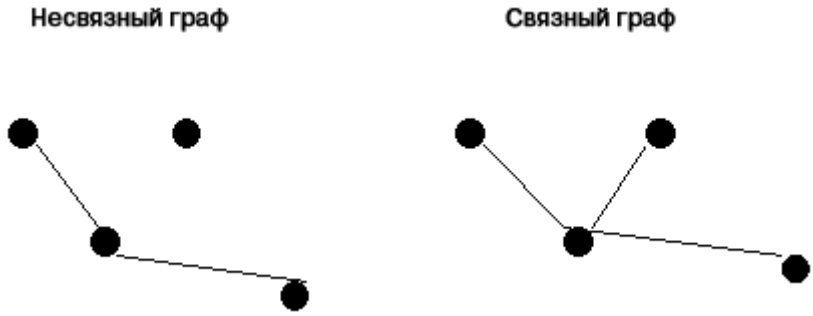

Рис 4.3. Примеры графов

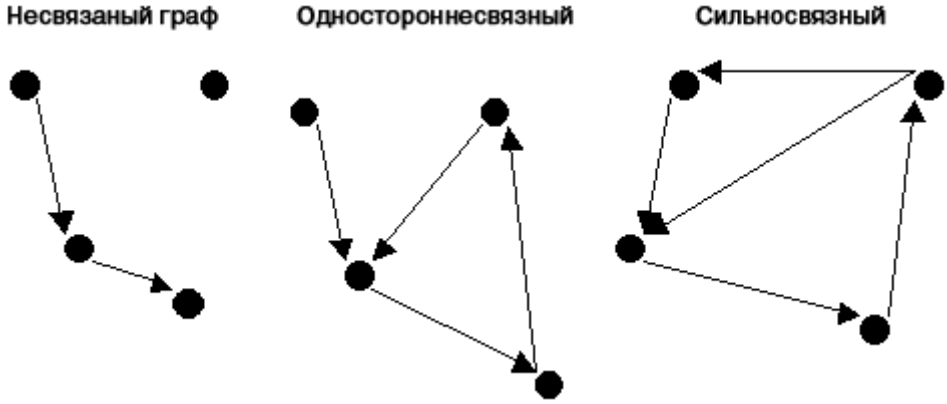

Рис 4.4. Примеры графов

Степенью вершины называется общее число дуг инцидентных вершине. Число дуг входящих в вершину называется полустепенью входа и число дуг исходящих из вершины называется полустепенью исхода. Заметим, что для неориентированных графов эти две полустепени равны, для ориентированных равенство не обязательно.

Предшественником вершины называется вершина, соединенная с данной дугой, по которой из предшественника можно пройти до данной. Сама данная вершина по отношению к предшественнику называется потомком.

Входом или начальной вершиной графа называется вершина s у которой полустепень входа равна нулю. Входов может быть несколько.

Выходом или конечной вершиной графа называется вершина t у которой полустепень выхода равна нулю. Выходов, как и входов может быть несколько.

Примечание. Понятно, что три последних определения действуют только для ориентированных графов.

Контур. Во многих графах, как например, на рис 4. можно построить путь который будет заканчиваться в той же вершине в какой он и начинается. Такой путь называется контуром.

Для ориентированного графа может так получиться, что две вершины соединены последовательностью ребер и все же одна из них не достижима из другой. Такие вершины есть на одностороннесвязном графе рис. 4.4. Но тем не менее в этом примере все вершины соединены между собой, ни одну из них нельзя убрать из графа не разрывая ребер. Для анализа таких ситуаций вводится еще два понятия:

Цепь. Цепью называется последовательность ребер ($p_1,p_2...p_n$) неориентированного графа следующего вида $p_i = (v_i, v_{i+1})$, i = 1,2, ...n. Вершины цепи могут иметь степень, равную 1. Вершина со степенью 1 называется концевой. Цепь называется составной если в ней повторяется хотя бы одно ребро, сложной если повторяется хотя бы одна вершина, и простой в противном случае.

