А. Ф. Рар

СТАНДАРТ ЯЗЫКА МОДУЛА-2
Пособие для изучения
Электронная версия, с исправлениями и уточнениями

Под редакцией
А. Е. Недори

Новосибирск 2002

Настоящее пособие посвящено описанию

Стандарта языка программирования Модула-2. Язык этот был первоначально разработан Н.Виртом для реализации сложных программных систем и имел своей характерной особенностью концепцию модульности, дающей возможность разбивать программную систему на множество раздельно компилируемых компонент. Стандарт языка был разработан в рамках международной организации ISO и отличается от исходного варианта Модулы-2 наличием более богатых программных средств и большей усложнённостью. Описание Стандарта осуществляется в данном пособии на весьма строгом уровне, но без использования таких формальных методов, которые требовали бы особого изучения. Изложение сопровождается значительным количеством примеров. При описании языка особо выделено то его подмножество, которое используется в одном из работающих трансляторов с языка. Приложение к пособию содержит — наряду с другим справочным материалом — сведения об особенностях реализации языка в системе XDS и о дополнительных возможностях языка, предоставляемых этой системой.

СОДЕРЖАНИЕ

ПРИЛОЖЕНИЯ

ПРЕДИСЛОВИЕ

В 1977 году, после года, проведённого в Xerox Palo Alto Research Center (XPARC), после знакомства с языком Mesa и персональной станцией Alto Никлаус Вирт приступил к разработке нового языка, пригодного для реализации больших и сложных программных систем. Язык этот получил название Модула-2. Принципиальным отличием Модулы-2 от Паскаля явилась концепция модуля, дающая возможность разбивать программную систему на совокупность раздельно компилируемых компонент.

Одновременно с разработкой языка (1977–1979) разрабатывались также компилятор для него и машина Lilith. В середине 1980 года была построена первая партия этих машин. На них работали операционная система MEDOS-2 и компилятор с Модулы-2. Всё программное обеспечение было написано на Модуле-2.

Язык быстро стал популярным, хотя и не таким как Паскаль. Появился ряд коммерческих реализаций, разработанных такими фирмами, как TopSpeed, StonyBrooks, Garden Point и др. В нашей стране первая (насколько нам известно) реализация Модулы-2 была осуществлена в Вычислительном центре СО АН СССР в 1984 году (Дмитрий Кузнецов, ЭВМ Burrough-6700); практически одновременно Леонид Захаров добавил ограниченную реализацию Модулы-

2 к многоязыковой системе БЕТА. С 1985 до 1990 года в том же вычислительном центре группой Кронос в рамках проекта СТАРТ была разработана архитектура Модула-2 машины, было спроектировано несколько вариантов этой машины и разработано необходимое программное обеспечение. Программное обеспечение было написано на Модуле-2. Оно включало в себя компилятор с Модулы-2, многопользовательскую операционную систему и несколько крупных приложений.

К сожалению, все разработчики компиляторов добавляли к языку собственные расширения и проектировали собственные библиотеки. Это привело к тому, что перенос программы с одного компилятора на другой стал трудным делом — даже если оба транслятора работали на одной платформе. Отсутствие стандарта на язык существенно осложняло его использование для реализации серьёзных приложений.

В 1985 году была организована группа по стандартизации языка при Британском комитете по стандартизации. В 1987 году она стала группой при ISO. Стандартизация языка заняла 10 лет и завершилась только в 1996 году. Язык, описанный в стандарте (ISO Модула-2), существенно отличается от языка, разработанного Виртом. В языке появились такие конструкции, как исключительные ситуации, комплексные числа, конструкторы массивов, записей,

множеств, многомерные открытые массивы и многое другое. Стандарт языка фиксирует набор стандартных библиотек.

Ещё до утверждения стандарта появились поддерживающие его реализации языка, такие как XDS (компания XDS, http://www.xds.ru) и GPM (Garden Point). Чуть позже появился компилятор фирмы StonyBrook.

Увы, стандарт появился слишком поздно. В 80-е годы язык был современным, но в 90-е стали модными новые идеи, такие как объектно-ориентированное программирование, и язык выглядит теперь несколько устаревшим. Многие потенциальные пользователи успели переключиться на другие языки программирования, не дождавшись стандарта. Тем не менее, для многих приложений, требующих надёжного языка программирования и строгого контроля типов, язык остаётся одним из лучших.

Сам стандарт описан на формальном языке VDL (Vienna Design Language) и, ввиду сложности этого описания, вряд ли может быть рекомендован широкому кругу читателей. Целью настоящего пособия было описать тот же язык, по возможности строго, на достаточно высоком формальном уровне, но без использования таких методов описания, которые требуют особого изучения.

Пособие знакомит читателя практически со всем языком, задаваемым стандартом, однако выделяет из

него некоторое подмножество —"усечённый вариант" языка.

Все те черты языка, которые не вошли в усечённый вариант, задаются в тех частях книги (разделах, пунктах, абзацах и подабзацах, синтаксических правилах и отдельных строках этих правил), которые помечены вертикальным отчерком.

Конкретный выбор конструкций для усечённого варианта был определён некоторым привходящим обстоятельством. Дело в том, что написание этой книги происходило в рамках проекта по переносу системы программирования XDS на БЦВМ С-32 VAX и именно конструкции, реализованные для этой БЦВМ, включены автором книги в избранный им усечённый вариант, в то время как прочие конструкции (сопрограммы, финализация, исключительные ситуации, защита от прерываний и пр.) оставлены за пределами этого варианта. Таким образом, пользователи упомянутой системы могут при желании изучать нужный им вариант Модулы-2, опуская при чтении данной книги всё, помеченное вертикальным отчерком.

Приложение к пособию содержит — наряду с другим справочным материалом — сведения об особенностях реализации языка в системе XDS и о дополнительных возможностях языка, предоставляемых этой системой.

А.Е.Недоря

А.Ф.Рар

Автор благодарен И.В.Поттосину, А.Д.Хапугину, Л.А.Захарову, Т.В.Кузьминову, Д.В.Лескову, В.А.Цикозе и другим научным сотрудникам Института систем информатики СО РАН за многочисленные полезные консультации и советы.

А.Рар

1. СТРУКТУРА ПРОГРАММЫ

Программу на языке Модула-2 можно рассматривать как совокупность *вложенных* друг в друга *модулей*, а именно:

- некоторого фиктивного (не имеющего представления) *внешнего модуля*;
- вложенного во внешний модуль обязательного и единственного программного модуля (см. 14.3);
- вложенных во внешний модуль необязательных раздельных модулей — модулей определений и модулей реализации (см. 14.2);
- необязательных локальных модулей (см. 14.1), каждый из которых может быть вложен в программный модуль, модуль реализации, другой локальный модуль или процедуру (см. 10).

Некоторые из раздельных модулей могут быть *стандартными*, то есть заданными самим языком или его системой программирования. Остальные раздельные модули, а также программный модуль являются *единицами компиляции*, представляемыми своими *программными текстами*.

Конкретные раздельные модули являются частью некоторой определённой программы только в том случае, если из них происходит *импорт* (см. 14.5) в некоторый другой модуль данной программы.

