7. Сложные типы данных

Сложность программы растет до тех пор, пока не превысит способности программиста.

6-й закон программирования

Значения сложных типов (composite data types, aggregate data types) не являются скалярными, т.е. они, с точки зрения языка программирования, включают в себя другие величины. Ясно, что в различных языках программирования сложные типы данных реализованы по-разному. Исходя из этого, полезнее сначала рассматривать абстрактные структуры (типы) данных, когда рассматриваются только их самые существенные свойства (семантика) с точки зрения программиста. Далее надо уточнять, как эта абстрактная структура данных реализована в конкретном языке программирования. Так мы и сделаем при описании сложных типов данных.

7.1. Массивы

Самая серьезная потребность есть потребность познания истины.

Георг Фридрих Гегель

Массивы состоят из элементов (elements). Абстрактный тип данных массив имеет следующие свойства.

1. Все элементы массива одинаковые (имеют один тип).
2. Элементы упорядочены, у каждого (кроме первого и последнего в массиве) есть следующий и предыдущий элементы. Чаще всего они просто пронумерованы. но не обязательно числами, например, кресла в самолёте «нумеруются» буквами A, B, C, D и т.д. Как мы знаем, свойством иметь следующее и предыдущее значения имеют величины дискретных (или порядковых) типов. Таким образом, для нумерации элементов массива подойдёт любой дискретный тип. Номер элемента называется его индексом, а сами элементы часто называются переменными с индексами. Индекс массива выступает в качестве селектора элемента, селектор позволяет выбрать из массива конкретный элемент.
3. Сам массив может иметь имя, но элементы массива являются *безымянными* переменными.

Многие свойства у абстрактного массива остаются неопределёнными:

1). Может ли массив быть пустым, т.е. не содержать ни одного элемента? Большинство языков программирование это запрещают. Например, в стандарте Паскаля в качестве индексов используются дискретные типы, а тип по определению не может быть пустым. В то же время в языке C, где значением массива по существу является *ссылка* на начало массива в памяти ЭВМ, можно «ухитриться» сделать и пустую ссылку. В языке Free Pascal есть так называемые динамические массивы, которые могут, в частности, быть и пустыми.

2). Как индексируются элементы массива? В Паскале индексом может быть любой дискретный тип, так что элементы можно «нумеровать» не только целыми числами, но и символами, логическим значениями, а также константами перечислимых типов. В большинстве языков, однако, индексами могут быть только целые числа, причём нумерация элементов начинается либо с единицы (например, в Фортране [1]), либо с нуля (языки C, JavaScript и другие). Далее, могут ли индексы занимать не один диапазон, скажем, можно ли описать массив, содержащий элементы с индексами от 1 до 10 и от 20 до 30, например, что-то вроде: `X: array[1..10:20..30] of char;` ? Обычно так делать нельзя, однако, скажем, в Фортране можно сделать нечно подобное («вырезку» из массива).

3). Может ли число элементов массива изменяться во время счёта? В *статических* массивах число элементов определяется до начала счёта и меняться не может, в стандарте Паскаля есть только статические массивы. В уже упоминавшихся выше *динамических* массивах число элементов может меняться. Ясно, что уменьшить размер массива легко, просто «лишние» элементы перестают использоваться. А вот увеличить массив сложно, в боль-

---

[1] В Фортране нумерация элементов массива с единицы производится по умолчанию, однако программист, как и в Паскале, может задать любой (но только целочисленный) диапазон индексов.

шинстве языков (и в языке Free Pascal) для этого сначала порождается новый массив бóльшего размера, затем из старого массива в новый копируются все элементы, и, наконец, старый массив уничтожается (занимаемая им память объявляется свободной).

4). Массивы могут быть многомерными, например, матрицами. Как хранить *двумерную* матрицу в *одномерной* памяти ЭВМ? Большинство языков, как говорят, линеаризуют матрицу построчно, т.е. сначала в памяти ЭВМ располагается первая строка, следом за ней вторая и т.д. В то же время язык Фортран и Бейсик линеаризует матрицу по столбцам: сначала располагает в памяти первый столбец, потом второй и т.д. Далее, все ли строки матрицы содержат одинаковое число элементов? В большинстве языков это так, но есть языки (C, Free Pascal и др.) где можно «ухитриться» создать матрицу, строки которой разной длины (или вообще пустые). В этом случае матрица больше похожа на текст из строк переменной длины.

5). Могут ли массивы менять свою размерность? Например, можно ли описать в программе матрицу, но обращаться к ней как к вектору? В этом случае можно считать, что все строки (или столбцы), как то расположенные в памяти друг за другом, образуют один вектор. И, наоборот, можно ли обращаться к вектору как к матрице? В большинстве языков это запрещено, но, например, в Фортране так делать можно.

