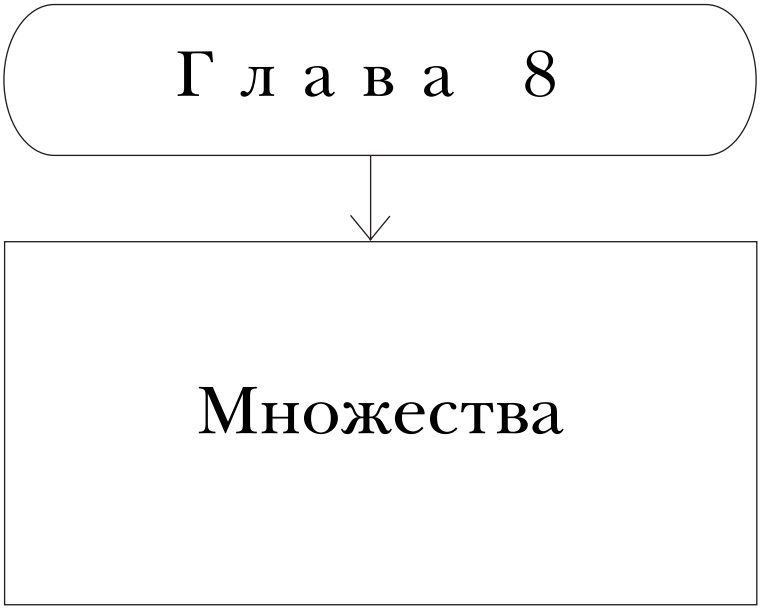

8.1. Множества и математика

Понятие “множество” в математике является одним из основополагающих, и поэтому считается неопределяемым. Содержание, смысл этого понятия можно лишь объяснить с помощью примеров и описания его свойств. Именно этим мы сейчас и займемся.

Первоначальное понятие о множестве можно получить, представив себе некую совокупность объектов произвольной природы. Объекты в составе совокупности могут быть либо абстрактными (числа, слова), либо вполне конкретными (пальцы на руках, предметы в карманах). Иначе говоря, объекты совокупности могут быть как сгенерированы нашим мыслительным процессом, так и позаимствованы из окружающего нас мира в виде реальных предметов.

Однако не любая совокупность может считаться множеством, хотя любое множество является совокупностью. Это следует из того, что объекты множества обязательно должны отличаться друг от друга. В то же время от совокупности объектов этого не требуется. Мы можем говорить о совокупности совершенно одинаковых десятикопеечных монет, полученных нами как часть зарплаты. Но говорить о множестве десятикопеечных монет мы получим право только в том случае, если нам удастся найти однозначный признак, по которому мы будем отличать их друг от друга. Таким признаком, например, может быть год выпуска монеты. Заметим, что в данном случае целесообразно использовать уже

другое название множества (например, “множество десятикопеечных монет разного года выпуска”).

Любопытно, что для абстрактных объектов совокупности чаще всего невозможно изобрести признак, по которому мы смогли бы отличить их друг от друга в том случае, если они одинаковы. Приведем пример: пусть совокупность оценок, полученных группой студентов на экзамене, включает несколько пятерок, четверок, троек и, увы, двоек. Однако множество этих оценок состоит всего из четырех перечисленных. И это потому, что такие абстрактные объекты, как “двойки” студентов Иванова и Петрова, совершенно неотличимы.

И еще одним свойством отличаются *совокупность* и *множество*. Совокупности оценок, полученных разными группами студентов, представляют собой разные совокупности. И в то же время множество оценок будет одним и тем же, независимо от того, какими группами студентов они были получены. Иначе говоря, объекты множества отличаются не только друг от друга, но и от объектов, которые в это множество не входят.

Объекты множества принято называть его элементами. Чаще всего множества обозначают прописными, а элементы множества — строчными буквами. При этом, если некий объект x является элементом множества M, то этот факт записывается в виде $x \in M$ (x принадлежит M). Противоположная ситуация обозначается как $x \notin M$ (x не принадлежит M). Например, если M — множество оценок, то $3 \in M$, а $13 \notin M$.

Число элементов множества может быть конечным или бесконечным. Исходя из этого характеризуют и множества. Множество оценок из вышеприведенного примера — конечно. Множество натуральных чисел — бесконечно. Логическим продолжением этого свойства является существование пустого множества, не содержащего элементов. Пустое множество обозначается символом $\varnothing$. Например, если M — множество действительных корней уравнения $x^2 + 1 = 0$, то $M = \varnothing$. Число элементов в конечном множестве называется *мощностью этого множества*.

Рассмотренные примеры позволяют нам определиться со способами задания множеств. Проще всего задать множество посредством указания общего свойства его элементов. Формально это выглядит так: $M = \{ x \mid P(x) \}$, т.е. множество M представляет собой множество всех x, обладающих свойством $P(x)$. Например, запись $A = \{ x \mid x$ – *натуральное число*$\}$ означает бесконечное множество A всех натуральных чисел, запись $B = \{ x \mid (x-1)(x-2)(x-3) = 0 \}$ означает конечное множество B, содержащее корни уравнения $(x-1)(x-2)(x-3) = 0$, а запись $C = \{ x \mid (x < 1)$ и $(x > 2) \}$ означает пустое множество C. Обратите внимание, что $P(x)$ представляет собой *предикат*, т.е. логическое выражение, которое может быть истинным или ложным в зависимости от конкретного значения элемента x. Элемент x принадлежит множеству M только в том случае, если $P(x)$ истинно.

Универсальный способ задания множества состоит в обычном перечислении его элементов. Например, запись $A = \{x_1, x_2, \ldots, x_n\}$ означает, что конечное множество A состоит из элементов $x_1, x_2, \ldots, x_n$. Множество $M = \{2, 3, 4, 5\}$ представляет собой множество экзаменационных оценок. Порядок перечисления элементов множества не имеет значения. Поэтому то же самое множество экзаменационных оценок можно записать в виде $M = \{5, 4, 3, 2\}$ и еще массой других способов.

Чтобы над множествами можно было выполнять те или иные операции, нужно предварительно определить способы их сравнивания между собой:

- некоторые множества A и B могут совпадать ($A = B$) или не совпадать ($A \neq B$) друг с другом; множества считаются совпадающими, если любой элемент множества A принадлежит множеству B, и наоборот, любой элемент множества B принадлежит и множеству A;
- если любой элемент множества A принадлежит множеству B, то говорят, что множество A является подмножеством (частью) множества B и записывают этот факт в виде $A \subset B$; если множества A и B совпадают ($A = B$), то они являются подмножествами друг друга, т.е. $A \subset B$ и $B \subset A$;
- если хотя бы один элемент множества A не принадлежит множеству B, то это можно записать в виде $A \not\subset B$; иначе говоря, множество A не является подмножеством множества B;
- пустое множество является подмножеством любого множества M, т.е. $\varnothing \subset M$; любое множество M является подмножеством самого себя, т.е. $M \subset M$.

В ряде случаев возникает необходимость сравнивать между собой не только сами множества, но и элементы одного и того же множества. Для этого элементам данного множества с помощью отношения “<” приписывают свойство так называемой *линейной упорядоченности*. Благодаря этому, из двух произвольных элементов множества один всегда “меньше” другого, т.е. “предшествует” ему. Проще всего обнаружить линейную упорядоченность множеств, элементами которых есть числа. Операции над множествами позволяют создавать из одних множеств другие. Таким образом, мы получаем еще один способ задания множества (кроме указания общего свойства либо перечисления его элементов). Известны три основные операции над множествами.

- Суммой или объединением двух данных множеств A и B называется множество $C = A \cup B$, каждый элемент которого принадлежит хотя бы одному из данных.
- Разностью двух данных множеств A и B называется множество $C = A \setminus B$, которое содержит все элементы множества A, за исключением содержащихся в множестве B.

- Пересечением двух данных множеств A и B называется множество $C = A \cap B$, каждый элемент которого принадлежит обоим данным множествам.