Исполнение программы состоит в исполнении

программного модуля (см. 14.7), что влечёт за собой исполнение других конструкций языка — модулей, операторов (см. 13), описаний (см. 5, 6, 7, 10, 14.1), процедур (см. 10.2), списков импорта и экспорта (см. 14.7). В ходе исполнения этих конструкций может происходить *обработка* представлений типов (см. 5) и формальных параметров (см. 10.2).

По своей сути исполнение программы сводится к работе с допускаемыми языком *значениями*. (Частным случаем значений являются числа — целые, вещественные и комплексные.) *Внутреннее* (машинное) *представление* значения — это некоторая комбинация битов. Такие объекты и конструкции языка, как константы (см. 6), переменные (см. 7) и выражения (см. 12) могут иметь свои значения (постоянные или текущие): *значения констант*, *значения переменных*, *значения выражений*. (Подробности см. в указанных выше разделах.)

Реализация может применить одну из двух *моделей исполнения программ*: *модель со строгим предшествованием* (declare before use model) или *модель с нестрогим предшествованием* (declare before use in declarations model). Смысл этих моделей будет рассматриваться в 4.1.

(О модели, выбранной реализацией XDS, см. Б.5.а.)

Во время исполнения программы может быть "возбуждена" *исключительная ситуация*. В усечённом варианте языка возбуждение исключительной

ситуации означает обнаружение некоторой ошибки и приводит к прекращению исполнения программы.

(В дальнейшем мы будем о тех или иных чертах языка говорить, что они зависят от реализации или от конкретной реализации. При этом не будет исключаться возможность того, что рассматриваемая черта зависит не только от собственно реализации, но и от операционной системы и, быть может, от машинной конфигурации.)

2. ЭЛЕМЕНТЫ ЯЗЫКА

Язык программирования можно рассматривать как бесконечное множество предложений, удовлетворяющих его синтаксису. В случае Модулы-2 эти предложения являются программными текстами для единиц компиляции программы. Каждое из этих предложений представляет собой конечную последовательность *лексем*. Каждая лексема сконструирована как некоторая последовательность *литер* (см. 3) и является либо идентификатором (простым или квалифицируемым — см. 3.1), либо символом (синтаксическим символом или символом операции — см. 3.2), либо изображением константы (см. 3.3). Для описания синтаксиса используются расширенные формы Бэкуса-Наура (РБНФ). При этом квадратные скобки означают, что заключенная в них последовательность может отсутствовать, а фигурные

скобки означают её вхождение 0, 1 или большее число раз. Нетерминальные синтаксические символы (синтаксические понятия) изображаются в данном пособии русскими словами (или назывными предложениями), выражающими интуитивный смысл этих понятий. Терминальные синтаксические символы изображаются расположенными между кавычками (" и " или же ' и ') последовательностями литер языка. (Как будет видно в 3.3.2, в одном из терминальных символов вместо литеры использован *пробел*.)

3 Лексика

Среди литер будем различать литеры обязательные (которые должны обеспечиваться любой приемлемой реализацией языка), желательные (предназначенные для более удобного изображения символов) и дополнительные (используемые в *примечаниях* и изображениях строк).

К обязательным литерам относятся следующие:

0 1 2 3 4 5 6 7 8 9

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

. , ; : ( ) ! + - * / = < > @ _ " '

К желательным литерам относятся следующие:

[ ] { } | ~ # ^

Дополнительные литеры определяются реализацией.

(О наборе литер, принятом в реализации XDS, см. Б.5.б.)

Для удобства написания программ между лексемами, а также в начале и конце исходного текста могут располагаться *ограничители*, то есть пробелы, *концы строчек* и примечания. Примечание есть последовательность литер и ограничителей, заключенная в скобки (* и *). Примечания могут быть вложены друг в друга. Примечания не влияют на смысл программы.

(О дополнительной форме примечаний, предоставляемой опцией M2EXTENSIONS системы XDS, см. Б.1.1.)

3.1. Идентификаторы

Простой идентификатор есть последовательность букв, цифр и подчерков, начинающаяся с буквы или

подчерка.

(Максимальную длину простого идентификатора, допускаемую реализацией XDS, см. в Б.6.)

Квалифицируемый идентификатор состоит из двух или более простых идентификаторов, разделенных точками. (О квалифицируемых идентификаторах и связи их с модулями см. в 14.4 и 14.5.)

```
идентификатор =
        простой идентификатор|
        квалифицируемый идентификатор;
простой идентификатор =
        (буква|подчерк),{буквенно-цифровое|подчерк};
буква = заглавная буква|строчная буква;
заглавная буква =
        "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|
        "M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|
        "Y"|"Z";
строчная буква =
        "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|
        "m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|
       "y"|"z";
буквенно-цифровое = буква|цифра;
цифра = восьмеричная цифра|"8"|"9";
восьмеричная цифра = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7";
подчерк = "_";
квалифицируемый идентификатор =
        идентификатор модуля,".",
```

простой идентификатор;
идентификатор модуля =
простой идентификатор модуля|
идентификатор модуля,
простой идентификатор модуля;

Примеры:

y lex Modula_2 ASCII RealMath.sqrt

Из числа идентификаторов исключаются *ключевые слова*. Ключевое слово есть такая последовательность заглавных букв, которая определена (см. 3.2) как некоторый синтаксический символ или символ операции.

3.2. Символы

символ = синтаксический символ|символ операции;
синтаксический символ =
"." | "," | ";" | ":" | "(" | ")" | ".." | "=" | ":=" |
левая квадратная скобка|
правая квадратная скобка|
левая фигурная скобка|правая фигурная скобка|
разделитель вариантов|символ разыменования|
синтаксическое ключевое слово;
левая квадратная скобка = "[" | "(!";
правая квадратная скобка = "]" | "!)";
левая фигурная скобка = "{" | "(:";

```
правая фигурная скобка = "}" | ":)";
разделитель вариантов = "|" | "!";
символ разыменования = "^" | "@";
синтаксическое ключевое слово =
        "ARRAY"|"BEGIN"|"BY"|"CASE"|"CONST"|
        "DEFINITION"|"DO"|"ELSE"|"ELSIF"|"END"|
        "EXIT"|"EXPORT"|"FOR"|"FROM"|"IF"|
        "IMPLEMENTATION"|"IMPORT"|"LOOP"|
        "MODULE"|"OF"|"POINTER"|"PROCEDURE"|
        "QUALIFIED"|"RECORD"|"REPEAT"|"RETURN"|
        "SET"|"THEN"|"TO"|"TYPE"|"UNTIL"|"VAR"|
        "WHILE"|"WITH"
        |"EXCEPT"|"FINALLY"|"FORWARD"|
            "PACKEDSET"|"RETRY"

    ;
символ операции =
        символ операции типа умножения|
        символ операции типа сложения|
        символ отношения;
символ отрицания = "~" | "NOT";
символ операции типа умножения =
        "*" | "/" | "DIV" | "MOD" | "REM" |
        символ логического умножения;
символ логического умножения = "&" | "AND";
символ операции типа сложения = знак | "OR";
```

знак = "+" | "-";
символ отношения =
"=" | символ неравенства | "<" |
">" | "<=" | ">=" | "IN";
символ неравенства = "#" | "< >";

В каждой паре терминальных символов для следующих синтаксических понятий:

- левая (правая) квадратная (фигурная) скобка,
- разделитель вариантов,
- символ разыменования,
- символ отрицания,
- символ логического умножения,
- символ неравенства —

второй символ пары должен быть в любой реализации доступен для программиста, а первый символ может быть доступен или недоступен в зависимости от наличия соответствующих (желательных) литер.