Циклом называется цепь у которой начальная и конечная вершины совпадают.

Примечание. Цикл и цепь, контур и путь используются для определения понятия связности, но первые два используются для неориентированных графов, а вторые два понятия для ориентированных.

Контур простой и эйлеровый. Путь, начало и конец которого совпадают называется контуром. Контур называется простым, если ни одна вершина в нем не повторяется дважды, эйлеровым, если он содержит все дуги графа в точности по одному разу, гамильтоновым если он содержит все вершины графа в точности по одному разу. Длиной пути или контура называется число дуг входящих в него.

Дерево. Существует еще один особый вид графов – это деревья. Деревом называется связный граф с одним входом в котором нет ни одного контура. На рис. 4.5 приведен пример двоичного дерева.

Конечно, дерево, как и любой другой граф может быть ориентированным и неориентированным. Нетрудно заметить, что ориентированное дерево не может обладать сильной связностью.

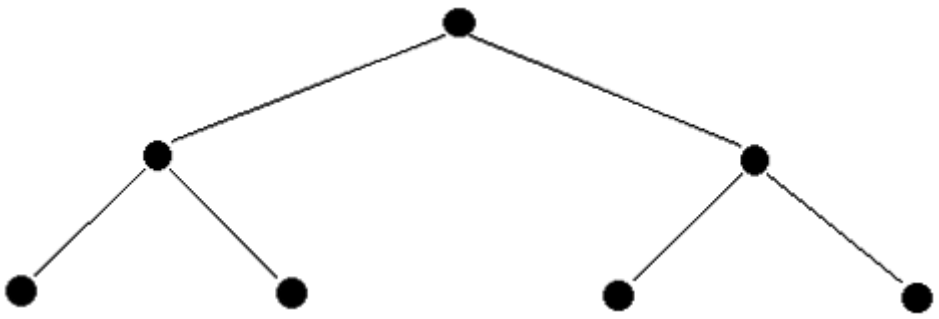

Рис. 4.5. Дерево

Взвешенный граф. Вернемся ненадолго к первому примеру задачи использующей понятие графа. В ней говорилось о карте на которой путешественнику необходимо просто найти путь из города A в город B. Более реальна задача в которой надо найти не просто путь, а например кратчайший или наиболее дешевый. Слова кратчайший, более дешевый говорят о том, что для решения задачи нам необходимо уметь сравнивать, а сравнивать на больше, меньше мы можем только числа, поэтому обычный граф теперь нам уже не подходит.

Взвешенный граф – это граф в котором каждой вершине (или ребру/дуге) сопоставлено некоторое число называемое весом.

Подграфы. Граф может оказаться очень большим, содержащим огромное количество вершин и ребер составляющих очень сложную структуру. В такой ситуации часто бывает полезно выделить из большого графа меньший, с какими-то определенными свойствами. Конечно, подграфы могут то же быть сколь угодно сложными, но существует три стандартных типа подграфов. Дадим им определения.

Частичным графом называется граф состоящий из некоторого подмножества дуг исходного орграфа вместе с их концами.

Подграфом называется граф состоящий из некоторого подмножества вершин исходного орграфа и тех дуг оба конца которых принадлежат данному подмножеству.

Суграфом называется граф содержащий все вершины исходного орграфа и некоторое подмножество дуг исходного орграфа.

Интересные факты и теоремы теории графов

Задача 1. Сколько существует деревьев с N вершинами.

Прежде чем разбираться в этом достаточно сложном вопросе, необходимо выяснить, какие два графа можно считать различными, а какие два являются одинаковыми. Вспомним, что длина ребер и их ориентация не являются характеристиками графа. Например, два графа изображенные на рис. 4.6 являются одинаковыми.