Приведем пример. Пусть $A = \{ x \mid x$ – *делитель числа* $15\}$, $B = \{ x \mid x$ – *простое число, меньшее* $10\}$. Иными словами говоря, $A = \{ 1, 3, 5, 15 \}$, $B = \{ 2, 3, 5, 7 \}$. Тогда: $C_1 = A \cup B = \{ 1, 2, 3, 5, 7, 15 \}$, $C_2 = A \setminus B = \{ 1, 15 \}$, $C_3 = A \cap B = \{ 3, 5 \}$.

8.2. Множественный тип и его свойства

Множественный тип в языке Паскаль не является стандартным и вводится с помощью зарезервированного слова `Set` описанием, имеющим такой вид.

```
Type
   T = Set Of T0;
```

Здесь тип `T0` называется базовым, и его роль состоит в указании некоторого достаточно широкого множества исходных элементов, из которых в программе можно будет строить те или иные конкретные множества типа `T`. Иначе говоря, идентификатором `T` именуется вполне конкретный исходный набор элементов типа `T0`, из которых мы сможем создавать самые различные множества. Любое из них будет считаться множеством типа `T`. Набор элементов типа `T0` назовем *базовым множеством*. Таким образом, множества в языке Паскаль только конечные.

Опишем некоторую переменную `X` множественного типа.

```
Var
   X: T;
```

Переменную `X` можно называть множеством `X`. Значениями величины `X` могут быть разные множества типа `T`, т.е. множества, составленные из различных элементов базового типа `T0`. Учитывая математическую сторону понятия множества, можно сказать, что возможные значения величины `X` — это все подмножества базового множества.

Оценим свойства типа `T` с количественной точки зрения:

- если базовый тип `T0` не содержит ни одного элемента, то из такого материала можно построить всего лишь одно пустое множество $\varnothing$;
- если базовый тип `T0` содержит один элемент, например `x1`, то возможно построение двух множеств — $\varnothing$ и $\{x_1\}$;
- если базовый тип `T0` содержит два элемента, например x_1 и x_2, то возможно построение четырех множеств — $\varnothing$, $\{x_1\}$, $\{x_2\}$ и $\{x_1, x_2\}$;
- в случае трех элементов базового типа `T0` возможно существование уже восьми множеств типа `T`, в том числе $\varnothing$, $\{x_1\}$, $\{x_2\}$, $\{x_3\}$, $\{x_1, x_2\}$, $\{x_1, x_3\}$, $\{x_2, x_3\}$ и $\{x_1, x_2, x_3\}$.

Продолжая ряд этих примеров, нетрудно заметить, что общее количество различных множеств, которые можно построить из элементов типа `T0`, определяется формулой $\mathrm{Card}(T) = 2^{\mathrm{Card}(T0)}$, где Card — условная функция, определяющая общее количество значений указанного типа. "Двоичный" характер формулы объясняется тем, что каждый элемент в составе множества также имеет "двоичный" характер: либо он там есть, либо его нет. Если учесть, что количество *n*-разрядных двоичных чисел определяется выражением 2^n, то это убеждает в правильности формулы для количества возможных значений типа `T`.

Возможности построения и использования множеств в языке Паскаль весьма ограниченны. Разумеется, если иметь в виду стандартные средства, а не искусственно создаваемые структуры. В частности, не могут быть произвольными элементы базового множества `z: T0`, что проявляется в виде следующих ограничений на базовые типы:

- базовым типом `T0` может быть только порядковый тип; благодаря этому линейная упорядоченность как элементов базового множества, так и элементов всех множеств типа `T` обеспечивается на основе их порядковых номеров `Ord(z)`;
- количество возможных значений базового типа должно удовлетворять условию $\mathrm{Card}(\texttt{T0}) \le 256$; иначе говоря, мощность базового множества не может превышать 256;
- порядковые номера элементов базового множества должны удовлетворять условию $0 \le \texttt{Ord(z)} \le 255$.

Таким образом, мы приходим к пониманию того, что из всех стандартных типов базовыми для множеств могут быть только типы `Byte`, `Char` и `Boolean`, значения которых помещаются в один байт. Только их свойства изначально удовлетворяют вышеперечисленным ограничениям. Попытка описания `Type T: Set Of Real;` приводит к появлению сообщения об ошибке `Error 29: Ordinal type expected` — ожидается порядковый тип. Невозможно воспользоваться также типом `Type T: Set Of Integer;`, так как это влечет за собой ошибку `Error 23: Set base type out of range`. Действительно, объем типа `SizeOf(Integer) = 2` байта, что выходит за допустимый диапазон в один байт. Поскольку `SizeOf(ShortInt) = 1` байт, то описание `Type T: Set Of ShortInt;` ошибочно по другой причине. Оказывается, для типа `ShortInt` порядковый номер наименьшего значения `Ord(Low(ShortInt)) = –128`, что выходит за допустимые границы для порядковых номеров элементов множества.

Базовыми для множеств могут быть использованы также и нестандартные типы: интервальный и перечисляемый. Естественно, для этого их нужно описать таким образом, чтобы удовлетворялись указанные выше ограничения. В частно-

сти, базовым типом для интервального типа можно выбрать только Byte или Char, а количество значений перечисляемого типа не должно превышать 256.

Примеры одноступенчатого описания множественных типов.

```
Type
   Simvol = Set Of Char; {в базовое множество входят все ASCII-символы}
   Litera = Set Of 'a'.. 'z'; {базовым множеством является отрезок типа
Char}
   Number = Set Of Byte; {в базовое множество входят все числа от 0 до 255}
   Diapazon = Set Of 1..100; {базовым множеством является отрезок типа Byte}
   Name = Set Of (Ivanov, Petrov, Sidorov); {в базовое множество входят все
указанные константы перечисляемого типа}
```

Еще раз напомним о роли базовых типов. Они определяют исходное базовое множество для соответствующего множественного типа. Так, интервальный базовый тип 1..100 множественного типа Diapazon указывает на то, что элементами любых множеств типа Diapazon могут быть только целые числа в пределах от 1 до 100.

Возможно, более удобным вам покажется двухступенчатое описание множественных типов с использованием промежуточных, например:

```
Type
   P1 = 'a'.. 'z';
   Litera = Set Of P1;
   P2 = 1..100
   Diapazon = Set Of P2;
   P3 = (Ivanov, Petrov, Sidorov);
   Name = Set Of P3;
```

В этом случае промежуточные типы P1, P2, P3 можно держать в отдельной варьируемой части описаний программы, чтобы удобнее было при необходимости оперативно вносить изменения в константы интервальных и перечисляемых типов.

При использовании интервального типа существуют синтаксические ограничения. Минимальное значение X и максимальное значение Y, указываемые через горизонтальное двоеточие, должны удовлетворять условию Ord(X) ≤ Ord(Y). Невыполнение этого условия фиксируется на этапе компиляции сообщением Error 28: Lower bound greater than upper bound — нижняя граница больше верхней.

Выше упоминалось о возможности использовать в качестве базового для множеств логический тип Boolean. Отметим, однако, что практического значения эта возможность не имеет.

8.3. Переменные и константы множественных типов

Описав типы множеств, можно описывать *переменные множественных типов*, например:

```
Var
   m1: Simvol;
   m2: Litera;
   m3: Number;
   m4: Diapazon;
   m5: Name;
```

В процессе выполнения программы эти переменные могут приобретать значения конкретных множеств. При этом элементами множества `m1` могут быть любые символы таблицы кодов ASCII в любых сочетаниях, элементами множества `m2` — любые символы из интервала `'a'..'z'`, элементами множества `m3` — любые числа от 0 до 255 и т.д. Любое из этих множеств может быть пустым.

Любая *константа множественного типа* представляет собой конкретное множество. Она имеет вид перечня своих элементов, записанных через запятую в квадратных скобках (математика использует фигурные). Порядок следования элементов в перечне может быть произвольным. Основные варианты тут таковы:

- элементы множества имеют вид отдельных значений, например:

  `['c','a','e'], [76,3,20,105];`

- элементы множества имеют вид отрезков значений, например:

  `[5..12], ['x'..'z', 'a'..'d'];`

- элементы множества имеют вид выражений, например:

  `['s',Chr(109)], [Round(Sin(1)),Abs(-7),2+3].`

Записывая константу, перечисленные варианты можно комбинировать. В частном случае константа не содержит ни одного элемента, т.е. имеет вид `[]`. Такая запись соответствует пустому множеству.