3.3. Изображения констант

изображение константы =
изображение числа|
изображение строки|изображение литеры;

Изображения констант являются частным случаем выражений; каждое изображение константы имеет свой тип (см. 5, 6) и своё (постоянное) значение.

3.3.1. Изображения чисел

```
изображение числа =
        изображение целого|
        изображение вещественного;
изображение целого =
       десятичное число|восьмеричное число|
       шестнадцатиричное число;
десятичное число = цифра,{цифра};
восьмеричное число =
      восьмеричная цифра,{восьмеричная цифра},"B";
шестнадцатиричное число =
       цифра,{шестнадцатиричная цифра},"H";
шестнадцатиричная цифра =
       цифра|"A"|"B"|"C"|"D"|"E"|"F";
изображение вещественного =
       целая часть,дробная часть,["E",показатель];
целая часть = цифра,{цифра};
дробная часть = ".",{цифра};
показатель = [знак],целая часть;
```

(Литеру "E" в изображении вещественного следует понимать как "умножить на десять в степени".)

Примеры изображений чисел:

1991 3764B 7BCH 12.3 45.67E-8

Изображение целого (вещественного) числа должно быть таким, чтобы соответствующее этому

изображению число можно было рассматривать (в случае вещественного числа, быть может, после необходимого, определяемого конкретной реализацией, округления) как некоторое значение **V** общего целочисленного (общего вещественного числового) типа (см. соответственно 5.1.1 и 5.2). При соблюдении этого условия значение данного изображения есть **V**.

3.3.2. Изображения строк и изображения литер

Изображение строки есть последовательность литер, заключенная в кавычки, либо числовое изображение литеры. В качестве кавычек могут использоваться как одиночные кавычки (апострофы), так и двойные кавычки, однако открывающей и закрывающей кавычкой должна быть одна и та же литера, не встречающаяся в данной последовательности. Кроме собственно литер в последовательности могут находиться пробелы (но не концы строчек — изображение строки не может переноситься на другую строчку). Числовое изображение литеры есть восьмеричное число, сопровождаемое буквой C.

(Максимальную длину изображения строки, допускаемую реализацией XDS, см. в Б.6.)

изображение строки =
    строка в кавычках|

числовое изображение литеры;
строка в кавычках =
' " ',{литера в строке|" ' "|" "},' " '|
" ' ",{литера в строке|' " '|" "}," ' ";
числовое изображение литеры =
восьмеричная цифра,
{восьмеричная цифра},"C";

Синтаксическое правило для синтаксического понятия литера в строке здесь явно не выписано. Подразумевается, что терминальные синтаксические символы этого правила суть все (заключённые в кавычки) литеры, доступные в данной реализации, за исключением кавычек " и '.

(О дополнении, предоставляемом опцией M2EXTENSIONS системы XDS, см. Б.1.1.)

Изображение литеры лексически совпадает с таким изображением строки, которое является числовым изображением литеры или не содержит в своей последовательности литер ни одного элемента или содержит в этой последовательности единственный элемент. (Одна и та же лексема может в разных контекстах восприниматься как изображение строки или как изображение литеры.)

изображение литеры =
литера в кавычках|

числовое изображение литеры;
литера в кавычках =
' " ',[литера в строке|" ' "|" "},' " '|
" ' ",[литера в строке|' " '|" "}," ' ";

Примеры изображений строк (изображения, приведённые во второй строчке, могут восприниматься также как изображения литер):

"MODULA" "Don't worry"
"M" ' " ' " " 33C

Значением изображения литеры является некоторое, определяемое реализацией, *литерное значение* **L**. Если данное изображение есть числовое изображение литеры, то **L** есть то литерное значение, порядковый номер которого в типе CHAR (см. 5.1, 5.1.2) имеет своим восьмеричным представлением содержащееся в изображении восьмеричное число. Иначе, если данное изображение состоит лишь из пары кавычек, то **L** совпадает с некоторым, определяемым реализацией, литерным значением — ***признаком конца строки***. В остальных случаях **L** есть литерное значение, соответствующее содержащейся в данном изображении литере (или пробелу). Указанное соответствие задаётся реализацией.

(О ***признаке конца строки*** в реализации XDS см. Б.5.в).

Значением изображения строки является последовательность литерных значений, рассматриваемая как строка (см. 5.7.3). В случае, если изображение строки состоит лишь из пары кавычек или если оно содержит в своей последовательности литер единственный элемент, значением которого является ***признак конца строки***, то эта последовательность литерных значений пуста. Во всех остальных случаях каждый **i**-й элемент последовательности литерных значений есть литерное значение, соответствующее **i**-му элементу последовательности литер данного изображения строки.

4. ОПИСАНИЯ И ОБЛАСТИ ВИДИМОСТИ

Простой идентификатор, входящий в некоторую определённую синтаксическую конструкцию, может быть *описан* при исполнении этой конструкции, или — говоря по-другому — описан в этой конструкции. (Такое вхождение простого идентификатора в программу будем называть *определяющим вхождением*.) Простой идентификатор, являющийся непосредственным составляющим (в синтаксическом смысле) модуля определений или модуля реализации (см. 14.2), или же описания типа (см. 5), описания константы (см. 6), списка идентификаторов переменных описания переменных (см. 7), описания процедуры (см. 10), описания локального модуля (см.

14.1), или же списка формальных параметров спецификации формальных параметров (см. 5.6.1), описывается при исполнении соответствующего модуля, соответствующего описания или соответствующей спецификации формальных параметров. Два лексически совпадающих простых идентификатора, описанных в разных местах программы считаются различными (исключения из этого правила см. в 10.1 и 14.2).

Каждый простой идентификатор при своём описании получает *область видимости*, каковой областью является (с учётом сказанного в следующем абзаце):

- в случае модуля (определений или реализации) — блок внешнего модуля (см. 14);
- в случае описания, непосредственно содержащегося в модуле определений, — этот модуль определений;
- в случае иного описания — тот блок модуля (см. 14) или блок процедуры (см. 10), во вступительной части которого непосредственно содержится это описание;
- в случае спецификации формальных параметров — блок процедуры из того полного описания процедуры, в заголовке которого содержится данная спецификация.

Здесь и в дальнейшем предполагается, что из области видимости, заданной как некоторый блок модуля или блок процедуры, исключены блоки модулей, вложенных в этот блок модуля (процедуры),

лей, вложенных в этот блок модуля (процедуры), а также блоки процедур, содержащих конструкции, в которых описаны идентификаторы, лексически совпадающие с данным.

В определённых случаях (см. 14.2, 14.4, 14.5) может иметь место *расширение областей видимости* тех или иных идентификаторов за счёт включения в эти области некоторых новых блоков. По отношению к этим новым блокам сохраняет силу оговорка, сделанная в предыдущем абзаце.

Наряду с идентификаторами, задаваемыми программистом, язык предопределяет некоторый набор *стандартных* (или *проникающих*) *идентификаторов*, которые считаются описанными в некотором неявном описании, "содержащемся" в блоке внешнего модуля. (Список стандартных идентификаторов приведен в В.3.) Идентификатор, лексически совпадающий со стандартным, но имеющий иную область видимости, стандартным не считается. (Как видно, если два лексически совпадающих идентификатора различны, то их области видимости не могут пересекаться.)