6). Хранятся ли элементы одномерного массива в памяти ЭВМ вплотную друг за другом? Для большинства языков программирования это так, такие массивы иногда называются *упакованными* (packed). Возможно, онако, что для ускорения чтения и записи под каждый элемент массива длиной три байта будут отводится четыре байта памяти компьютера, обращение к таким элементам может производиться в ЭВМ быстрее.

Прежде, чем работать с массивом, в Паскале надо описать тип этого массива. Синтаксис типа массива в стандарте Паскаля:

```
<тип массива> ::= array[<тип индекса>] [1] of <тип элемента>;
<тип индекса> ::= <дискретный тип>
```

Итак, тип индекса задаёт способ нумерации элементов массива, он же однозначно определяет число элементов в массиве, оно известно *до начала счёта* и меняться не может. Далее задаётся тип элемента массива, все элементы одного типа. Например:

```
array[1..100] of integer;
array[char] of boolean;
array[true..true] of real;
array[1..8] of (Up,Down,Left,Right);
```

Все описанные выше типы определяют конечные безымянные наборы величин. Например, если в типе `char` 256 символов, то тип массива `array[char] of boolean;` содержит 2^{256} различных массивов. Это соизмеримо с количеством элементарных частиц во Вселенной, но всё-таки конечно. Работать с безымянными наборами величин не всегда удобно (а в некоторых случаях и невозможно), поэтому следует дать им имена в разделе описания типов, Например:

```
type Arr100=array[1..100] of integer;
     DigitArrOfReal=array['0'..'9'] of real;
     Arrow=array[1..8] of (Up,Down,Left,Right);
```

В разделе переменных можно описывать переменные-массивы как именованного, так и безымянного типов, например:

```
var x,y: Arr100; a: array[char] of boolean;
```

Для указания конкретного элемента массива используется селектор в виде индекса в квадратных скобках, например:

```
x[i]:=y[i+20]; a['Z']:=true;
```

---

[1] Здесь в описании массивов квадратные скобки используются как символы языка Паскаль, а не как служебные символы метаязыка БНФ (для задания необязательности конструкции).

При выходе индекса элемента за границы массива (например, при обращении `x[0]`) результат в стандарте Паскаля не определён. В языке Free Pascal всё зависит от текущего режима работы, для режима `{$R+}` будет исключительная ситуация, а для `{$R-}` возникнет так называемое неопределенное поведение программы, о чём мы уже говорили (см. разд. 3.1.2).

Отметим, что есть языки, где все массивы динамического класса памяти, в таких языках при выходе индекса за границу массива программа может вести себя странно. Например, в языке Swift в этом случае просто автоматически увеличивает размер массива, в него добавляется необходимое число новых элементов (с неопределёнными значениями). А некоторые языки в этом случае ведут себя ещё более странно, например, если для массива из 10 элементов `X[1..10]` обратиться к `X[20]`, то в массив добавляется только один (двадцатый) элемент, т.е. будет `X[1..10,20]`  . Вообще-то, это уже и не массив…

Отметим, что в описании

```
var X: array[1..1] of integer; Y: integer;
```

имя `X` задаёт массив из одного элемента, который не совместим по присваиванию со скалярной переменной `Y`, т.е. оператор `X:=Y` будет неправильным.

В стандарте Паскаля все переменные (и массивы) порождаются с неопределёнными значениями. В языке Free Pascal переменные *статического* класса порождаются с «нулевыми» значениями (очищаются), а переменные *автоматического* и *динамического* классов памяти – с неопределёнными значениями. При порождении массива не очень большого размера можно указать конкретные значения его элементов, например

```
var x: array[1..5] of integer=(3,-10,6,1,3);
```

В качестве примера рассмотрим такую задачу. Необходимо ввести 1000 вещественных чисел и вывести их сумму. Обычно учащиеся осознают, что для решения этой задачи не нужен массив для хранения вводимых чисел и предлагают такой вариант программы (эта же программа приведена и в большинстве учебников по программированию и на сайтах в Интернете):

```
  const n=1000;
  var x,sum: real; i: integer;
begin
  sum:=0.0;
  for i:=1 to n do begin
    read(x); sum:=sum+x
  end;
  Writeln(sum)
end.
```

Как люди уже опытные, сначала зададимся вопросом, когда (т.е. в каком предусловии) эта программа является правильной? Естественно предложить такое предусловие для этой задачи:

$$Pred=\{\forall i\in 1..n \mid \underline{x_i}\in R \ \wedge \ \Sigma\underline{x_i}(i\in 1..n)\in R\}$$

Необходимо осознать, что в этом предусловии написанная выше программа *неправильная*. Действительно, в процессе суммирования *частичная сумма* может выйти за допустимый диапазон представимых вещественных чисел, и правильного ответа не получится. При выполненном предусловии это может случиться, например, если сначала вводятся в основном положительные числа, а затем отрицательные, так что общая сумма принадлежит R, но не будет получена программой  .