Синтаксис записи констант допускает повторения и наложения элементов. В этом случае "лишние" элементы убираются из состава константы автоматически. Для отрезка значений допустимо указание сначала большего, а затем меньшего значения. Такой элемент считается пустым. Например, константа `[7,5..8,7]` воспринимается просто как `[5..8]`, а константа `[7,8..5,7]` — как `[7]`.

Константы множественных типов могут фигурировать в составе описаний именованных и типизированных констант. Например:

```
Const
   k = 5;
   m6 = ['a'.. 'k', 'x'];              {именованная константа}
   m7: Litera = ['c', 'a', 'e'];       {типизированная константа}
   m8 = [2*3,k];                       {именованная константа}
   m9: Diapazon = [76,k,10..20];       {типизированная константа}
```

Обратите внимание! Вышеприведенный пример показывает, что в качестве элементов множеств можно использовать *именованные константы* простых типов.

Так, множественные константы m8 и m9 используют в качестве элемента именованную константу k. Эта возможность касается исключительно именованных констант. Попытка в данной ситуации включить в множество типизированную константу или переменную приводит к появлению сообщения об ошибке Error 133: Cannot evaluate this expression — не могу вычислить выражение.

Непосредственно в программе переменным и типизированным константам множественных типов могут присваиваться те или иные значения с помощью оператора присваивания. При этом в его правой части должно находиться выражение множественного типа. Однако присваивание Y := X будет синтаксически правильным только в случае совместимости типов переменной Y и выражения X.

В случае множественных типов все зависит от их базовых типов. Вполне естественно, что совместимыми считаются только те множественные типы, у которых совместимы базовые типы. А для этого необходимо выполнение такого условия: базовые типы или совпадают, или один из них является отрезком другого, или они оба являются отрезками одного и того же типа. При совместимости множественных типов команда присваивания будет не только синтаксически правильной, но и будет правильно выполнена, если значение выражения X укладывается в диапазон значений переменной Y.

Отдельно следует остановиться на типе пустой множественной константы. Ее можно использовать не только в виде [], но и оформить либо как именованную, либо как типизированную константу.

```
Const
   p1 = [];                    {именованная константа}
   p2: Set Of Char = [];       {типизированная константа}
   p3: Set Of Byte = [];       {типизированная константа}
```

По типу пустые константы [] и p1 соответствуют любому множественному типу. Типизированным пустым константам это не свойственно. Например, пустая константа p2 соответствует по типу только символьным множественным типам, а p3 — только числовым.

8.4. Выражения множественного типа

Компонентами (операндами) выражений множественного типа могут быть переменные и константы множественных типов, а также множества-функции, о которых речь пойдет ниже. Все операнды выражения должны быть совместимыми по типу.

При построении выражений можно использовать бинарные операции суммы "+", разности "–" и пересечения "*" множеств. Продемонстрируем результаты выполнения этих операций на примерах с участием двух множеств mn1=['a'..'c'] и mn2=['b'..'d']:

- сумма `mn1+mn2` дает результат `['a'..'d']`, т.е. каждый элемент суммы двух данных множеств принадлежит хотя бы одному из данных;
- разность `mn1-mn2` дает результат `['a']`, т.е. разность двух данных множеств содержит все элементы множества `mn1`, за исключением содержащихся в множестве `mn2`;
- пересечение `mn1*mn2` дает результат `['b','c']`, т.е. каждый элемент пересечения двух данных множеств принадлежит обоим данным множествам.

Как и в обычных арифметических выражениях, операции над множествами имеют приоритеты, а последовательностью их выполнения можно управлять с помощью круглых скобок. Наивысший приоритет у операции пересечения. Операции суммы и разности имеют равный приоритет и в составе выражения выполняются в порядке их записи. Рассмотрим примеры.

1) `[2,3]-[2,3]*[2]=[3]`, так как операция пересечения выполняется в первую очередь. При использовании скобок получаем `([2,3]-[2,3])*[2]=[]`.
2) `[2,3]+[2,3]-[2]=[3]`, так как операции выполняются последовательно. При использовании скобок получаем `[2,3]+([2,3]-[2])=[2,3]`.

Соответствующие по типу множества можно сравнивать между собой и таким образом строить отношения. Эти отношения можно обычным образом включать в состав логических выражений. Продемонстрируем результаты сравнения на примерах с участием двух множеств `mn1=['x','y']` и `mn2=['x'..'z']`.

- Отношение `mn1=mn2` представляет собой условие совпадения множеств, когда любой элемент каждого из них одновременно принадлежит и другому множеству; для данного примера значение этого отношения `False`.
- Отношение `mn1<>mn2` представляет собой условие несовпадения множеств, когда они отличаются хотя бы одним элементом; для данного примера значение этого отношения `True`.
- Отношение `mn1<=mn2` истинно, если множество `mn1` является подмножеством множества `mn2`; для данного примера значение этого отношения `True`.
- Отношение `mn1>=mn2` истинно, если множество `mn2` является подмножеством множества `mn1`; для данного примера значение этого отношения `False`.
- Отношение `<`*элемент*`> In <`*множество*`>` истинно, если элемент принадлежит множеству; например, `'x' In mn1 = True`, `'z' In mn1 = False`.

До сих пор все рассмотренные нами свойства и особенности множеств Паскаля имели соответствующий математический аналог. К сожалению, если ограничиться исключительно ними, то построение более или менее сложных алгоритмов обработки множеств невозможно. Это можно объяснить тем, что математи-

ческое понятие множества, по существу, является статическим. В этом понятии отсутствует движение. Иначе говоря, для математического множества не предполагается участие в каком-либо процессе его изменения.

Например, множество $A = \{ x \mid x$ — *простое число, не превышающее* $10\}$ и множество $B = \{ x \mid x$ — *простое число, не превышающее* $12\}$ близки между собой по смыслу, но фактически являются разными, и процедура перехода от одного к другому не предусмотрена. Вместе с тем, математика открывает прекрасные возможности для построения алгоритмов на основе понятия функции. Основополагающим тут является то, что одна и та же функция $f(x)$ может принимать различные значения в зависимости от значений своего аргумента.

Для устранения отмеченной статичности математических множеств в Паскале реализована возможность использования множеств-функций. И сделать это оказалось совсем просто. Достаточно было только разрешить использование в составе констант множественных типов элементов в виде переменных величин соответствующих базовых типов.

Например, множество `mn = [2,4,i]` можно считать функцией `mn = f(i)`. Действительно, при `i = 6` мы получаем множество `mn = [2,4,6]`, при `i = 8` — множество `mn = [2,4,8]` и т.д.

Такое соглашение имеет далеко идущие последствия. Например, множество `mn = [2,4,i]` следует считать неопределенным, если неопределена переменная `i`. Согласитесь, что такая ситуация в корне противоречит математическому понятию множества. Тем не менее мы приобретаем возможность влияния на мощность множества на базе соотношения `mn + [i]`. За счет этого, например, при программировании практически нереализуемое математическое представление о заданности множества удается трансформировать в последовательный процесс его создания.

8.5. Представление множеств в памяти ЭВМ

Множество любого типа представляется в памяти ЭВМ в виде последовательности нулей и единиц в соседних битах. Значение бита является индикатором наличия того или иного элемента в составе множества (1 — элемент есть, 0 — элемента нет). Таким образом, максимальный размер памяти для размещения множества составляет 256 бит, что соответствует 32 байт. Такую память занимают множества с базовым типом `Byte` или `Char`.

Если в качестве базового типа множества использован интервальный тип, то размер памяти может уменьшиться. Но он всегда равен целому числу байт. Например, для `Type T1 = Set Of 1..10;` размер памяти составляет `SizeOf(T1)=2` байта, а для `Type T2 = Set Of 'a'..'d';` — `SizeOf(T2)=1` байт. В то же время, для размещения элементов множества в первом случае достаточно 10 бит, а во втором — 4 бит.

Исследуем вопрос расположения элементов множества в отведенной памяти более тщательно с помощью экспериментальной программы `E_Set` (см. приложение В, листинг В.16). Некоторые результаты этого исследования представлены в виде нижеследующих примеров.

Пример 1. Рассмотрим множество

`Const X: T = [8..10,13,23];` типа `Type T = Set Of 8..23;`.