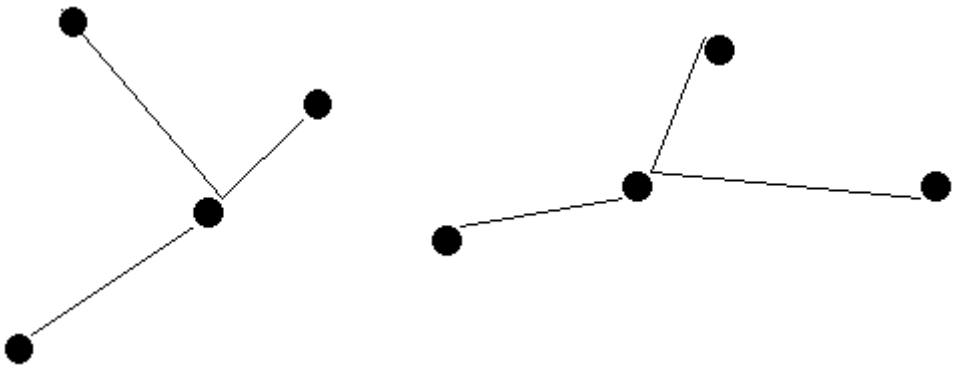

Рис. 4.6 Изоморфные графы

Иначе их можно назвать изоморфными. Изоморфными можно назвать два такие графа которые отличаются друг от друга только длиной ребер и расположением своих вершин. А как определить изоморфны два графа или нет мы разберем немного ниже.

Нетрудно представить себе, что для двух сложных графов с большим количеством вершин и запутанной системой ребер очень трудно будет установить изоморфны они или нет. Поэтому необходим критерий позволяющий сделать нужный вывод простым подсчетом. Такой критерий есть и вот как он звучит:

Два графа изоморфны если их вершины можно пронумеровать целыми числами так, что для любых двух номеров их носители либо соединены ребром на обоих графах либо нет.

Разобравшись с вопросом изоморфизма вернемся к задаче. Заметим для начала, что задача пересчета любых деревьев слишком сложна для нашего небольшого приложения и мы ее упростим. Займемся пересчетом двоичных деревьев. Пример такого дерева показан на рисунке 4.5. Видно, что двоичное дерево это такое дерево в котором из любого узла выходит ровно две ветви.

Итак вопрос – сколько существует двоичных деревьев с N вершинами.

Заметим, что отрубив корень двоичного дерева мы получим два двоичных дерева меньшего размера. Это означает, что любое двоичное дерево можно собрать из меньших. Отсюда возникает идея решения. *Можно подсчитать количество деревьев, если известно количество меньших деревьев.* Запишем это более строго.

Обозначим через B_N количество деревьев с N вершинами. Предположим, что все числа B_1, B_2,.....B_N известны. Необходимо найти величину B_{N+1}.

Так как новое дерево мы получаем склеиванием двух меньших, то можно записать равенство $N+1 = i + k$ где

$i = 1,2,.....N$
$k = N, N-1,.....1$

Деревьев с i вершинами B_i штук, а деревьев с k вершинами B_k штук. Эти деревья можно попарно склеить $B_i B_k$ способами. Если же учесть интервал изменения i и k то общее количество возможных комбинаций деревьев будет выражаться следующей суммой:

$$B_{N+1} = B_1B_N + B_2B_{N-1} +.....+B_NB_1$$

Эта формула предполагает, что из любой вершины всегда выходят две ветви и следовательно любое дерево содержит два поддерева. Однако это не обязательно так. Вполне возможна ситуация когда из вершины выходит только одна ветвь. Чтобы эта формула продолжала работать и в такой ситуации будем считать, что из каждого узла все же выходят два дерева, только возможно, одно из них не совсем настоящее без вершин. Для такого дерева положим $B_0 = 1$. Тогда наша формула перепишется так:

$$B_{N+1} = B_0B_N + B_2B_{N-1} +.....+B_NB_0$$

Вычислим несколько таких последовательных чисел:

$B_0 = 1$
$B_1 = 1$
$B_2 = 1*1 + 1*1 = 2$
$B_3 = 1*2 + 1*1 + 2*1 = 5$
$B_4 = 1*5 + 1*2 + 2*1 + 5*1 = 14$

Эти числа называются числами Каталана и их можно вычислять не только, как последовательный ряд, существует и явная формула для расчета:

$B_N = (2N!)/((N+1)!N!)$

Примечание. Значение чисел Каталана выходит за рамки проблемы перечисления двоичных деревьев. Числа Каталана равны например

- количеству способов разбиения многоугольника на треугольники;
- количеству способов попарного соединения 2n точек окружности непересекающимися хордами;
- количеству способов правильной расстановки n пар скобок.