Назовём *использующим вхождением* идентификатора любое его вхождение, отличное как от определяющего вхождения, так и от вхождения в список импорта или экспорта (см. 14.1), и будем считать, что любой простой идентификатор при каждом своём использующем вхождении в область

видимости лексически совпадающего с ним простого идентификатора совпадает с последним.

Каждый простой идентификатор в своей области видимости *обозначает* в каждый данный момент исполнения программы некоторый объект (тип, константу, переменную, процедуру, формальный параметр, модуль — см. соответственно 5, 6, 7, 10, 10.2, 14), отличный от других объектов. (Впредь, если только это не приводит к двусмысленности, мы будем говорить "объект **D**" вместо "объект, обозначаемый идентификатором — или обозначением — **D**". Об обозначениях см. в 8.)

Вообще говоря, идентификатор может в разные моменты исполнения программы обозначать разные объекты. Будем считать, что с каждым идентификатором связан некоторый *стек*, в вершине которого в каждый данный момент находится обозначаемый этим идентификатором объект. Если в какой-то момент идентификатор начинает обозначать некоторый объект, то этот объект одновременно *опускается* в стек, а если идентификатор перестаёт обозначать объект, то последний *поднимается* из стека, а идентификатор начинает обозначать тот объект (если он существует), который оказался в вершине стека. (То обстоятельство, что идентификатор обозначает в общем случае не единственный объект, а несколько, организованных по стековому принципу, связан с необходимостью

обеспечить возможность *рекурсивного вызова* процедур в языке Модула-2; пример на использование рекурсивных вызовов см. в 14.8.)

Другие случаи (и несколько другие правила) описания идентификаторов будут даны в пп. 5.1.3 и 14.2. О применении идентификаторов полей будет идти речь в пп. 5.8, 8.2 и 13.5.

4.1. Дополнительные замечания: модели исполнения программ и авансированные позиции

Из сказанного выше видно, что к моменту исполнения части программы, содержащей использующее вхождение идентификатора, которое должно обозначать некоторый объект, объект этот должен уже существовать и обозначаться определяющим вхождением того же идентификатора. Если применяется модель с нестрогим предшествованием, описания, находящиеся во вступительных частях блоков (см. 10), исполняются в таком порядке (если только программа допускает хоть один такой порядок), чтобы указанное условие выполнялось. Если же применяется модель со строгим предшествованием, то рассматриваемые описания исполняются в порядке их встречаемости. Поэтому не всякая программа, правильная с точки зрения первой модели, будет подчиняться условию, указанному в начале этого пункта, и, следовательно, являться

правильной с точки зрения второй модели.

Но дело облегчается тем, что для указанного условия языком предусмотрены некоторые исключения. А именно, будем называть некоторые определённые позиции из числа тех, в которых могут находиться использующие вхождения, *авансируемыми позициями*. Если при исполнении того места программы, где расположена авансируемая позиция, соответствующее использующее вхождение идентификатора "не находит" объекта, обозначаемого определяющим вхождением того же идентификатора, но характер программы таков, что искомый объект будет обязательно создан и обозначен определяющим вхождением, то эта ситуация не рассматривается как ошибка, а использующее вхождение начинает обозначать некоторый условный объект в ожидании того момента, когда этот условный объект будет заменён настоящим.

Авансируемые позиции мы встретим в 5.5 и 5.6.

Кроме того, мы их встретим в 10.1.

5. ТИПЫ

Языком Модула-2 определяются *стандартные типы* (см. 5.1.1, 5.1.2, 5.1.3.1, 5.2, 5.3, 5.4.3, 5.6.2) и *общие типы* (см. 5.1.1, 5.2, 5.3, 5.7.3), а в ходе исполнения программы создаются *новые типы*. Каждый такой *тип* определяет некоторое

(упорядоченное или неупорядоченное) множество значений. Если значение **V** есть элемент множества значений, определяемого типом **T**, то говорят, что **V** имеет тип **T**. Одно и то же значение может иметь разные типы. Каждый из таких, встречающихся в программе, объектов или конструкций языка, как константы (в частном случае изображения констант — см. 6), переменные (см. 7), выражения (см. 12), также имеет некоторый тип. (Общий тип отличается от других типов тем, что его могут иметь только изображения констант и выражения.) О значениях, константах, переменных и выражениях языка, имеющих тип **T**, говорят также, что они суть значения (константы, переменные, выражения) типа **T**.

(Стандартный тип определен также в 5.9.)

Для всех типов, кроме общих, имеются представления типа, являющиеся либо явными представлениями типа, либо идентификаторами типа. Явное представление стандартного типа есть некоторый стандартный идентификатор типа. Явными представлениями новых типов служат некоторые, определяемые ниже, конструкции языка. (Два вхождения в программу явных представлений новых типов, даже если они лексически совпадают, рассматриваются как представления разных типов.)

В языке различаются типы *элементарные* и *структурные*. К элементарным типам относятся

порядковые типы (целочисленные типы, литерный тип, типы-перечисления, типы-отрезки — см. 5.1), вещественные и комплексные числовые типы (см. 5.2 и 5.3), типы-множества (см.5.4.1), типы-указатели (см. 5.5), типы-процедуры (см. 5.6).

К элементарным типам относятся также типы-упакованные-множества (см. 5.4.2) и тип защиты (см. 5.9).

К структурным типам относятся типы-массивы (см. 5.7) и типы-записи (см. 5.8).

```
набор описаний типов = "TYPE",{описание типа,";"};
описание типа =
        идентификатор типа,"=",представление типа;
идентификатор типа = простой идентификатор;
представление типа =
        явное представление типа|идентификатор типа;
явное представление типа =
      явное представление стандартного типа|
      явное представление нового типа ;
явное представление стандартного типа =
      явное представление стандартного порядкового
      типа|
      PROTECTION|
      REAL | LONGREAL | COMPLEX |
       LONGCOMPLEX | BITSET | PROC;
явное представление стандартного порядкового
 типа=
      INTEGER | CARDINAL | CHAR | BOOLEAN;
```

явное представление нового типа =
    явное представление порядкового типа|
    явное представление типа-множества|
    явное представление типа упакованного
        множества|
    явное представление типа-указателя|
    явное представление типа-процедуры|
    явное представление типа-массива|
    явное представление типа-записи;

Исполнение описания типа осуществляется следующим образом:

- представление типа обрабатывается и становится представлением типа **T** — результата этой обработки;
- идентификатор типа начинает обозначать тип **T** и становится представлением этого типа.

(Можно, следовательно, говорить об *идентичности* типов, представлениями которых служат идентификатор типа и представление типа, которые входят в одно и то же описание типа.)

Результат обработки представления типа, являющегося идентификатором типа, есть тип, обозначаемый этим идентификатором. Правила обработки явного представления типа задаются в дальнейших пунктах настоящего раздела.

5.1. Порядковые типы

Каждый *порядковый тип* определяет некоторое конечное упорядоченное множество значений, элементы которого снабжены *порядковыми номерами*, взятыми из некоторого диапазона целых чисел. Считается, что значения порядкового типа находятся в таких же арифметических отношениях "равно", "не равно", "меньше", "больше", "меньше или равно", "больше или равно", как и их порядковые номера.