Для размещения в памяти любых множеств данного типа необходимо `SizeOf(T)=2` байт. При этом распределение битов таково.

Байт	2								1							
Номер	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
Значение	1	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1

Пример 2. Рассмотрим множество

`Const X: T = [77..80,83];` типа `Type T = Set Of 75..85;`.

Для размещения в памяти любых множеств данного типа необходимо `SizeOf(T)=2` байта. При этом распределение битов таково.

Байт	2								1							
Номер	87	86	85	84	83	82	81	80	79	78	77	76	75	74	73	72
Значение	0	0	0	0	1	0	0	1	1	1	1	0	0	0	0	0

Основные выводы таковы.

Байты множества располагаются в порядке возрастания адресов памяти. Расположение элементов множества в пределах отведенной для него памяти строго фиксировано. Нумерация битов, как это принято, ведется справа налево. При этом некоторому элементу множества `x` соответствует бит с порядковым номером `Ord(x)`. Начало нумерации битов в отведенной памяти всегда кратно восьми и выбирается по максимуму, а общее количество байт — по минимуму, с учетом возможности размещения всех элементов базового множества.

Первый пример демонстрирует случай, когда интервальный множественный тип в точности кратен байту. При этом имеется в виду не только количество элементов базового множества (шестнадцать), но и то, что порядковые номера этих элементов как раз вписываются в пределы последовательной нумерации битов в составе байтов (с 8 по 23). Действительно, нумерация битов в первом байте с 0 по 7, во втором — с 8 по 15, в третьем — с 16 по 23 и т.д. Первый байт, а также все последние байты, начиная с четвертого, отбрасываются, поскольку не содержат

элементов базового множества. Эксперименты с программой E_Set показывают, что попытки включить в состав множества X элементы, выходящие за пределы типа (числа от 0 до 7 и от 24 до 255), ни к чему не приводят. С одной стороны, не появляется никаких сообщений об ошибке, а, с другой стороны, на множестве это никак не сказывается.

Второй пример демонстрирует случай, когда интервальный множественный тип не кратен байту ни по количеству элементов базового множества (одиннадцать), ни по их нумерации (с 75 по 85). При этом для заданного базового множества автоматически определяется ближайшее множество, кратное байту, исходя из чего и выделяется память. Таким образом, получается, что для размещения элементов с 75 по 85 достаточно два байта: первый — с нумерацией битов от 72 до 79, второй — с нумерацией от 80 до 87. По этой причине "лишними" оказываются элементы с 72 по 74, а также 86 и 87. Эксперименты с программой E_Set показывают успешность попыток включения в состав множества X "лишних" элементов, несмотря на их выход за пределы типа. Числа, выходящие за пределы выделенной памяти (числа от 0 до 71 и от 88 до 255), в состав множества включить не удается. Во всех случаях никаких сообщений об ошибках не появляется.

8.6. Ввод-вывод элементов множеств

Начнем с того, что значения переменных множественных типов невозможно ни ввести с клавиатуры, ни вывести на экран. Эти операции допустимы только по отношению к отдельным элементам множеств с базовыми числовыми и символьными типами.

Процедура SetRead позволяет вводить с клавиатуры строку элементов из некоторого базового множества T0. При этом образуется подмножество X типа T = Set Of T0. Поскольку текущий элемент представлен как E: T0, то в процессе набора строки пользователь должен самостоятельно контролировать принадлежность элементов базовому множеству. В противном случае будет получено сообщение об ошибке Error 201: Range check error — выход значения переменной за пределы допустимого диапазона.

```
Procedure SetRead(Var X: T);
   Var E: T0;
Begin
   X := [];
   While Not EoLn Do Begin Read(E); X := X+[E] End
End;
```

Процедура формирует подмножество поэлементно оператором X := X+[E], где E — значение очередного элемента, прочитанное с клавиатуры. Естественно, что первоначально подмножество X должно быть задано пустым, т.е. X:=[].

Конец вводимой с клавиатуры строки контролируется логической функцией `Function EoLn [(Var F: Text)]: Boolean`, которая принимает истинное значение при появлении признака конца строки некоторого текстового файла `F: Text`. При использовании этой функции без параметров по умолчанию применяется стандартная файловая переменная `Input: Text`, связанная с клавиатурой. При этом конец строки фиксируется по нажатию клавиши <Enter>. Таким образом, цикл формирования подмножества `X` может быть прочитан следующим образом: "пока не нажата клавиша <Enter>, читать с клавиатуры очередной элемент и включать его в подмножество". Кстати, для ввода пустого множества достаточно просто нажать клавишу <Enter>.

Примером может быть использование процедуры `SetRead` для образования подмножества двузначных целых чисел, т.е. для случая

```
Type
   T0 = 10..99;
   T = Set Of T0;
```

В этом случае на клавиатуре набирается через пробел строка чисел, в конце которой нажимается клавиша <Enter>. Числа не должны выходить за пределы интервала `10..99`.

Достаточно просто при этом преодолеть ограничение Паскаля для длины вводимой строки, которое, как известно, равно 127 знакам. Если вы нажмете клавишу <Enter> после пробела, то ввод строки чисел можно будет продолжать. Если по каким-либо причинам самостоятельный контроль диапазона вводимых чисел нежелателен, то используют директиву компиляции `{$R-}`. При этом можно вводить с клавиатуры произвольные целые числа. Числа, выходящие за пределы интервала, отсеиваются автоматически. Однако придется считаться с возможностью появления в составе множества "лишних" элементов, связанных со спецификой представления его в памяти.

Другим примером может быть использование процедуры `SetRead` для образования подмножества строчных букв латинского алфавита, т.е. для случая

```
Type
   T0 = 'a'..'z';
   T = Set Of T0;
```

В этом случае на клавиатуре набирается строка букв, в конце которой нажимается клавиша <Enter>. Буквы не должны выходить за пределы интервала `'a'..'z'`. Однако, чтобы избавиться от самостоятельного контроля вводимых знаков и в этом случае, можно воспользоваться директивой компиляции `{$R-}`. К сожалению, в рамках рассмотренной процедуры возможен ввод лишь таких строк, длина которых не превышает 127 знаков.

Рассмотренная процедура ввода с клавиатуры имеет, с точки зрения пользователя, существенный недостаток. Он состоит в том, что пользователь должен не только помнить элементы базового множества, но и отыскивать их на клавиатуре. А если их там нет? Ведь общеизвестно, что далеко не все символы таблицы кодов ASCII представлены на клавиатуре.

С помощью процедуры `SetReadLn` можно формировать некоторое подмножество `X` типа `T = Set Of T0` с клавиатуры поэлементно. При этом в режиме диалога пользователю предлагаются на выбор все элементы базового множества `T0`.

```
Procedure SetReadLn(Var X: T);
   Var E: T0; I: Byte;
Begin
   X := [];
   For E := Low(T0) To High(T0) Do Begin
      Write('Вводить элемент с порядковым номером ',Ord(E),'? ');
      ReadLn(I); If I = 1 Then X := X+[E] End;
End;
```

В данной процедуре с помощью оператора арифметического цикла просматриваются все элементы базового множества `T0`, начиная с наименьшего `Low(T0)` и кончая наибольшим его элементом `High(T0)`. В процессе просмотра на экран выводится порядковый номер (код) очередного элемента, а пользователю предлагается ввести с клавиатуры значение индикатора `I = 1` в том случае, если этот элемент должен быть включен в состав подмножества `X`. Если элемент не предполагается включать в состав подмножества, то для соответствующего индикатора может быть введено любое другое значение.

Процедура `SetReadLn` может быть использована для образования подмножества произвольных символов ASCII, т.е. когда, например,

```
Type
   T0 = #20..#35;
   T = Set Of T0;
```

В процессе выполнения процедуры пользователю остается следить за кодами предлагаемых элементов и управлять процессом включения их в подмножество посредством ввода того или иного значения индикатора. Необходимость утруждать себя запоминанием и поиском элементов на клавиатуре отпадает. Приятно также то, что при использовании процедуры снимается ограничение длины вводимой с клавиатуры строки.