Задача 2. Какие графы можно расположить на плоскости без самопересечений

Для начала рассмотрим пример. В плоской местности расположено три дома и три колодца (рис 4.7). Можно ли проложить дороги так, чтобы выполнялись следующие условия:

- каждый дом соединен с каждым колодцем;
- никакая пара дорог не пересекается.

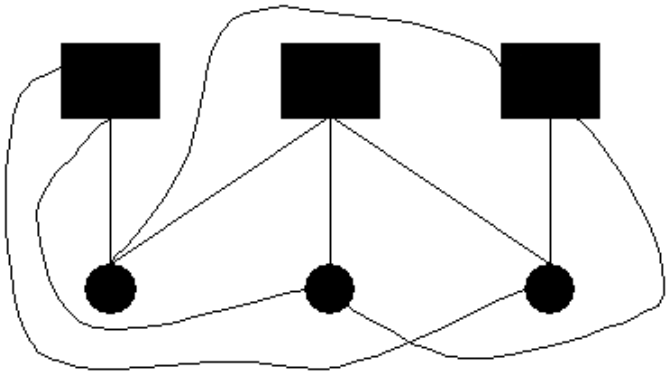

Рис. 4.7. Три дома и три колодца

На рисунке не удалось соединить дома и колодцы, более того, можно доказать, что эта задача не имеет решения при любом расположении домов и колодцев и при любом способе их соединения. Это означает, что не любой граф можно расположить на плоскости без пересечений.

А между прочим умение строить именно такие графы имеет значительное практическое применение. Например, в электронике требуется разрабатывать такие печатные платы, в которых проводящие дорожки не пересекались бы, в противном случае плата не будет работоспособной.

Графы которые можно расположить на плоскости без пересечений называются планарными и сейчас мы рассмотрим критерий позволяющий в отношении любого графа определить планарный он или нет.

Введем важное вспомогательное понятие – ГОМЕОМОРФИЗМ.

Допустим, что мы имеем два графа и нам разрешено делать с ними следующие операции:

- можно растягивать или наоборот сжимать ребра;
- можно искривлять ребра;
- можно изменять положение вершин.

Нельзя только две вещи, нельзя граф резать и склеивать. Если при выполнении этих условий окажется, что один из данных графов можно преобразовать в другой, то можно сказать, то графы гомеоморфны. Например изоморфные графы с рис. 4.6 являются также и гомеоморфными. Еще один несложный пример гомеоморфных графов (рис 4.8):

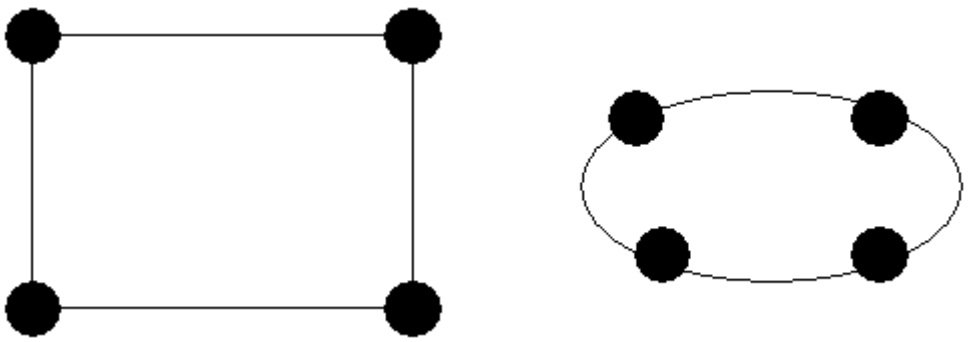

Рис. 4.8. Гомеоморфные графы

А теперь более сложный пример (рис. 4.9).