На самом деле написанная выше программа верна в таком предусловии:

$$Pred=\{\forall x_i\ (i\in 1..n) \mid \underline{x_i}\in R \ \wedge \ \forall j\le n \mid \Sigma\underline{x_i}(i\in 1..j)\in R\}$$

Другими словами, допустимой должна быть любая частичная сумма. Это предусловие, конечно, для заказчика плохое, так как многие *представимые* суммы программой не будут получены. Как же,

однако, написать программу в хорошем первоначальном предусловии? Необходимо понять, что без использования массива, в котором будут храниться все введённые числа, при этом уже не обойтись.[1]

Новая программа сначала должна будет ввести все числа в некоторый массив, а затем упорядочить его по не убыванию. После этого надо организовать «встречное» суммирование, двигаясь одновременно от начала к концу и от конца к началу массива. Ниже приведена возможная программа (она будет выдавать аварийный останов только при выходе всей суммы за допустимый диапазон).

```
  const n=1000;
  type Mas=array[1..n] of real;
  var i,j: integer; x: Mas; sum,temp: real;
begin
  for i:=1 to n do read(x[i]);
{ простое упорядочивание массива по не убыванию }
  for i:=1 to n-1 do
    for j:=i+1 to n do
      if x[j]<x[i] then begin
         temp:=x[i]; x[i]:=x[j]; x[j]:=temp
      end;
  sum:=0.0; i:=1; j:=n; { ↓↓ встречное суммирование }
  while i<=j do
    if sum>0 then begin sum:=sum+x[i]; i:=i+1 end
             else begin sum:=sum+x[j]; j:=j-1 end;
  Writeln(sum)
end.
```

Еще одной широко распространённой задачей является нахождение среднего арифметического числа в последовательности, для которой во многих учебниках приводится примерно такое решение.

```
  const n=1000;
  var x,SA: real; i: integer;
begin
  SA:=0.0;
  for i:=1 to n do begin
      read(x); SA:=SA+x;
  end;
  SA:=SA/n; Writeln(SA)
end.
```

Предусловие, в котором написанная программа будет правильной, следующее

$$Pred=\{\forall i\in 1..n \mid \underline{x_i}\in R \ \wedge \forall j\le n \mid \Sigma\underline{x_i}(i\in 1..j)\in R\}$$

Таким образом, для подавляющего большинства введённых массивов написанная программа *неправильная*.[2] В то же время, если все числа последовательности существуют, то среднее арифметическое обязательно существует ⚠, вне зависимости от существования суммы всех чисел. Таким образом, «хорошее» (и естественное) предусловие будет таким:

$$Pred=\{\forall i\in 1..n \mid \underline{x_i}\in R\}$$

Правильной в таком «хорошем» предусловии, например, будет такая программа, в которой можно обойтись без массива. В этой программе мы будем делить на число элементов массива n не всю сумму в конце, а каждый элемент по отдельности:

```
  const n=1000;
  var x,r,SA: real; i: integer;
```

---

[1] Если вводимых чисел не очень много, и они помещаются в массив (расположенный в оперативной памяти компьютера), то использования для хранения этих чисел других структур данных (файлов, списков и т.д.) не рассматривается, как крайне неэффективное.

[2] В оправдание авторов таких примеров можно заметить, что в большинстве задач обрабатываемые числа редко бывают очень большими.

```
begin
  SA:=0;
  for i:=1 to n do begin
      read(x); SA:=SA+x/n
  end;
  Writeln(SA)
end.
```

К сожалению, при это резко возрастёт число команд деления. Рассмотренные примеры призваны показать, с какой осторожностью программист должен относится к «очевидным» решениям даже самых простых задач.

Рассмотрим теперь многомерные массивы. Из описания синтаксиса следует, что элемент массива может быть любого типа, в частности, он снова может быть массивом, например:

```
var X,Y: array[1..10] of array[1..20] of integer;
```

описывает переменные X и Y как двумерные массивы (матрицы) целых чисел размером 10 строк на 20 столбцов. Переменные X и Y совместимы по присваиванию, т.е. допустим оператор `X:=Y`. Далее, каждая строка этих матриц тоже переменная, и они совместимы по присваиванию между собой, т.е. можно, например, писать `X[3]:=Y[5]`.[1] И, наконец, элементы матриц тоже являются (целочисленными) переменными, например, `X[3][4]:=Y[5][6]` или `X[7][8]:=0`.