Вывод элементов множества на экран можно реализовать с помощью универсальной, но простой процедуры `SetWrite`.

```
Procedure SetWrite(X: T);
   Var E: T0;
Begin
```

```
   For E := Low(T0) To High(T0) Do
      If E In X Then Write(E,' '); WriteLn
End;
```

Процесс вывода базируется на арифметическом цикле просмотра всех элементов базового множества `T0`, начиная с наименьшего `Low(T0)` и кончая наибольшим элементом `High(T0)`. При этом проверяется принадлежность каждого из них данному множеству `X`. В случае положительного исхода проверки соответствующий элемент может быть выведен на экран.

Рассмотренные процедуры и примеры их применения реализованы в экспериментальной программе `E_Set_IO` (см. приложение В, листинг В.17).

8.7. Задачи обработки множеств

Ниже представлены основные задачи обработки множеств, демонстрирующие типичные приемы алгоритмизации.

8.7.1. Подсчет мощности множества

Задача М-1. "Мощность". Задано множество `X`. Требуется подсчитать его мощность, т.е. количество элементов в нем.

```
Function Count_Set(Var X: T): Byte;
   Var E: T0; K: Byte;
Begin
   K := 0;
   For E := Low(T0) To High(T0) Do If E In X Then K := K+1;
   Count_Set := K
End;
```

Рассмотрим свойства и особенности функции `Count_Set`.

- Единственным параметром функции является множество `X` типа `T`, содержащее элементы, количество которых требуется подсчитать. Множественный тип `T` определяется во внешнем блоке как `T = Set Of T0`, где в свою очередь тип `T0` определяет базовое множество, подмножеством которого является `X`. В качестве типа `T0` можно использовать стандартные типы `Byte` или `Char`, а также соответствующие интервальные типы. Множество `X` определено в заголовке функции как `Var`-параметр во избежание излишних затрат памяти.
- Основная идея подсчета количества элементов множества `X` состоит в том, чтобы просмотреть все элементы соответствующего базового множества и для каждого из них выполнить проверку, не принадлежит ли он также и множеству `X`. В случае положительного результата проверки следует увеличивать значение счетчика элементов.

- Промежуточная переменная `E: T0` представляет собой текущее значение элемента из базового множества, определяемого типом `T0`. Промежуточная переменная `K: Byte` — счетчик количества элементов. Очевидно, что тип `Byte` является вполне достаточным для этой переменной, поскольку мощность базового множества не может превышать 256.
- Значение переменной `E` изменяется с помощью оператора арифметического цикла, начиная с наименьшего `Low(T0)` и кончая наибольшим элементом `High(T0)` базового множества. Принадлежность значения `E` данному множеству `X` проверяется с помощью логического выражения `E In X` в операторе ветвления.
- Значением функции `Count_Set` является количество элементов в составе множества `X`, что и обеспечивается оператором `Count_Set := K`. Отметим, что "лишние" элементы, наличие которых обусловлено способом представления множества в памяти, данной функцией обнаружены не будут.

В табл. 8.1 работа алгоритма функции `Count_Set` прослеживается более подробно.

Таблица 8.1. **Таблица выполнения алгоритма функции Count_Set**

Дано: `T0=2..4; Low(T0)=2; High(T0)=4; X=[3, 4]`. *Результат:* `K=2`.

K	E	E ≤ 4	E In X
0	2	True	False
	3	True	True
1	4	True	True
2	5	False	

8.7.2. Построение дополнения множества

Задача М-2. "Дополнение". Задано множество `X`. Требуется построить множество `Y`, которое было бы дополнением данного множества до базового.

```
Procedure Supplement_Set(Var X,Y: T);
   Var E: T0;
Begin
   Y := [];
   For E := Low(T0) To High(T0) Do Y := Y+[E];
   Y := Y-X
End;
```

Рассмотрим свойства и особенности процедуры `Supplement_Set`.

- Параметрами процедуры являются исходное множество `X`, содержащее некоторые элементы, а также множество `Y`, состав элементов которого требуется определить. Оба множества имеют тип `T`, который определяется так же, как и в предыдущей задаче.
- По определению, дополнением `Y` является множество, содержащее все те элементы базового множества, которые не принадлежат данному множеству `X`. Возможны, по крайней мере, два способа построения дополнения. Первый способ: нужно просмотреть все элементы базового множества, проверить, какие из них принадлежат данному, и включить в дополнение только те из них, результат проверки для которых оказался отрицательным.
- Второй способ построения дополнения не использует проверку элементов базового множества. Вместо этого можно сначала построить множество, которое содержит все элементы базового, а затем найти разность между ним и данным множеством. Именно этот способ и реализован в вышеприведенной процедуре.
- Промежуточная переменная `E: T0` представляет собой текущее значение элемента из базового множества, определяемого типом `T0`. Значение переменной `E` изменяется с помощью оператора арифметического цикла, начиная с наименьшего `Low(T0)` и кончая наибольшим элементом `High(T0)` базового множества. Благодаря этому множество `Y` постепенно заполняется от пустого состояния до состояния, совпадающего с базовым. По завершении цикла остается только вычислить разность.

8.7.3. Вычисление элементов множеств

Задача М-3. "Вычисление". Нужно построить множество `X`, порядковые номера элементов которого имели бы в своем составе цифру "пять".

Решая подобные задачи, можно строить множества, элементы которых имели бы заданные свойства. Для таких задач характерно то, что включение (либо не включение) некоего элемента в состав множества зависит от результата выполненных вычислений. При этом характер вычислений задается условием задачи. Вычислительные операции могут выполняться как над порядковыми номерами элементов, так и над ними самими в случае построения числовых множеств.

```
Procedure Property_Set(Var X: T);
   Var j: Byte; s: String;
Begin
   X := [];
   For j := Ord(Low(T0)) To Ord(High(T0)) Do Begin
      Str(j,s); If Pos('5',s)<>0 Then X := X+[T0(j)] End
End;
```

Рассмотрим свойства и особенности процедуры `Property_Set`.

- Единственным параметром процедуры является множество X, состав элементов которого требуется определить. Тип множества T определяется во внешнем блоке как T = Set Of T0, где тип T0 определяет базовое множество. Это значит, что мы должны построить X как подмножество базового множества, причем порядковые номера элементов подмножества должны обладать качествами, указанными в условии задачи.
- Основой процедуры является арифметический цикл с параметром j, который представляет собой текущий порядковый номер элемента множества. Вполне естественно, что j изменяется в пределах от наименьшего значения порядкового номера элемента базового множества Ord(Low(T0)) до наибольшего Ord(High(T0)).
- В процессе выполнения цикла очередное значение текущего порядкового номера преобразуется в строку, после чего выполняется проверка наличия в составе этой строки цифры "пять". В случае положительного результата проверки элемент базового множества T0(j) с порядковым номером j присовокупляется ко множеству X.

8.7.4. Построение подмножества кратных

Задача М-4. "Кратное". Задано множество X. Требуется построить его подмножество Y, порядковые номера элементов которого были бы кратны заданному числу D.

```
Procedure Divisible_Set(D: Byte; Var X,Y: T);
   Var E: T0; M,N,K,Q,I: Byte;
Begin
   M := Ord(Low(T0)); N := D*((M+D-1) Div D);
   M := Ord(High(T0)); K := D*(M Div D);
   Q := (K-N) Div D + 1; Y := [];
   For I := 1 To Q Do
      If T0(N+(I-1)*D) In X Then Y := Y+[T0(N+(I-1)*D)];
End;
```

Рассмотрим свойства и особенности процедуры Divisible_Set.

- Параметрами процедуры являются исходное множество X и результирующее множество Y. Оба множества имеют типа T, который определяется так же, как и в предыдущих задачах. В число параметров входит также число D, для которого вполне достаточным является тип Byte.
- И в этой задаче возможны, по крайней мере, два способа достижения результата. Первый способ состоит в том, что сначала строится множество, содержащее все те элементы базового, которые удовлетворяют условию кратности. Затем остается найти пересечение между построенным и данным множеством.