Эти два графа также гомеоморфны. Граф б) получается из графа а) одноразовой перекруткой окружности.

Если, вы хорошо уяснили понятие гомеморфизма, то понять критерий планарности будет нетрудно. Оказывается все непланарные графы строятся с помощью только двух основных. Они на рис. 4.10

А теперь критерий планарности.

Теорема Понтрягина – Куратовского. Граф планарен тогда и только тогда, когда он не содержит подграфа, гомеоморфного двум элементарным непланарным.

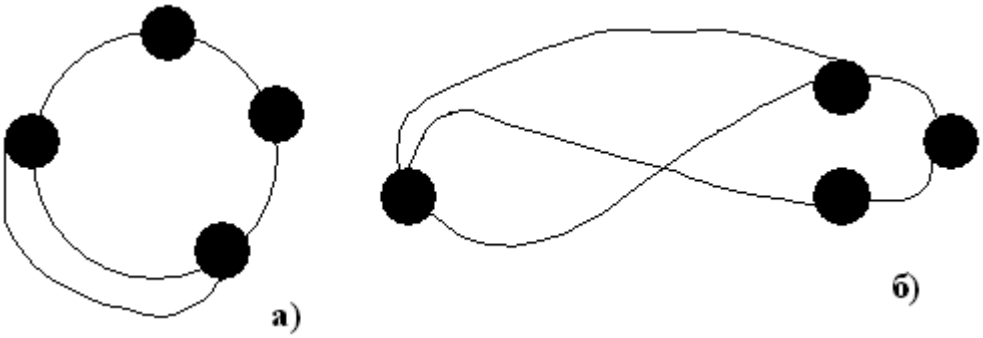

Рис. 4.9. Еще один пример гомеоморфизма

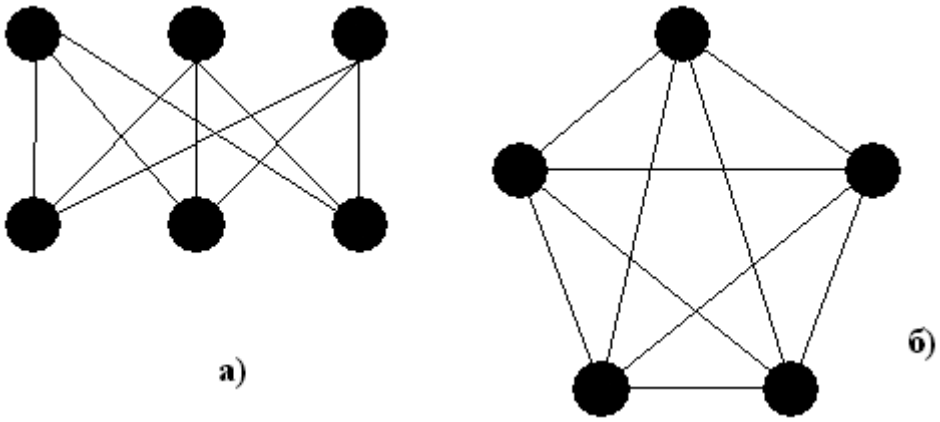

Рис. 4.10. Основные непланарные графы

Пример непланарного графа. Рассмотрим следующий граф (рис. 4.11):

Этот граф содержит в себе в качестве подграфа граф типа **б)** и следовательно по теореме Понтрягина он не планарен.