В стандарте Паскаля столбцы матриц переменными не являются. Есть языки программирования, в которых и столбцы матриц являются переменными, например в старом языке PL/I можно было писать `X[*,4]=Y[*,6]` для присваивания четвёртому столбцу матрицы X значение шестого столбца матрицы Y. А вот в языке Fortran-90 можно работать с любым минором (сечением) матрицы (с символа `!` в Фортране начинается комментарий):

```
INTEGER X(10,20),Y(10)  ! Примеры сечений массивов
X(1:10,3)     ! Третий СТОЛБЕЦ X
X(4,1:20)     ! Четвёртая СТРОКА X
X(2:6,3:13)   ! Минор 5x10
Y(1:10:2)     ! Вектор из 4-х элементов: 2-го, 4-го, 6-го и 10-го
Y(/9,2,6/)    ! Вектор из 3-х элементов: 9-го, 2-го и 6-го ⚠
```

Вот что значит язык для физиков 🙂. Далее мы рассмотрим нечто похожее и в языке Free Pascal.

При работе с многомерными массивами в Паскале допускаются условные обозначения для сокращения записи, можно писать

```
var X,Y: array[1..10,1..20] of integer;
```

и употреблять сокращённую запись для элементов матрицы, например `X[7,8]:=0`, при этом полная и сокращённая записи являются эквивалентными. Это удобно, а вот, например, в языке C так не сделано и всегда надо писать `int X[10][20];` 🙁

Тип индекса лучше записывать с использованием *именованных* констант, например

```
const N=10; M=20;
var X,Y: array[1..N,1..M] of integer;
```

Предостережём, однако, о серьёзной ошибке, допускаемой учащимися, нельзя писать

```
var N: integer; Z: array[1..N] of integer; { ОШИБКА }
```

Здесь надо понять, что запись `1..N` является ограниченным типом, в описании которого могут участвовать только константы, а в последнем примере N является переменной.

Рассмотрим вопрос о совместимости массивов по присваиванию. Пусть есть описание массивов:

---

[1] Так будет только тогда, когда в конкретной реализации допускается (необязательная) структурная совместимость по присваиванию. Переменные совместимы структурно, если написание их типов совпадают. Например, у нас переменные `X[3]` и `Y[5]` обе имеют (безымянный) тип `array[1..20] of integer;`. В языке Free Pascal структурная совместимость разрешена.

```
type Mas1=array[1..10] of integer;
     Mas2=array[1..10] of integer;
     Mas3=Mas1;
var A,B: Mas1; C1: Mas1; C2: Mas2; C3: Mas3;
    D,E: array[1..10] of integer;
    F: array[1..10] of integer;
```

В стандарте Паскаля массивы A,B,C1 и C3 совместимы между собой по присваиванию (у них, как говорят, именная совместимость, иногда говорят эквивалентность), но не совместимы с остальными массивами C2, D, E и F. Аналогично, массивы D и E совместимы по присваиванию между собой (это тоже именная совместимость, хотя имени то здесь и нет 😕), но не совместимы с остальными массивами. И, наконец, массив F не совместим по присваиванию со всеми остальными массивами. В языке Free Pascal, как уже говорилось, допускается дополнительная, структурная совместимость (эквивалентность) по присваиванию, поэтому в этом языке все описанные выше массивы совместимы между собой по присваиванию.

Отметим, что структурная совместимость менее надёжна, поэтому именная совместимость достаточно широко распространена в языках программирования, например, она есть в языках Ada, C++, C#, Java и других.

7.2. Строки символов

Строка – это застывшая структура данных, и повсюду, куда она передается, происходит значительное дублирование процесса. Это идеальное средство для сокрытия информации.

Алан Перлис,
первый лауреат премии Тьюринга

Теперь рассмотрим важный случай одномерных массивов, элементами которых являются символы, такие массивы обычно называются строками. Вспомним, что мы уже знаем о строковых константах, являющихся лексемами языка (непустой последовательностью символов в апострофах). Теперь осталось изучить строковые переменные, которые могут хранить такие константы.

Итак, для хранения значения строковой константы 'ABCD' в стандарте Паскаля надо описать переменную

```
var X: packed array[1..4] of char;
```

Как видим, описание строковой переменной содержит дополнительное служебное слово **packed**, это описывает строковую переменную как *упакованный* массив символов. Слово «упакованный» указывает Паскаль-машине отвести под хранение этой переменной как можно меньше памяти ЭВМ. Упакованные и неупакованные переменные считаются в стандарте Паскаля несовместимыми по присваиванию. А вот наш язык Free Pascal снимает почти все ограничения работы со строками одинаковой длины, но результаты будут странными, например:

```
var x: array[1..10] of char;
    y: packed array[1..10] of char;
begin x:='ABCD'; { здесь x[5..10]=#0 🐷 }
      y:=x; y:='EF'; { y[3..4]='CD' }
```