- Второй способ: нужно просмотреть все те элементы базового множества, которые удовлетворяют условию кратности, и проверить, какие из них принадлежат данному множеству. Во множество `Y` следует включить только те из элементов, результат проверки для которых оказался положительным. Именно этот способ и реализован в вышеприведенной процедуре.
- Порядковый номер `J` некоторого элемента `E: T0` базового множества, определяемого типом `T0`, можно, как известно, получить по формуле `J=Ord(E)`. При этом выражения `N=Ord(Low(T0))` и `K=Ord(High(T0))` дают соответственно наименьший и наибольший порядковые номера элементов в составе базового множества. Обратный переход — от порядкового номера к значению элемента — можно осуществить по формуле `E=T0(J)`, используя идентификатор типа как имя функции преобразования.
- Задача "Кратное" проста только на первый взгляд. Она содержит в себе сложную в разрешении проблему. Дело в том, что, просматривая порядковые номера `J` элементов базового множества типа `T0` в пределах от наименьшего `N` до наибольшего `K`, мы не должны выходить за пределы оных. В противном случае для операторов вида `E:=T0(J)` уже на этапе компиляции мы рискуем получить сообщение об ошибке `Error 76: Constant out of range` — выход значения константы за допустимые пределы. Это условие не оставляет нам выбора, т.е. из всех операторов цикла мы вправе использовать только арифметический. В свою очередь использование арифметического цикла предопределяет необходимость предварительного вычисления начального и конечного значений параметра цикла, а также количества повторений.
- Получим наименьший порядковый номер элемента базового множества, кратный заданному `D`. Для этого нужно решить такую задачу: для последовательности целых чисел, начинающейся с `M≥0`, найти наименьшее `N≥M`, кратное `D`. Нужная формула получается в результате размышлений, логических выводов и перебора вариантов: `N = D * ((M + D – 1) Div D)`. Результаты тестирования этой формулы при `D = 4` подтверждают ее правильность.

M	0	1	2	3	4	5	6	7	8	9	10	11
N	0	4	4	4	4	8	8	8	8	12	12	12

- Теперь получим наибольший порядковый номер элемента базового множества, кратный заданному `D`, посредством решения такой задачи: для последовательности целых чисел, заканчивающейся числом `M≥0`, найти наибольшее `K≤M`, кратное `D`. Аналогично предыдущей получается формула `K = D * (M Div D)`. Результаты тестирования этой формулы при `D = 4` подтверждают и ее правильность.

M	0	1	2	3	4	5	6	7	8	9	10	11
K	0	0	0	0	4	4	4	4	8	8	8	8

- Зная начальное и конечное значения порядковых номеров элементов базового множества, удовлетворяющих условию кратности, нетрудно вычислить и количество таких элементов: `Q = (K-N) Div D + 1`. Используя арифметический цикл `For I:=1 To Q`, мы обеспечиваем просмотр не всех, а только тех элементов базового множества, порядковые номера которых кратны `D`. Естественно, что быстродействие возрастает при этом, ориентировочно, в `D` раз.
- Специального рассмотрения требует случай, когда базовое множество не содержит элементов, порядковые номера которых кратны `D`. Например, базовое множество, определяемое типом `T0=1..4`, не содержит элементов, кратных `D=5`. Используем этот пример для определения условий работы упомянутого выше цикла: `N = 5 *((1 + 5 – 1)Div 5) = 5`, `K = 5 *(4 Div 5) = 0`, `Q = (0 - 5) Div 5 + 1 = 0`. Таким образом арифметический цикл приобретает невыполняемый вид `For I:=1 To 0`, т.е. просмотр элементов базового множества осуществляться не будет.
- Параметр `I` рассматриваемого арифметического цикла представляет собой текущий порядковый номер элемента из числа тех, которые удовлетворяют условию кратности. Зная `I`, несложно вычислить порядковый номер элемента `J=N+(I-1)*D`, соответствующий его порядковому типу в составе базового множества. Таким образом, выражение `T0(N+(I-1)*D)` дает значение текущего элемента базового множества, порядковый номер которого в составе типа удовлетворяет условию кратности. Остается проверить, принадлежит ли этот элемент данному множеству `X`, и в случае положительного результата проверки включить его в результирующее множество `Y`.

8.7.5. Построение ограниченного подмножества

Задача М-5. “Ограничение”. Задано множество `X`. Требуется построить его подмножество `Y`, элементы которого не превышали бы значения некоторого граничного элемента `G`.

```
Procedure Limitation_Set(G: T0; Var X,Y: T);
   Var E: T0;
Begin
   Y := [];
   For E := Low(T0) To G Do Y := Y+[E];
   Y := Y*X
End;
```

Рассмотрим свойства и особенности процедуры `Limitation_Set`.

- Параметрами процедуры являются исходное множество X и результирующее множество Y. Оба множества имеют тип T, который определяется так же, как и в предыдущих задачах. В число параметров входит также граничный элемент G, тип которого, естественно, должен совпадать с типом T0 элементов данного множества.
- Для решения задачи целесообразно избрать следующий подход. Сначала построим подмножество, содержащее все те элементы базового множества, которые не превышают значения заданного граничного элемента G. Затем останется найти пересечение между построенным подмножеством и данным множеством X. При таком подходе удается избежать сравнений элементов базового множества с граничным элементом G, а также проверок условий их принадлежности множеству X.
- Для построения упомянутого подмножества используется арифметический цикл, пробегающий все значения от минимального элемента базового множества до заданного граничного.

8.7.6. Формирование множеств из заданных элементов

Для формирования множеств могут использоваться элементы, не только вводимые с клавиатуры, но и хранящиеся в составе иных структур данных.

Задача М-6а. "Преобразование строки". Задана строка символов X. Требуется преобразовать символы строки во множество Y.

```
Procedure Transformation_Char(Var X: String; Var Y: T);
   Var j: Byte;
Begin
   Y:=[];
   For j:=1 To Length(X) Do Y:=Y+[X[j]];
End;
```

Образуемое множество Y имеет тип `T = Set Of T0`, где базовый тип T0 — тип `Char` либо его отрезок. Тип заданной строки символов `X: String`.

Преобразование символов заданной строки во множество происходит без проблем, если строка не содержит символов, выходящих за пределы типа T0. В противном случае символы либо игнорируются, либо попадают в состав множества в качестве "лишних".

Задача М-6б. "Преобразование массива". Задан одномерный целочисленный массив X, содержащий N чисел. Нужно преобразовать числа массива в множество Y.

```
Procedure Transformation_Byte(N: Byte; Var X: P; Var Y: T);
   Var j: Byte;
Begin
```

```
   Y:=[];
   For j:=1 To N Do Y:=Y+[X[j]];
End;
```

Образуемое множество Y имеет тип T = Set Of T0, где базовый тип T0 — тип Byte либо его отрезок. Заданный одномерный массив X имеет тип P, определяемый как P = Array[0..Rmax] Of T1, где тип элементов массива T1 — тип Byte либо его отрезок. Здесь 0..Rmax — граничные значения индексов элементов массива, Rmax — именованная константа, заданная в разделе Const внешнего блока.

Заданное фактическое количество значений в массиве N должно удовлетворять условиям 0 ≤ N ≤Rmax. Подразумевается, что значения в массиве имеют порядковые номера от 1 до N. Элемент массива X[0] не используется.

Дополнительное ограничение N ≤ 256 связано с ограниченным объемом множества. Как и в случае со строкой, числа массива беспрепятственно преобразуются во множество, если они не выходят за пределы типа T0. В противном случае числа либо игнорируются, либо попадают в состав множества в качестве "лишних".

8.7.7. Построение множеств из элементов с заданными свойствами

В общем виде задачи подобного типа связаны с выборкой элементов из некоторой произвольной совокупности. При этом выбираемые и включаемые в состав множества элементы должны обладать определенными наперед заданными свойствами.

Задача М-7а. "Набор чисел". Одномерный массив X содержит набор N произвольных целых чисел. Нужно построить множество Y, содержащее четные числа из этого массива.

```
Procedure Collection_Number(N: Word; Var X: P; Var Y: T);
   Var j: Word;
Begin
   Y:=[];
   For j:=1 To N Do
      If (X[j]>=0) And (X[j]<=255) And (X[j] Mod 2 = 0) Then
Y:=Y+[X[j]];
End;
```

Поскольку исходный массив содержит произвольные числа, то образуемое множество Y должно иметь тип T = Set Of Byte.