Если **N** есть порядковый номер некоторого значения определённого порядкового типа, а **n** — целое число, то **n**-ым *преемником* (*предшественником*) этого значения относительно данного порядкового типа есть то значение этого типа (если оно имеется), порядковый номер которого есть **N** + **n** (соответственно **N** – **n**).

Каждому порядковому типу сопоставлен его *объемлющий тип*. Если порядковый тип не есть тип-отрезок (см. 5.1.4), то его объемлющий тип совпадает с ним самим.

представление порядкового типа =
    явное представление порядкового типа|
    идентификатор порядкового типа;
явное представление порядкового типа =
    явное представление типа-перечисления|
    явное представление типа-отрезка ;
идентификатор порядкового типа =

идентификатор типа;

5.1.1. Целочисленные типы

К *целочисленным типам* относятся стандартные порядковые типы CARDINAL и INTEGER, а также *общий целочисленный тип*. Тип CARDINAL определяет множество идущих подряд целых неотрицательных чисел, находящихся в диапазоне от 0 до MAX(CARDINAL). Тип INTEGER определяет множество идущих подряд целых чисел, находящихся в диапазоне от MIN(INTEGER) до MAX(INTEGER). (О стандартных функциях MIN и MAX см. в 15.2.4.) Общий целочисленный тип определяет множество идущих подряд целых чисел, находящихся в диапазоне от MIN(INTEGER) до максимального из чисел MAX(CARDINAL) и MAX(INTEGER). Порядковый номер числа, являющегося значением целочисленного типа, есть само это число.

(Значения вызовов функций MIN и MAX в реализации XDS см. в Б.6.)

(О дополнительных целочисленных типах, вводимых при наличии опции M2ADDTYPES системы XDS, см. Б.2.)

5.1.2. Литерный тип

Литерный тип есть стандартный порядковый тип CHAR. Значениями этого типа являются

определяемые реализацией литерные значения, к числу которых относятся (наряду, быть может, с другими) литерные значения, соответствующие всем обязательным, имеющимся желательным и имеющимся дополнительным литерам, а также пробелу. При этом упорядочение литерных значений должно обеспечить последовательное и непрерывное расположение тех значений изображений литер, которые соответствуют:

- цифрам от 0 до 9;
- заглавным буквам от A до Z;
- строчным буквам от a до z.

Некоторые значения литерного типа конкретная реализация может отнести к числу *управляющих* и *форматирующих литерных значений*.

5.1.3. Типы-перечисления

Тип-перечисление есть порядковый тип, определяющий множество значений, отличных от значений любого другого типа, не являющегося типом-отрезком (см. 5.1.4) самого этого типа-перечисления. Порядковые номера элементов этого множества начинаются с числа 0.

явное представление типа-перечисления =
"(",список идентификаторов,")";
список идентификаторов =

простой идентификатор,{",",
простой идентификатор};

Обработка явного представления типа-перечисления осуществляется следующим образом:

- для каждого очередного простого идентификатора, содержащегося в представлении, исполняется некоторое неявное описание константы (см. 6), у которого "идентификатор константы" есть данный простой идентификатор, а значением "константного выражения" является некоторое новое значение;
- результатом обработки представления становится новый тип-перечисление, определяющий множество указанных выше значений, упорядоченных так же, как соответствующие им простые идентификаторы.

Примеры описаний для типов-перечислений:

а) spectrum = (red, orange, yellow, green, blue,
dark_blue, violet)

б) week = (Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday)

5.1.3.1. Логический тип

Логический тип есть стандартный порядковый тип BOOLEAN. Он определяется языком так, как если бы он был задан (в блоке внешнего модуля) описанием

типа BOOLEAN = (FALSE, TRUE) и при обработке этого описания значением константы, обозначаемой идентификатором FALSE, стало бы значение ***ложь***, а значением константы, обозначаемой идентификатором TRUE, стало бы значение ***истина***.

5.1.4. Типы-отрезки

Тип-отрезок есть порядковый тип, определяющий такое множество значений, которое заполняет некоторый замкнутый интервал упорядоченного множества значений, определяемого другим порядковым типом, называемым *исходным типом* данного типа-отрезка. Интервал этот определяется своими границами — нижней и верхней. Порядковый номер элементов типа-отрезка совпадает с порядковыми номерами тех же элементов в исходном типе. Если тип-отрезок **T1** имеет своим исходным типом тип **T2**, то говорим, что **T1** есть тип-отрезок типа **T2**. При этом типы нижней границы и верхней границы должны быть совместимы по выражениям (см. 5.10.1) с исходным типом. Значение нижней границы не должно превосходить значения верхней границы.

явное представление типа-отрезка =
    [представление исходного типа],
    левая квадратная скобка,
    нижняя граница,"..",верхняя граница,
    правая квадратная скобка;

представление исходного типа =
    идентификатор типа;
нижняя граница = константное выражение;
верхняя граница = константное выражение;

Результат обработки явного представления типа-отрезка есть тип, определяющий множество значений, которое заполняет такой замкнутый интервал упорядоченного множества значений, определяемого исходным типом, что наименьший и наибольший порядковые номера значений этого интервала суть соответственно значения нижней и верхней границ.

При этом исходным типом считается тип, представлением которого служит представление исходного типа, если последнее имеется. Если же представление исходного типа опущено, то должно выполняться одно из следующих условий и тогда соответствующий исходный тип определяется как сказано ниже:

- значение нижней границы и значение верхней границы суть значения одного и того же нечислового типа **T**; исходным типом в этом случае является тип **T**;
- значение нижней границы и значение верхней границы суть целые неотрицательные числа; исходным типом в этом случае является стандартный тип CARDINAL;

- значение нижней границы и значение верхней границы суть целые числа, по крайней мере одно из которых отрицательно; исходным типом в этом случае является стандартный тип INTEGER.

Объемлющим типом типа-отрезка является объемлющий тип соответствующего исходного типа. Таким образом, тип-отрезок определяется в конечном счёте своим объемлющим типом, а также нижней и верхней границами некоторого интервала этого объемлющего типа, причём порядковый номер любого его элемента совпадает в конечном счёте с порядковым номером того же элемента в объемлющем типе. Тип-отрезок будем по отношению к его объемлющему типу называть *вложенным типо*м, а также, для общности, любой порядковый тип, не являющийся типом-отрезком, будем называть вложенным типом по отношению к самому себе.

Примеры описаний для типов-отрезков. (см. пример из п. 5.1.3):

а) red_end_of_the_spectrum = spectrum[red..orange]
б) digits = CARDINAL[0..9]
в) around_zero = INTEGER[-3..3]
г) week_days = week[Monday..Friday]

Во всех этих примерах представление исходного типа может быть опущено.

д) days_of_hard_work = week_days[Thursday..Friday]

Объемлющий тип в последнем примере есть week.

5.1.4.1. Приведённые типы-отрезки

Пусть **T** есть некоторый порядковый тип, число значений которого равно **n**. Тогда тип-отрезок [0..**n** – 1] будем называть *приведённым типом-отрезком* типа **T**.