В языке Free Pascal есть также служебное слово **bitpacked**, оно предписывает провести и упаковку элементов, для которых можно отвести меньше одного байта памяти (скажем, для типа boolean достаточно одного бита). Например

```
var A: array[0..255] of boolean; { 256 байт }
    B: bitpacked array[0..255] of boolean; { 32 байта }
```

При этом чтение и запись таких элементов может занять в несколько раз больше времени, чем для неупакованных данных. Работа с упакованными структурами данных подробно рассматривается в курсе по архитектурам ЭВМ. Кроме того, компоненты упакованных структур данных (элементы

массивов, поля записей) перестают быть полноценными переменными ⚠ (например, их уже нельзя передавать в подпрограммы по ссылке).

И вот теперь в стандарте Паскаля для переменной

```
var X: packed array[1..4] of char;
```

будет правильным оператор присваивания `X:='ABCD'`. В то же время оператор `X:='ABC'` будет неправильным, так как есть несоответствие типов: константа принадлежит к типу

```
packed array[1..3] of char;
```

Таким образом, длина переменной и константы должны совпадать. Отметим, что для переменной `Y`, описанной как

```
var Y: array[1..4] of char;
```

оператор `Y:='ABCD'` будет в стандарте Паскаля тоже неправильным (опять несоответствие типов, нет слова **packed**).

В языке Free Pascal упакованные и неупакованные массивы символов (даже с разным количеством символов) считаются совместимыми по присваиванию, а после оператора `X:='AB'` две последние позиции массива не изменяются.

Почему работа со строковыми переменными в стандарте Паскаля такая неудобная?! Здесь всё дело в надёжности программы: если допустить, чтобы оператор `X:='ABC'` был правильным, то как будет, например, работать условный оператор

```
if X[4]='*' then S { ? }
```

Здесь `X[4]` ничему не равно, получается типичное неопределённое поведение программы, аналогичное использованию, например, `X[5]`. Для обнаружения ошибки `X[5]`, однако, можно включить режим контроля `{$R+}` выхода за допустимый диапазон, а для `X[4]` уже ничего не поможет 😕.

Во всех языках, однако, существуют более удобные для программиста, но менее надёжные, так называемые строки переменной длины. Для таких строк существуют два различных механизма их хранения в памяти ЭВМ, обычно они называются строками с явной и неявной длиной.

- Строки с явной длиной.

Всё между строк.
Остальное не считается.
Вэй Дэ-Хань
«Книга беспредельной мудрости»

Во-первых, можно хранить длину строки явно в виде префикса в начале такой строки (length-prefixed strings), это строки с явной длиной. В языке Free Pascal для этого предназначен тип `string` (есть синоним `ShortString`, намёк на то, что длина таких строк ограничена числом 255). Например, опишем две такие строки, каждая из которых может хранить не более восьми символов:

```
var X,Y: string[8];
```

Для таких переменных отводят в памяти место, необходимое для хранения строки символов максимальной нужной длины, но допускают записывать в это место и строки *меньшей* длины. Конечно, каждая такая строка должна знать, сколько символов она хранит в каждый момент времени. Обычно для этого к таким строкам впереди добавляется служебный элемент массива с нулевым индексом, в нём хранится символ с текущей длиной строки, например, после оператора `X:='ABC'` строка будет иметь такой вид:

	0	1	2	3	4	5	6	7	8
X:	#3	A	B	C	?	?	?	?	?

Как видно, в нулевом элементе массива хранится символ, порядковый номер которого равен текущей длине строки, эту длину можно определить, вычислив выражение `ord(X[0])`. Программисту для вычисления длины строки удобнее пользоваться стандартной функцией `length(X)`, которая возвращает текущую длину любого (не обязательно символьного) массива. Обратите внимание, что, начиная с четвёртой позиции символы строки не имеют конкретных значений, попытка, например, анализировать `X[4]` приведёт к неопределённому поведению программы.

Для работы с такой «ненадёжной» структурой данных программисту предлагается не анализировать символы строки *напрямую*, с помощью их индексов, а использовать в работе предоставляемые языком новые стандартные возможности. Здесь мы рассмотрим только часть этих возможностей.

Во-первых, у полиморфной операции `+` добавляется новый смысл – конкатенация (сцепление) двух строк, например, после оператора `X:=X+'1*A'` переменная X будет выглядеть так

	0	1	2	3	4	5	6	7	8
X:	#6	A	B	C	1	*	A	**?**	**?**

Конкатенация выполняется «разумно» и никогда не даёт неопределённых результатов, например, после следующего оператора `X:=X+'BCDE'` переменная X будет выглядеть так

	0	1	2	3	4	5	6	7	8
X:	#8	A	B	C	1	*	A	B	C

Как видим, не допускается выход строки за её максимальный размер, и лишние символы просто отбрасываются, и никакого контроля выхода индекса строки за максимальный диапазон не производится, т.е. директива `{$R+}` здесь *не работает*. Ясно, что надо осторожно пользоваться этим ненадёжным механизмом.