Тип самого массива X можно определить как P = Array[0..Rmax] Of Integer, подчеркивая тем самым, что числа в нем могут быть отрицательными, а также чрезмерно большими для множества. Размеры массива 0..Rmax и фактическое количество значений N в нем условиями задачи не ограничиваются, поскольку больше чем 256 значений во множество не поместится. Именно потому для

величины N выбран тип Word. Кроме того, подразумевается, что значения в массиве имеют порядковые номера от 1 до N, а элемент массива X[0] не используется.

Процесс решения задачи состоит в том, что просматриваются по порядку все N чисел массива. Чтобы получить право попасть в состав множества, каждое число должно быть не только четным, но и не выходить за пределы отрезка 0..255. Первое из упомянутых условий имеет вид (X[j] Mod 2 = 0), второе — (X[j]>=0) And (X[j]<=255).

Обратите внимание, что свойство, в соответствии с которым числа выбираются из массива и включаются в состав множества, имеет, если можно так выразиться, "вычисляемый" характер (в данном случае, это их четность). Разумеется, предложенный подход к решению задачи можно использовать и в случае любых других простых и сложных свойств (например, кратность заданному числу, наличие в составе записи числа заданной цифры, вхождение в состав некоей арифметической прогрессии или какой-либо иной замечательной последовательности и т.п.).

Задача М-7б. "Набор символов". Задана строка произвольного текста X. Необходимо построить множество Y, содержащее гласные буквы русского языка, использованные в этом тексте.

```
Procedure Collection_Symbol(Var S,X: String; Var Y: T);
   Var j: Byte;
Begin
   Y := [];
   For j := 1 To Length(S) Do
      If Pos(S[j],X)<>0 Then Y := Y+[S[j]];
End;
```

Образуемое множество Y имеет тип T = Set Of T0, где в качестве базового типа целесообразно принять T0 = Char. В этом случае предложенная процедура приобретает полезную универсальность, поскольку ее без каких-либо изменений можно будет приспособить не только для гласных букв, но и для любой другой группы символов.

По условию задачи тип заданного произвольного текста X: String. А вот в представлении комплекта гласных букв полной однозначности нет. Это может быть и строка, и множество. Исключительно с позиций удобства записи зададим набор гласных букв в виде именованной константы Const S: String = 'аеёиоуыэюя'.

В отличие от предыдущей задачи, в которой заданное свойство имело "вычисляемый" характер, гласные буквы не связаны какой-либо закономерностью, и поэтому заданное свойство элементов множества Y нам придется обеспечивать посредством сравнения с образцом. В качестве такого образца как раз и будет использоваться строка гласных букв S.

Суть алгоритма процедуры `Collection_Symbol` состоит в последовательном просмотре строки гласных букв и в проверке, используется ли очередная гласная в заданном тексте `X`. Для этого вычисляется уже знакомое нам логическое выражение `Pos(S[j],X)<>0`, истинное при наличии буквы `S[j]` в строке `X`. В случае положительного результата проверки соответствующая буква включается в состав множества.

Обращаем внимание читателя на наличие очень похожего варианта этой же процедуры, в котором просматривается не строка гласных букв, а заданный текст.

```
Procedure Collection_Symbol(Var S,X: String; Var Y: T);
   Var j: Byte;
Begin
   Y := [];
   For j := 1 To Length(X) Do
      If Pos(X[j],S)<>0 Then Y := Y+[X[j]];
End;
```

При этом для каждой буквы выполняется проверка, не является ли она гласной. Иначе говоря, проверяется ее наличие в составе строки гласных букв `Pos(X[j],S)<>0`.

Какой из этих двух вариантов процедуры лучше? Лучшим кажется первый вариант, поскольку в нем выполняется циклический просмотр более короткой строки `S` по сравнению с длиной заданного текста `X`. Однако, не теряем ли мы быстродействие, вычисляя условие `Pos(S[j],X)<>0` для более длинной строки? Иные варианты этой процедуры связаны с другим способом представления совокупности гласных букв. Не улучшится ли процедура, если совокупность гласных букв изначально представить в виде множества? Ответы на поставленные вопросы в данном случае невозможно получить умозрительно. Требуется эксперимент с определением быстродействия процедуры.

Всевозможные варианты процедуры с оценкой быстродействия представлены в экспериментальной программе `E_Speed_Set` (см. приложение В, листинг В.18).

8.7.8. Определение минимального элемента множества

Задача М-8. "Минимальный элемент". Требуется в данном непустом множестве `X` найти минимальный элемент.

Идея построения алгоритма решения задачи проста. По порядку просматриваем элементы базового множества, начиная с наименьшего, и при этом проверяем, не содержится ли очередной элемент в данном множестве. Поскольку данное множество не пусто, то рано или поздно результат проверки оказывается положительным. В этот момент процесс просмотра базового множества можно прекратить, а соответствующий элемент определить как найденный минимальный.

Итак, мы видим, что процесс поиска минимального элемента множества принципиально отличается от поиска минимального элемента одномерного массива. Здесь нет никакой необходимости просматривать все множество полностью. Все это, скорее, похоже на обычную задачу поиска в одномерном массиве.

```
Function Minimum_Set(Var X: T): T0;
   Var E: T0;
Begin
   E := Low(T0);
   While Not (E In X) Do E := Succ(E);
   Minimum_Set := E
End;
```

Единственным параметром функции является непустое множество `X` типа `T`, содержащее элементы, среди которых требуется найти минимальный. Множественный тип `T` определяется во внешнем блоке как `T = Set Of T0`. В качестве типа `T0`, определяющего базовое множество, можно использовать стандартные типы `Byte` или `Char`, или соответствующие интервальные.

Алгоритм поиска минимального элемента состоит в следующем. Первоначально значение текущего элемента `E: T0` принимается равным наименьшему значению из базового множества. Далее в цикле выполняется проверка принадлежности текущего элемента данному множеству. Это будет так, если выражение `Not (E In X)` ложно. При этом выполнение цикла прекращается. В противном случае значение текущего элемента заменяется следующим в составе типа `T0` с помощью выражения `Succ(E)`, и выполнение цикла продолжается. После завершения цикла поиска значению функции присваивается значение текущего элемента, которое оказывается минимальным.

Аналогичный вид имеет функция поиска максимального элемента в данном непустом множестве.

```
Function Maximum_Set(Var X: T): T0;
   Var E: T0;
Begin
   E := High(T0);
   While Not (E In X) Do E := Pred(E);
   Maximum_Set := E
End;
```

Отличия таковы: поиск ведется начиная с наибольшего элемента базового множества, а если текущий его элемент не принадлежит данному множеству, то значение текущего элемента заменяется предшествующим ему.

8.8. Примеры построения программ обработки множеств

Большинство обычных задач обработки множеств может быть решено на основе использования типичных подпрограмм, рассмотренных в предыдущем разделе.

Пример 1. Постройте множество чисел Фибоначчи. Используя операции со множествами, выделите подмножество чисел Фибоначчи, кратных некоторому заданному D. Составьте программу решения этой задачи, в которой предусмотрите вывод указанного подмножества, а также значения его мощности.

Предварительные соображения по поводу решения таковы.

- В последовательности чисел Фибоначчи 1, 1, 2, 3, 5, 8,… каждое число, кроме первых двух, равно сумме двух предыдущих. Для получения этого множества мы составим специальную процедуру, взяв за основу процедуру `Property_Set` (задача М-3 "Вычисление").
- Поскольку условием задачи предусмотрено оперирование множествами, то нам понадобится также множество чисел, кратных заданному D. И для этого мы также используем процедуру `Property_Set`.
- Требуемое условием задачи подмножество получим пересечением двух предыдущих. Для подсчета мощности подмножества используем функцию `Count_Set` (задача М-1 "Мощность").
- Результирующая процедура (листинг 8.1) должна будет объединить все перечисленные этапы. В выполняемой части программы поместим обращение к ней, а также вывод требуемого подмножества и значения его мощности.