Пример:

Типы а)—д) из предыдущего примера имеют приведённые типы-отрезки со следующими явными представлениями:

а) [0..1] б) [0..9] в) [0..6] г) [0..4] д) [0..1]

5.2. Вещественные числовые типы

К *вещественным числовым типам* относятся стандартные типы REAL и LONGREAL, а также *общий вещественный числовой тип*. Каждый вещественный числовой тип **R** определяет некоторое конечное множество вещественных чисел, причём множество, определяемое типом REAL, есть подмножество множества определяемого типом LONGREAL, которое в свою очередь есть подмножество множества, определяемого общим вещественным числовым типом. Каждое значение **x** типа **R** должно быть по абсолютной величине не больше чем **L** и не меньше чем **S**, где положительные

числа **L** и **S** задаются реализацией для каждого из рассматриваемых типов. Если тип **R** есть REAL или LONGREAL, то соответствующее значение **L** есть значение вызова функции MAX(**R**) (см. 15.2.4).

(Реализация может некоторым образом расширить множество значений типов REAL и LONGREAL, о чём см. в 17.1.2. Значением встречающейся в указанном пункте константы large является, как правило, значение соответствующего **L**, а значение константы small может быть значением соответствующего **S**.)

(Значение вызова функции MAX в реализации XDS см. в Б.6.)

5.3. Комплексные числовые типы

К *комплексным числовым типам* относятся стандартные типы COMPLEX и LONGCOMPLEX, а также *общий комплексный числовой тип*. Комплексный числовой тип определяет то множество комплексных чисел, действительные и мнимые части которых являются соответственно значениями типа REAL, типа LONGREAL или общего вещественного числового типа.

5.4. Типы, относящиеся к множествам

5.4.1. Типы-множества

Каждому *типу-множеству* **S** соответствует

некоторый порядковый тип **B**, называемый *базовым типом* типа **S**. Количество элементов множества, определяемого типом **B**, не должно превышать некоторого, определяемого конкретной реализацией, числа. Тип **S** определяет множество всех подмножеств множества, определяемого типом **B**.

(Упомянутое выше граничное число для реализации XDS см. в Б.6.)

явное представление типа-множества =
"SET","OF",представление базового типа;
представление базового типа =
представление порядкового типа;

Результат обработки явного представления типа-множества есть тип, определяющий множество всех подмножеств множества, определяемого тем типом, представлением которого служит представление базового типа.

Пример описания для типов-множеств (см. пример из п. 5.1.3):

combination_of_colours = SET OF spectrum

Конкретная реализация может применять различные способы внутреннего отображения значений типа-множества. В частности, значение — подмножество элементов базового типа может представляться

- списком элементов, входящих в это подмножество;
- такой последовательностью битов, перенумерованных порядковыми номерами элементов базового типа, что бит, соответствующий элементу, входящему в данное подмножество, содержит единицу, а соответствующий элементу, не входящему в данное подмножество, содержит ноль.

Если подмножества, содержащие последний способ, представления подмножеств будут считаться "упакованными".

(О способе внутреннего отображения значений типа-множества в реализации XDS см. в Б.5.г.)

5.4.2. Типы-упакованные-множества

Тип-упакованное-множество, его *базовый тип* и результат обработки его явного представления определяются так же, как в предыдущем пункте определялись соответственно тип-множество, его базовый тип и результат обработки его явного представления.

```
явное представление типа-упакованного-
  множества =
    "PACKEDSET","OF",
    представление базового типа;
```

Конкретная реализация может применять лишь такой способ внутреннего отображения значений типа-упакованного-множества, при котором представления подмножеств будут упакованы.

5.4.3. Тип BITSET

Если в данной конкретной реализации представления значений типа-множества упакованы, то стандартный тип BITSET может рассматриваться как один из типов-множеств, а именно как такой тип-множество, у которого базовый тип **B** есть определённый тип-отрезок типа CARDINAL. Интервал, соответствующий этому типу-отрезку, заключается между нулем и целым числом, являющимся значением выражения

BITSPERLOC∗LOCSPERWORD - 1 ,

где BITSPERLOC и LOCSPERWORD импортированы из системного модуля SYSTEM (см. 16.1).

Стандартный тип BITSET может рассматриваться также как один из типов-упакованных-множеств, а именно как тип-упакованное-множество с базовым типом **B**.

Пример:

Пусть значение типа BITSET есть множество, состоящее из целых чисел 0, 5 и 8. Это значение можно представить себе (а по существу и надо представлять себе) как некоторую шкалу двоичных

значений, в которой 0-е, 5-е и 8-е двоичные значения суть единицы, а остальные двоичные значения суть нули.

5.5. Типы-указатели

Каждому *типу-указателю* **T** соответствует некоторый тип **P**, называемый *связанным типом* данного типа-указателя. Каждое значение типа **T** есть либо *указатель*, "ссылающийся" на некоторую переменную типа **P** (то есть логический адрес ячеек, отведённых для этой переменной — см. 7), либо определённое, общее для всех типов-указателей, значение ***псевдоимя***.

```
явное представление типа-указателя =
      "POINTER","TO",
      представление связанного типа;
представление связанного типа =
      представление типа;
```

Результат обработки явного представления типа-указателя есть тип, определяющий множество, элементами которого могут быть либо указатели, ссылающиеся на переменные того типа, представлением которого служит представление связанного типа, либо значение ***псевдоимя***.

Пример описания для типа-указателя (см. пример из 5.1.3):

```
pspectrum = POINTER TO spectrum
```

Если в некотором описании типа представление типа есть явное представление типа-указателя, то любое входящее в это явное представление использующее вхождение идентификатора типа находится в авансируемой позиции.

Пример:

Следующая последовательность описаний является допустимой.

```
TYPE P = POINTER TO POINTER TO R;
       S = POINTER TO S;
VAR p: P; s: S;
TYPE R = POINTER TO P;
```

(О дополнении, предоставляемом опцией M2EXTENSIONS системы XDS, см. Б.1.2.)

5.6. Типы-процедуры

Тип-процедура определяет некоторое множество *процедур* (см. также 10). Как сам тип-процедура, так и все процедуры этого типа одновременно характеризуются:

- некоторым (возможно, пустым) списком характеристик типов формальных параметров;

- возможно, некоторым типом, называемым *типом результата*.

Каждая *характеристика типа формального параметра* есть:

- либо некоторый тип,
- либо некоторые признаки открытого массива.

Кроме того, каждая позиция списка характеристик типов формальных параметров есть либо *позиция формального параметра – переменной*, либо *позиция формального параметра – значения*.

Признаки открытого массива состоят из некоторого целого числа, называемого *размерностью открытого массива*, и некоторого типа, называемого *типом компоненты открытого массива*.

Если тип-процедура и процедуры этого типа характеризуются типом результата, то они соответственно называются *типом-процедурой-функцией* и *процедурами-функциями*. В противном случае они называются *типом-собственно-процедурой* и *собственно процедурами*.