Можно, конечно, «обрезать» строку, изменив её нулевой символ, например, после присваивания `X[0]:=#4` (можно написать и более понятно `setlength(X,4)`) и оператор `write(X)` выведет `ABC1`, а строка X будет иметь вид

	0	1	2	3	4	5	6	7	8
X:	#4	A	B	C	1	*	A	B	C

Учтите, что вызов `setlength(X,10)` приведёт к «странному» результату

	0	1	2	3	4	5	6	7	8
X:	#10	A	B	C	1	*	A	B	C

Здесь Free Pascal будет считать, что к строке «присоединены» два байта из области памяти, расположенной сразу вслед за переменной X. Содержимое этих байт, обычно никому не известно, в частности, там могут содержаться важные данный из переменной, расположенной вслед за нашей строкой (у нас там описана строка Y). Всё это показывает, насколько такие строки являются ненадёжной структурой данных.

Заметим, что в языке Free Pascal, в отличие от стандарта, допускаются и пустые строки, например после оператора `Y:=''` переменная Y будет иметь вид

	0	1	2	3	4	5	6	7	8
Y:	#0	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**

Как видим, здесь возникает естественное ограничение на максимальную длину строки, она на единицу меньше, чем число символов в алфавите (так как символы нумеруются, начиная с нуля). Для большинства реализаций Паскаля число символов в алфавите 256,[1] поэтому максимальная длина строки 255. Длину строки можно опускать и писать просто `string`, в этом случае в режиме работы `{$LongStrings OFF}` (короткий синоним `{$H-}`) будет предполагаться `string[255]`, а в режиме `{$LongStrings ON}` (короткий синоним `{$H+}`) специальный динамический строковый тип `AnsiString`, о котором будет немного рассказано далее.

Теперь рассмотрим полезную стандартную функцию `pos(s1,s2)`. Для своих параметров-строк эта функция возвращает позицию, начиная с которой строка `s1` входит в строку `s2`. Когда вхождения нет, то возвращается число ноль, а когда есть несколько вхождений, то возвращается позиция *первого* из них. Отметим, что эта функция работает точно так же, как и операция поиска вхождения в Нормальных Алгоритмах Маркова, например, для `X='ABCD1*AB'` будет

---

[1] Как уже упоминалось, есть и «большой» алфавит Unicode, в языке Free Pascal строки в этом алфавите имеют тип `WideString`, мы с ним работать не будем (процедура `write` выводит их только в графических окнах). Кроме того, как говорилось в разделе 6.2, в языке Free Pascal есть 255 основных символов с 8 битной кодировкой и 256 дополнитедьных символов с 16 битной кодировкой (с первым нулевым байтом).

```
pos('B',X)=2          pos('AB',X)=1       pos('CB',X)=0
pos('B','CCBB')=3     pos('','123')=1     pos('','')=1
```

Как видно, по определению пустая строка входит в любую другую строку (включая и пустую), начиная с первой позиции.

Теперь рассмотрим полезную стандартную функцию `copy(s,n1,n2)`, которая в качестве результата возвращает *подстроку* из первого параметра `s`, начиная с позиции `n1` и содержащую `n2` символов, например, для строки `X='ABCD1*AB'` будет

```
copy(X,2,3)='BCD'      copy(X,-2,4)='ABCD'
copy(X,7,5)='AB'       copy(X,5,-2)=''
```

Итак, попытка задать параметры копирования, выходящие за текущие размеры строки игнорируются, и выдаются только те символы, которые присутствуют в строке. В языке Free Pascal есть и другие стандартные функции работы со строками, но мы их рассматривать не будем, отметим только полезную процедуру `delete(s,n1,n2)` для удаления из строки `s` подстроки, начиная с позиции `n1` и содержащую `n2` символов.

В языке Free Pascal можно подключит модуль с именем `StrUtils` (**uses** `StrUtils`), в котором содержатся много других полезный операций над строками, например, функция:

```
DupeString('123',4)
```

возвращает «учетверённую» строку `'123123123123'`.