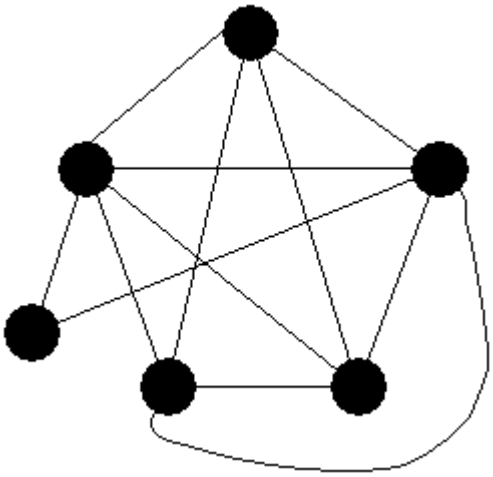

Рис. 4.11. Пример непланарного графа

Представление графа матрицами

Выше было дано формальное определение графа и было сказано, что картинка иллюстрирующая граф и его свойства это не граф, а наглядное изображение графа. Зачастую же о картинке говорят, как о самом графе. Это удобно, но все же необходимо понимать, что изображение математического объекта и сам математический объект это не одно и тоже.

Дело в том, что граф допускает еще одно представление, не такое наглядное, но быть может более полезное. Это представление называется матрицей (таблицей) инцидентности.

Из формального определения следует, что граф состоит из двух видов компонентов: вершин и ребер (или дуг). Если ребро входит или выходит из вершины, то говорят, что эта вершина и это ребро инциденты друг другу.

Инцидентность (или неинцедентность) всех пар ребер и вершин можно записать в виде таблицы. Покажем на примере как это делается. Возьмем простейший неориентированный граф. Вот такой (рис. 4.12):

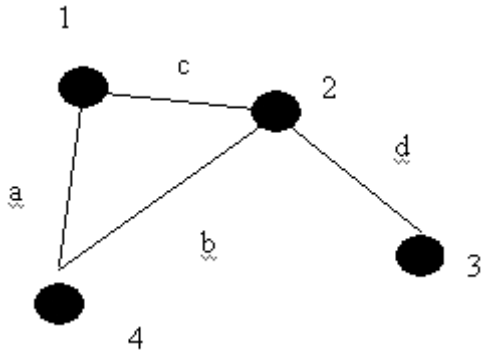

Рис. 4.12. Пример для построения матриц

Этот граф состоит из четырех пронумерованных вершин и четырех ребер обозначенных буквами a, b, c, d. Матрица инцидентности для этого графа выглядит так:

Таблица 4.1. Матрица инцидентности

	a	b	c	d
1	1	0	1	0
2	0	1	1	1
3	0	0	0	1
4	1	1	0	0

Дадим строгое определение матрицы. Матрицей инцидентности называется матрица a[i , j] такая, что:

a[i,j] = 1 если i-ое ребро инцидентно j-ой вершине
a[i,j] = 0 если i-ое ребро не инцидентно j-ой вершине

Конечно, данное определение работает только для неориентированных графов. Для ориентированных определение выглядит так:

a[i,j] = 1 если i-ая дуга входит в j-ую вершину
a[i,j] = -1 если i-ая дуга выходит из j-ой вершины
a[i,j] = 1 если i-ая дуга неинцидентна j-ой вершине

Необходимо заметить, что для ориентированного графа с петлями (то есть такими дугами, которые входят в ту же вершину из которой и выходят) матрицу инцидентности не построить, так как одна и та же ячейка в таблице должна быть равна и 1 и -1. В этих случаях строят две матрицы, одна содержит информацию о входящих дугах, а вторая о выходящих.

Еще одна форма матричного представления графа называется матрицей смежности. Эта матрица содержит информацию о парах вершин соединенных ребрами (дугами). Граф из предыдущего примера может быть представлен матрицей смежности так:

Таблица 4.2. Матрица смежности

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	0
4	1	1	0	0

Представление графа матрицами смежности и инцидентности помогает решать многие задачи. Для примера рассмотрим задачу поиска пути из пункта A в пункт B. Пункты A и B это города на карте, на которой есть и еще города. Некоторые из них соединены дорогами, некоторые нет. В общем случае такую карту можно представить в виде графа с циклами. Так как по каждой дороге можно ехать, как в одну, так и в другую сторону, то этот граф конечно неориентированный. Представим этот граф в виде матрицы смежности.