явное представление типа-процедуры =
"PROCEDURE",
"(",[список спецификаций типов формальных параметров],")",
[":",представление типа результата];
список спецификаций типов формальных параметров = ["VAR"],

```
      спецификация типа формального параметра,
      {",",["VAR"],
       спецификация типа формального параметра};
спецификация типа формального параметра =
       идентификатор типа|
       представление открытого массива;
представление открытого массива =
       "ARRAY","OF",{"ARRAY","OF"},
       представление типа компоненты открытого
       массива;


представление типа компоненты открытого массива
  = идентификатор типа;
представление типа результата =
                                  идентификатор типа;
```

Результат обработки явного представления типа-процедуры есть тип-процедура-функция, если это представление содержит представление типа результата, и тип-собственно-процедура в противном случае. Список характеристик типов формальных параметров полученного типа-процедуры состоит из стольких элементов, сколько спецификаций типов формальных параметров содержится в явном представлении типа. Если **k**-я по счёту спецификация типа есть идентификатор типа, то **k**-м элементом списка характеристик является тип, представлением которого служит этот идентификатор типа; иначе,

если **k**-я спецификация есть представление открытого массива с **n** вхождениями ключевого слова "ARRAY" и представлением типа компоненты открытого массива, служащего представлением типа **C**, то **k**-м элементом списка характеристик являются признаки открытого массива — размерность открытого массива **n**, и тип компоненты открытого массива **C**. Кроме того, **k**-я позиция списка характеристик есть позиция формального параметра – переменной, если **k**-я спецификация типа содержит ключевое слово "VAR", и позиция формального параметра – значения в противном случае.

Примеры описаний для типов-процедур (см. пример из п. 5.1.3):

```
next_colour_1 = PROCEDURE (VAR spectrum)
next_colour_2 = PROCEDURE (spectrum): spectrum
pure_proc = PROCEDURE ()
pure_func = PROCEDURE (): INTEGER
init = PROCEDURE (VAR ARRAY OF ARRAY OF
                                  REAL)
```

Если в некотором описании типа представление типа есть явное представление типа-процедуры, то любое входящее в это явное представление использующее вхождение идентификатора типа находится в авансируемой позиции.

Пример:

Следующая последовательность описаний является

допустимой.

```
TYPE
      T = PROCEDURE (U): V;
      W = PROCEDURE
                  (ARRAY OF W,ARRAY OF T): W;
VAR t: T; w: W;
TYPE  U = REAL; V = T;
```

(О предоставляемых опцией M2EXTENSIONS системы XDS дополнениях, связанных со спецификацией типа формальных параметров, см. Б.1.2.)

5.6.1. Заголовки процедур

Заголовки процедур входят в описания процедур (см. 10) и определения процедур (см. 14.2).

```
заголовок процедуры =
      заголовок собственно процедуры|
      заголовок процедуры-функции;
заголовок собственно процедуры =
      "PROCEDURE",идентификатор процедуры,
      ["(",[список спецификаций формальных
            параметров],")"];
заголовок процедуры-функции =
      "PROCEDURE",идентификатор процедуры,
      "(",[список спецификаций формальных
           параметров],")",
```

```
        ":",представление типа результата;
идентификатор процедуры =
                                простой идентификатор;
список спецификаций формальных параметров =
        спецификация формальных параметров,
        {";",спецификация формальных параметров};
спецификация формальных параметров =
        спецификация формальных параметров –
        значений |
        спецификация формальных параметров –
        переменных;
спецификация формальных параметров – значений =
        список формальных параметров,":",
        спецификация типа формального параметра;
спецификация формальных параметров –
    переменных =
        "VAR",список формальных параметров,":",
        спецификация типа формального параметра;
список формальных параметров =
        формальный параметр,{",",формальный
                                        параметр};
формальный параметр = простой идентификатор;
```

Формальный параметр, входящий в заголовок процедуры, называется *формальным параметром – значением* или *формальным параметром – переменной* в зависимости от того, является ли

содержащая его спецификация формальных параметров спецификацией формальных параметров – значений или спецификацией формальных параметров – переменных. Рассматриваемый формальный параметр характеризуется некоторым типом **T**, если спецификация типа формального параметра, входящая в указанную спецификацию формальных параметров, есть идентификатор типа, служащий представлением типа **T**, и характеризуется *признаками открытого массива* — размерностью открытого массива **n** и *типом компоненты открытого массива* **C**, если эта спецификация есть представление открытого массива, содержащее **n** вхождений ключевого слова "ARRAY" и представление типа компоненты открытого массива, служащее представлением типа **C**.

Заголовок процедуры **H** считается *согласованным* с типом-процедурой **T**, если выполняется каждое из следующих условий:

- либо **H** есть заголовок собственно процедуры, а **T** — тип-собственно-процедура, либо **H** есть заголовок процедуры-функции, а **T** — тип-процедура-функция;
- если заголовок **H** есть заголовок процедуры-функции, то представление типа результата этого заголовка служит представлением типа результата, характеризующего тип **T**;

- заголовок процедуры **H** содержит столько же формальных параметров, сколько элементов содержит список характеристик типов формальных параметров типа **T**;
- каждый **k**-й по счёту формальный параметр заголовка процедуры **H** и соответственно **k**-й элемент списка характеристик типов формальных параметров типа **T** характеризуются либо одним и тем же типом, либо одинаковыми признаками открытого массива;
- каждый **k**-й формальный параметр заголовка процедуры **H** есть формальный параметр – переменная (или формальный параметр – значение) тогда и только тогда, когда **k**-я позиция списка характеристик типов формальных параметров типа **T** есть позиция формального параметра – переменной (соответственно, формального параметра-значения).

Два заголовка процедуры, согласованные с одним и тем же типом-процедурой, будем считать *согласованными заголовками процедур*.

Примеры заголовков процедур (см. пример из п. 5.1.3):

```
PROCEDURE what_next_1 (VAR x: spectrum)
PROCEDURE what_next_2 (x: spectrum): spectrum
PROCEDURE proc_without_parms
PROCEDURE func_without_parms (): INTEGER
```

```
PROCEDURE init_matrix
        (VAR matrix: ARRAY OF ARRAY OF REAL)
```

Каждый из этих заголовков согласован с соответствующим (тем же по счёту) типом-процедурой, приведённым в примерах из п. 5.6.

5.6.2. Тип PROC

Стандартный тип PROC есть тип-собственно-процедура с пустым списком характеристик типов формальных параметров.

5.7. Типы-массивы

Каждый *тип-массив* характеризуется двумя типами — *типом индекса* и *типом компоненты*. Тип индекса может быть только порядковым типом. Тип компоненты может быть любым.

Значение типа-массива называется *массивом*.

```
явное представление типа-массива =
        "ARRAY",представление типа индекса,
        "OF",представление типа компоненты;
представление типа индекса =
        представление порядкового типа;
представление типа компоненты =
                                представление типа;
```

Если некоторое явное представление типа-массива начинается с фрагмента

"ARRAY **J1** OF ARRAY **J2**" ,

то фрагмент этот может быть без нарушения смысла заменён на

"ARRAY **J1**, **J2**" .

Пример:

Представления типов

ARRAY [1..1] OF ARRAY [1..2] OF ARRAY [1..3] OF REAL ,
ARRAY [1..1] OF ARRAY [1..2],[1..3] OF REAL ,
ARRAY [1..1],[1..2] OF ARRAY [1..3] OF REAL и
ARRAY [1..1],[1..2],[1..3] OF REAL

равносильны.

Результат обработки явного представления типа-массива есть тип-массив, характеризующийся такими типом индекса и типом компоненты, представлениями которых служат соответственно данные представление типа индекса и представление типа компоненты.

Пусть тип индекса и тип компоненты типа-массива **A** суть соответственно **I** и **C**. Если каждому значению **i** типа **I** сопоставить некоторое значение **c** типа **C**, то набор **M** всех таких сопоставленных значений будет являться по определению массивом типа **A**, а **I** и **C** будут являться соответственно *типом индекса* и

типом компоненты этого массива. При этом каждое такое **c** будет называться в **M** *компонентой массива*, а соответствующее **i** — *индексом* этой компоненты. О массивах с компонентами типа **C** будем говорить также как о массивах значений типа **C**.