Попробуем теперь в качестве примера работы со строками реализовать алгоритм работы исполнителя Нормальных Алгоритмов Маркова. Сначала, конечно, сделаем необходимые ограничения, без которых реализовать абстрактный исполнитель не представляется возможным. Как Вы должны помнить, программа для НАМ записывается в виде

$$\begin{aligned} &\alpha_1 \to \beta_1 \\ &\alpha_2 \to \beta_2 \\ &\quad \dots \\ &\alpha_n \to \beta_n \end{aligned}$$

Сначала определим, как будем хранить программу. Сделаем ограничение, что число правил `n` не будет превышать `Nmax=100`, а длина каждого α_i и β_i не более 255 (чтобы хранить их в переменных типа `string`). Тогда правила будем хранить в виде трёх массивов: массива левых частей `A`, массива правых частей `B` и массива стрелок `ST`. Сделаем спецификацию, что при превышении длины обрабатываемого слова остановимся с выдачей аварийной диагностики. Получится такая программа

```
const Nmax=100;
var A,B: array[1..Nmax] of string; { αi и βi }
    ST: array[1..Nmax] of (Ter,NTer);
{ Стрелки перечислимого типа Ter = ↦ и NTer = → }
    W: string; { Обрабатываемое слово }
    n,i,k,LA,LB: integer; { Прочие переменные }
begin
{ Здесь ввод программы и входного слова W
  Необходим контроль входных данных и диалоговый интерфейс }
repeat { Главный цикл работы НАМ }
  i:=0; { Номер правила }
  repeat { Цикл по правилам }
    i:=i+1; k:=pos(A[i],W); { Входит ли αi в W ? }
    if k>0 then begin { Входит, W=W1αiW2 → W1βiW2 }
      LA:=length(A[i]); LB:=length(B[i]);
      if length(W)-LA+LB<=255 then
         W:=copy(W,1,LA-1)+B[i]+copy(W,k+LA,(1) 255)
      else begin { Аварийный выход } i:=n; k:=0;
         W:='Превышена max длина слова'
      end
```

```
    end
  until (k>0) { Была замена } or (i=n) { Больше правил нет }
until (k=0) { Все правила не подошли } or
      (ST[i]=Ter); { Была стрелка ↦ }
Writeln('Результат=',W)
end.
```

Обратите внимание, что при копировании в результат подслова W_2 мы указали не его настоящую длину, а число ❶ 255, т.е. «до конца строки», всё равно «лишние» символы браться не будут.

На этом примере показано, что для работы с такими «ненадёжными» структурами данных, как строки, нельзя «ходить» по строке, перебирая её символы (скажем, анализируя `W[i]`). Вместо этого следует использовать только предоставляемые системой программирования «надёжные» средства, в нашем случае это стандартные функции `length`, `pos` и `copy`, а также операция конкатенации "+". Итак, работая со строками, программист *ограничивает* свои действия только «надёжными» операциями, в то время, как используя все возможности языка легко совершить ошибку.

Описанный строковый тип данных очень удобен, однако имеет один серьёзный недостаток. Это, как можно догадаться, маленькая максимальная длина строки. Как здесь быть?

Ну, во-первых, можно преодолеть этот недостаток методом «грубой силы», отведя под хранение длины строки не символьную, а целую переменную. В языке Free Pascal для этого предусмотрен строковый тип `AnsiString`, в которых под хранения длины отведена переменная типа `longint` (это 4 байта, длина строки до 2^{31} символов). Переменные этого типа порождаются пустыми. При присваивании таким переменным строковых значений им автоматически отводится минимально необходимый объём памяти (они порождаются) и они автоматически уничтожаются, когда в них отпадает необходимость.[1] Отметим, что строковые константы (например, `'ABCD'`) всегда имеют тип `AnsiString`, а в режиме `{$LongStrings ON}` (короткий синоним `{$H+}`) строки `string` без заданной длины тоже трактуются как `AnsiString`. Впрочем, типы `string` и `AnsiString` взаимно совместимы по присваиванию и по всем операциям над ними. В некоторых языках строки устроены ещё более сложно, например, в языках Java, Swift и C# строки описываются *дескрипторами* примерно такого вида:

```
type
  DStr=record
    Len: longint; {текущая длина строки}
    MaxLen: longint; {max длина строки}
    Ref: Pointer; {указатель на начало строки}
  end;
```

Впрочем, внутренняя структура дескриптора чаще всего скрыта от программиста и для работы ему приходится использовать встроенные функции (например, `length`). Работа с такими строками будет понятна после изучения сылочных типов данных в главе 12.

- Строки с признаком конца.

Всё хорошо, что хорошо кончается.
Из пьесы Уильяма Шекспира

Далее, можно использовать и принципиально другие строки, так называемые строки с признаком конца (zero/null terminated strings), часто их называют строками с неявной длиной. Идея здесь совсем простая. Один из символов алфавита объявляется *служебным*, он всегда стоит *в конце* строки, но *внутрь* строки входить не может. Обычно это символ с нулевым номером в алфавите, который обозначается в Паскале как `#0`, а в языке C как `\0`.