Листинг 8.1. Программа решения задачи примера 1

```
{$B+,D+,E+,I+,L+,N+,Q+,R+,X-}
Program Task_01_Set;
Type
   T0 = Byte;
   T = Set Of T0;
Procedure Property_Set_1(Var X: T);
   Var a: Byte; b,c: Word;
Begin
   X := []; a := 1; b := 1;
   Repeat
      If b >= Ord(Low(T0)) Then X := X+[T0(b)];
      c := a+b; a := b; b := c
   Until b > Ord(High(T0))
End;
Procedure Property_Set_2(D: Byte; Var X: T);
   Var j: Byte; s: String;
Begin
```

```
   X := [];
   For j := Ord(Low(T0)) To Ord(High(T0)) Do
      If j Mod D = 0 Then X := X+[T0(j)]
End;
Function Count_Set(Var X: T): Byte;
   Var E: T0; K: Byte;
Begin
   K := 0;
   For E := Low(T0) To High(T0) Do If E In X Then K := K+1;
   Count_Set := K
End;
Procedure Task(D: Byte; Var K: Byte; Var X: T);
   Var X1,X2: T;
Begin
   Property_Set_1(X1);
   Property_Set_2(D,X2);
   X := X1*X2;
   K := Count_Set(X)
End;
Procedure SetWrite(X: T);
   Var E: T0;
Begin
   For E := Low(T0) To High(T0) Do
      If E In X Then Write(E,' '); WriteLn
End;
Var
   X: T; D,K: Byte;
Begin
   Write('D? '); ReadLn(D);
   Task(D,K,X);
   If X = [] Then WriteLn('Результирующее множество пусто') Else Begin
      WriteLn('Элементы множества:'); SetWrite(X);
      WriteLn('Количество элементов = ',K) End
End.
```

Описание программы.

- Нестандартные типы программы. Используется множественный тип `T=Set Of T0`, где `T0` — числовой тип (`Byte` или соответствующий ему интервальный).
- Процедура `Property_Set_1(Var X: T)`. Формирует множество `X` чисел Фибоначчи. Это множество может включать только те из них, значения которых не выходят за пределы значений типа `T0`. В процедуре используется скользящая пара соседних чисел Фибоначчи `a` и `b`. Первоначально они совпадают с первыми двумя числами из последовательности. Вспомогательная переменная `c` хранит текущее значение их суммы. Для образования множества используется циклический процесс. В состав множества всегда включается второе число из пары (это допустимо, поскольку в начальной паре они одинаковы). Контроль включения состоит в том, что число не должно нарушать нижний предел значений типа `T0`. Нижний

предел значений указан в виде `Ord(Low(T0))`, а число включается в форме `T0(b)`. Благодаря этому достигается универсальность процедуры: множество Фибоначчи можно построить не только из чисел, но и из знаков с соответствующими порядковыми номерами. Циклический процесс завершается, как только второе число из пары превысит верхний предел значений типа `T0`. Кстати, именно поэтому для числа `b` выбран тип `Word`, тогда как для числа `a` вполне достаточен тип `Byte`.

- Процедура `Property_Set_2(D: Byte; Var X: T)`. Формирует множество `X` элементов, порядковые номера которых кратны заданному `D`. Более подробные сведения можно найти в описании задачи М-3 "Вычисление".
- Функция `Count_Set(Var X: T): Byte`. Используется для подсчета количества элементов множества `X`. Более подробные сведения содержатся в описании задачи М-1 "Мощность".
- Процедура `Task(D: Byte; Var K: Byte; Var X: T)`. Решает основную задачу. Ее параметрами являются исходные данные (число `D`) и требуемые результаты задачи (подмножество `X` и количество его элементов `K`). Состав операторов процедуры соответствует запланированному ходу решения задачи, изложенному в предварительных соображениях.
- Процедура `SetWrite(X: T)`. Используется для вывода элементов множества и имеет вид, который обсуждался в разделе 8.6.
- Раздел переменных программы. В нем указаны величины, являющиеся параметрами основной процедуры `Task`.
- Выполняемая часть программы. Содержит операторы диалогового ввода исходных данных (число `D`), обращение к процедуре `Task` и вывод результатов. Вывод результатов оформлен с помощью оператора ветвления и зависит от того, пустым или непустым оказалось результирующее множество.

Пример 2. Задана строка произвольного текста. Сформируйте множества знаков, которые были использованы в строке не менее двух раз, всего лишь по одному разу, а также тех, которые не использовались вовсе. Составьте программу решения этой задачи, в которой предусмотрите вывод всех перечисленных множеств (листинг 8.2).

Листинг 8.2. Программа решения задачи примера 2

```
{$B+,D+,E+,I+,L+,N+,Q+,R+,X-}
Program Task_02_Set;
Type
   T0 = Char;
   T = Set Of T0;
Procedure Collection_Symbol(X: String; Var A,B: T);
```

```
   Var j: Byte; z: Char;
Begin
   A := []; B := [];
   For j := 1 To Length(X) Do Begin
      z := X[1]; Delete(X,1,1);
      If Pos(z,X)<>0 Then A := A+[z] Else B := B+[z] End;
   B := B-A
End;
Procedure Supplement_Set(X: T; Var Y: T);
   Var E: T0;
Begin
   Y := [];
   For E := Low(T0) To High(T0) Do Y := Y+[E];
   Y := Y-X
End;
Procedure SetWrite_1(X: T);
   Var E: T0;
Begin
   For E := Low(T0) To High(T0) Do
      If E In X Then Write(E,' '); WriteLn
End;
Procedure SetWrite_2(X: T);
   Var E: T0;
Begin
   For E := Low(T0) To High(T0) Do
      If E In X Then Write(Ord(E),' '); WriteLn
End;
Var
   A,B,C: T; X: String;
Begin
   Write('Строка? '); ReadLn(X);
   Collection_Symbol(X,A,B);
   Supplement_Set(A+B,C);
   WriteLn('Знаки, используемые дважды и чаще:'); SetWrite_1(A);
   WriteLn('Знаки, используемые однажды:'); SetWrite_1(B);
   WriteLn('Порядковые номера неиспользуемых знаков:');
SetWrite_2(C)
End.
```

Описание программы.

- Нестандартные типы программы. Используется множественный тип `T=Set Of T0`, где `T0` — символьный тип (`Char` или соответствующий ему интервальный).
- Процедура `Collection_Symbol(X: String; Var A,B: T)`. Построена на основе результатов решения задачи М-7б "Набор символов". Формирует два множества `A` и `B`, используя знаки строки `X`. Множество `A` содержит знаки, используемые в строке `X` не менее двух раз, множество `B` — знаки, используемые только однажды. Вполне очевидно, что объединение этих множеств содержит все используемые в строке знаки. В процессе образования множеств заданная строка изменяется, поэтому параметр процедуры

X оформлен как параметр-значение, а не как параметр-переменная. В циклическом процессе просмотра строки каждый раз запоминается ее первый знак z, который тут же из строки удаляется. Если в оставшейся части строки знак z встречается еще раз, то он включается во множество A, а в противном случае — во множество B. Таким образом, во множестве B оказываются все используемые знаки строки. Окончательное значение множества B устанавливается посредством вычисления разности между ним и множеством A.

- Процедура `Supplement_Set(X: T; Var Y: T)`. Формирует множество Y элементов, не содержащихся во множестве X. Более подробные сведения о ней можно найти в описании задачи М-2 "Дополнение". Отличие состоит в том, что параметр X оформлен теперь как параметр-значение. Это даст возможность использовать соответствующий фактический параметр в виде выражения. Если известно множество знаков X, использованных в данной строке, то в результате обращения к этой процедуре можно получить множество знаков, не использованных в ней.
- Процедура `SetWrite_1(X: T)`. Имеет вид, который обсуждался в разделе 8.6. Используется для вывода множеств знаков, которые были в данной строке использованы не менее двух раз и всего лишь по одному разу. Для вывода множества знаков, которых в строке нет, применение этой процедуры нецелесообразно (возможно, среди этих знаков есть непечатаемые). Именно поэтому в данном случае используется процедура `SetWrite_2(X: T)`, отличающаяся лишь тем, что вместо самих элементов множеств выводятся их порядковые номера.
- Раздел переменных программы. В нем указаны заданная строка `X: String`, а также результирующие множества `A,B,C: T`.
- Выполняемая часть программы. Содержит операторы диалогового ввода заданной строки X, обращение к процедуре `Collection_Symbol(X,A,B)` для формирования заданных множеств A и B, обращение к процедуре `Supplement_Set(A+B,C)` для формирования множества C неиспользуемых знаков, а также операторы вывода полученных множеств.