Тогда из построенной матрицы легко определить, соединены ли два города дорогой непосредственно или нет. Напомним, для последующих рассуждений, что название каждого города присутствует, в верхней строке матрицы и в левом столбце. Вот так:

Таблица 4.3. Матрица смежности

	A	B
A		
B		

Только конечно в матрице описывающей реальную ситуацию, строк и столбцов будет значительно больше. Если нас интересует соединены ли два города дорогой, мы можем поступить так:

1. Имя первого найти в верхней строке матрицы. Соответствующий ему столбец обозначим как СТОЛБЕЦ.
2. Имя второго найти в левом столбце. Соответствующую ему строку обозначим как СТРОКА.
3. Найдем пересечение СТРОКИ и СТОЛБЦА. Если на пересечении стоит единица, то два города соединены дорогой, иначе нет.

Нумерации на графе

Множество вершин графа никак не упорядочено, мы не можем сказать какая вершина первая, какая вторая и т.д. Это плохо, если бы вершины графа пронумеровать, то работать с графом было бы намного проще. Поэтому сейчас мы займемся разработкой способов нумерации вершин. Таких способов существует достаточно много, мы рассмотрим только один из них называемый – базисной нумерацией.

Базисная нумерация

Базисная нумерация строится на способах обхода графа в глубину. Вот алгоритм такого обхода:

1. Находясь в ОЧЕРЕДНОЙ вершине, ищем, среди вершин связанных с данной еще не пройденную.
2. Если таковая вершина нашлась, то переходим в нее, запоминая дугу по которой пришли.
3. Если таковой вершины нет, то из ОЧЕРЕДНОЙ вершины возвращаемся в ПРЕДЫДУЩУЮ, по дуге которая была запомнена на предыдущем шаге.

Нумерация при обходе по описанному выше алгоритму строится так: узлы графа нумеруются в порядке, в котором они посещаются первый раз в процессе обхода графа. Нумерация, построенная таким образом для ориентированных графов обладает несколькими свойствами:

Свойство 1: После завершения обхода орграфа при поиске в глубину частичный граф, образованный отмеченными дугами, есть корневое дерево с корнем в начале обхода.

Свойство 2: Для того чтобы после завершения обхода орграфа при поиске в глубину все вершины оказались пройденными, необходимо и достаточно, чтобы обход начинался во входе и этот вход был единственным. При этом корневое ордерево, образованное отмеченными дугами, имеет множество вершин, совпадающее со всем множеством вершин графа, и тем самым является каркасом графа, называемым деревом поиска в глубину.

Свойство 3: Текущий путь есть простой путь в орграфе. Его можно продолжить либо до тупикового, т.е. заканчивающегося висячей вершиной, либо до циклического, то есть до встречи с уже пройденной вершиной.

Заключение

Если у вас хватило терпения на проработку всего материала этой книги, то разрешите поздравить вас с приобретением устойчивых навыков решения задач и хорошего усвоения языка Компонентный Паскаль. Однако вы должны понимать, что путь к мастерству достаточно долог и если эта книга ваш первый учебник, то вам предстоит еще довольно большая работа. Во-первых, необходима большая, непрерывная практика в решении задач. Если КП ваш первый язык, то он не должен стать последним. Конечно, языка Паскаль и одной среды программирования вполне достаточно для любой задачи, но хороший программист должен быть многоязычным.

Наша книга не описывает все возможные методы и технологии программирования, поэтому, вам потребуется потратить время на изучение объектно-ориентированного программирования, программирования как реакции на события. Технологический набор современного программиста сложен и разнообразен. Кроме того, наша область знания постоянно развивается, появляются новые технологии и новые знания.

Но наверное наиважнейшее дело это изучение математики. Программирование, как область знания исторически вышло из математики, сейчас связь с математикой несколько ослабла, достаточно много прикладных задач практически не нуждаются в знании математики, но хорошее знание математического аппарата значительного расширит ваши возможности. Ну и конечно самое главное это практика, практика и еще раз практика.