В языке Free Pascal такие переменные имеют тип `Pchar`. Сама строка символов, конечно, является массивом, однако, в отличие от типов `string` и `Ansistring`, символы нумеруются не с единицы, а с нуля:

---

[1] Для продвинутых читателей. Переменные типа `AnsiString`, как и упоминаемые ниже переменные типа `Pchar` (Pointer to CHAR) всегда динамического класса памяти. Они являются *ссылочными* переменными, а сами строки являются *динамическими* переменными, это мы будем изучать позже. Кроме того, для таких строк работает так называемая автоматическая сборка мусора.

```
var X: Pchar='ABCD';
```

	0	1	2	3	4
X:	A	B	C	D	#0

Строки типа `Pchar`[1] (как и строки рассмотренные выше `Ansistring`), могут иметь любую длину (в пределах доступной памяти). Это, однако, их единственное достоинство перед типом `string`, зато есть два больших недостатка. Во-первых, теперь признак конца не может входить внутрь строки, а во-вторых, определить длину такой строки будет не так просто. Приходится в цикле просматривать всю строку от начала до конца, в поиске её конца. Для строки длиной, например, в несколько миллионов символов, операция определения длины несколько затянется 🙂. Более тяжёлой становится и операция конкатенации (сцепления) двух строк.[2]

Между строками `Ansistring` и `Pchar` есть ещё одно существенное отличие, касающееся операции присваивания. Переменные `Ansistring` являются строками символов, а переменные `Pchar` являются ссылками на строки символов. Приведём пример:

```
var X: Ansistring='ABCD'; Y: Ansistring;
begin Y:=X; { это одна и та же строка! }
```

X,Y:	A	B	C	D	#0

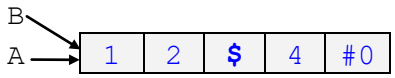
```
    Y[2]:='$'; { сейчас создаётся копия строки }
```

X:	A	**B**	C	D	#0

Y:	A	**$**	C	D	#0

Итак, как только после присвоения `Y:=X` мы попытаемся изменить `Y`, то создаётся копия `X`, которая и становится новым значением `Y`. Такая техника называется «копирование при записи» (Copy on Write). Можно принудительно сделать копию, вызвав процедуру `UniqueString(Y)`. А для типа `Pchar` это не так:

```
var A: Pchar='1234'; B: Pchar;
begin B:=A; A[2]:='$'; { копия строки не создаётся }
```



Таким образом, для типа `Pchar` ссылочные переменные `A` и `B` указывают в памяти на одну и ту же переменную-строку, так что изменение `A` является изменением `B`. Для «настоящего» копирования таких строк следует использовать функцию `B:=copy(A)`. Отметим также, что завершающий ноль в строке `Ansistring` «не функциональный», так как он не определяет длину строки и оставлен просто для совместимости с типом `Pchar`. Как уже говорилось, «настоящая» длина строк типа `Ansistring` хранится в 4-байтном поле, предшествующем строке. Как следствие, внутрь строки типа `Ansistring` (как и строки типа `string`) могут входить и нулевые символы.

Далее, прежде, чем переходить к изучению других сложных типов данных (записей, множеств и файлов) надо сначала рассмотреть описание и использование пользовательских подпрограмм.

Вопросы и упражнения

Принимаясь за дело, соберись с духом.
Козьма Прутков

1. Чем сложные типы данных отличаются от простых?
2. Что такое абстрактная структура данных?
3. Что такое статические и динамические массивы?
4. Какого типа может быть индекс массива в Паскале?

---

[1] Строки `Pchar` в основном используются для совместимости с программами, написанными на языке C. В частности, это все библиотеки операционных систем (Windows, Unix, Android и т.д.), для доступа к подпрограммам в этих библиотеках служат так называемые системные вызовы API (Application Programming Interface).

[2] Современные процессоры фирмы Intel могут работать с так называемыми векторными регистрами. Например, на регистры XMM длиной 128 бит можно считать сразу 16 символов строки и одной командой искать так нулевой символ. Это позволяет достаточно быстро искать конец строки даже в динных строках.

5. Чем отличаются индексы массивов в языках Паскаль, Фортран и C?
6. Почему элементы массива являются безымянными переменными?
7. Что происходит при выходе индекса массива за допустимый (описанный) диапазон?
8. Как в Паскале описать многомерный массив?
9. Что такое строка символов в стандарте Паскаля? Чем отличаются упакованные и неупакованные строки символов?
10. Какие достоинства и недостатки типа `string` в языке Free Pascal?
11. Почему работа с типом `string` может быть ненадёжной?
12. Какие стандартные подпрограммы предназначены для работы с типом `string`?
13. Что такое строка символов с признаком конца? Какие у неё достоинства и недостатки, по сравнению с типом `string`?