Пример:

Значением типа-массива с представлением

ARRAY["A".."Z"] OF BOOLEAN

может быть, например, такой массив, состоящий из 26 компонент, что каждая компонента, которой соответствует гласная буква, есть ***истина***, а компонента, которой соответствует согласная, есть ***ложь***.

5.7.1. Типы – многомерные массивы

Если тип индекса и тип компоненты некоторого типа-массива суть соответственно **I** и **C**, то этот тип-массив можно рассматривать как тип – одномерный массив с типом индекса **I** и типом компоненты **C**. Каждое значение (массив) данного типа-массива может рассматриваться как одномерный массив с компонентами типа **C** и соответствующими им индексами типа **I**.

Если тип индекса некоторого типа-массива есть **J**, а тип компоненты этого типа-массива есть некоторый тип – **n**-мерный массив с типами индексов $\mathbf{I}_1$, ..., $\mathbf{I}_n$ и типом компоненты **C**, то данный тип-массив

можно рассматривать как тип – (**n** + 1)-мерный массив с типами индексов **J**, $\mathbf{I}_1$, ..., $\mathbf{I}_\mathbf{n}$ и типом компоненты **C**. Каждое значение (массив) рассматриваемого типа-массива может рассматриваться как (**n** + 1)-мерный массив с компонентами типа **C** и соответствующими им последовательностями индексов типов **J**, $\mathbf{I}_1$, ..., $\mathbf{I}_\mathbf{n}$.

Таким образом, каждый тип-массив может рассматриваться как *тип – многомерный массив* с некоторой *размерностью* **n**, последовательностью **n** *типов индексов* и некоторым определённым *типом компоненты* (который, в частности, может сам быть типом-массивом), а каждый массив данного типа — как *многомерный массив* с такой же *размерностью*, а также *компонентами* и соответствующими им последовательностями *индексов* таких же типов.

Пример (см. пример из п. 5.1.3):

Рассмотрим тип-массив **T** с представлением

ARRAY week, [0 .. 23] OF BOOLEAN .

Значения этого типа можно рассматривать как массивы массивов значений логического типа, либо как некоторые двумерные массивы, в которых первый и второй индексы принимают соответственно значения от Sunday до Saturday и от 0 до 23. Эти массивы легко проинтерпретировать как недельные расписания рабочих и свободных часов некоторого служащего. Более или менее нормальная рабочая

неделя (хотя и без обеда) может быть представлена некоторым массивом **A**, в котором компонента имеет значение ***истина*** тогда и только тогда, когда первый индекс имеет своим типом тип-отрезок week_days, а второй индекс — тип-отрезок [9 .. 16].

5.7.2. Приведённые последовательности индексных значений

Пусть **T0** и **T1** суть типы — **n**-мерные массивы с типами индексов $\mathbf{I}_1$, ..., $\mathbf{I}_n$ и $\mathbf{J}_1$, ..., $\mathbf{J}_n$ соответственно и одним и тем же типом компоненты. Каждый тип индекса $\mathbf{I}_k$ и тип индекса $\mathbf{J}_k$ ($1 \le k \le n$) определяет некоторое упорядоченное множество значений. Пусть количество значений для каждой такой пары типов будет одинаковым.

Каждое значение $\mathbf{j_{k,m_k}}$ типа $\mathbf{J}_k$ расположено в соответствующем упорядоченном множестве в некоторой $\mathbf{m}_k$-ой позиции. Соотнесём каждому такому значению $\mathbf{j_{k,m_k}}$ то значение $\mathbf{i_{k,m_k}}$ типа $\mathbf{I}_k$, которое расположено в своём упорядоченном множестве также в $\mathbf{m}_k$-ой позиции. Каждой последовательности **Sj** значений $\mathbf{j}_1\mathbf{,m}_1$, ..., $\mathbf{j_{n,m_n}}$ соотнесём далее такую последовательность **Si** значений $\mathbf{i}_1\mathbf{,m}_1$, ..., $\mathbf{i_{n,m_n}}$, элементы которой были ранее соотнесены элементам последовательности **Sj**. Последовательность **Si** будем называть

последовательностью, *приведённой* из последовательности **Sj** относительно типов **T1** и **T0**.

Пример:

Рассмотрим типы-массивы **T0** и **T1** с представлениями соответственно

ARRAY [0..9], [0..9] и
ARRAY [11..20], [101..110].

Ясно, что, например, последовательность индексных значений **i**, **j** ($0 \le \mathbf{i} \le 9$, $0 \le \mathbf{j} \le 9$) является последовательностью, приведённой из последовательности индексных значений **i** + 11, **j** + 101 относительно типов **T1** и **T0**.

5.7.3. Строковые типы и строки

Тип-массив, у которого тип компоненты есть литерный тип, является некоторым *строковым типом*. Значение строкового типа, рассматриваемое как непустая последовательность литерных значений, называется *строковым значением*. Последовательность из **m** ($\mathbf{m} \ge 0$) литерных значений, перенумерованных числами из начального отрезка типа CARDINAL, не содержащую ***признака конца строки***, будем называть *строкой*, а число **m** — её *длиной*. (Из этого видно, что непустая строка является строковым значением.) Две строки одинаковой длины считаются совпадающими, если попарно совпадают занумерованные одним и тем же числом компоненты

обеих строк.

Пусть некоторое строковое значение состоит из **n** компонент, некоторая строка длины **m** (**m** ≤ **n**) состоит из начальных компонент этого строкового значения и либо **m** = **n**, либо компонента, находящаяся в строковом значении на (**m** + 1)-м месте относительно начала, есть ***признак конца строки***. В этом случае данную строку будем считать строкой, *абстрагированной* из данного строкового значения.

К числу строковых типов отнесём также *общие строковые типы*, характеризующиеся разными длинами и имеющие своими значениями строки соответствующей длины. (В частности, тип пустой строки есть общий строковый тип, характеризующийся длиной 0.)

5.8. Типы-записи

Типы-записи, их явные представления и их значения имеют в общем случае весьма сложную структуру, точному определению которой будут посвящены следующие пункты. (К счастью, программисту вряд ли когда приходится использовать этот механизм во всей его общности.)

5.8.1. Определение типа-записи

Тип-запись характеризуется некоторым *списком характеристик*, каждый элемент которого есть либо *характеристика поля*, либо *характеристика*

вариантов. Характеристика поля состоит из некоторого простого идентификатора (идентификатора поля) и некоторого типа, называемого *типом поля*. Характеристика вариантов состоит из *характеристики признака* и некоторого *списка вариантов*. Характеристика признака состоит из (необязательного) идентификатора поля и некоторого порядкового типа, называемого *типом признака*. Элементы списка вариантов (кроме, быть может, последнего) суть *варианты*. Последним элементом этого списка может быть некоторый *остаточный список* характеристик. Каждый вариант состоит в свою очередь из

- списка *меток варианта*, каждая из которых есть либо некоторое значение типа, идентичного с данным типом признака, либо упорядоченная пара таких значений, задающих — в качестве соответственно нижней и верхней границы — некоторый отрезок значений данного типа признака;
- некоторого списка характеристик.

Будем считать, что все элементы списка характеристик из любого варианта некоторой характеристики вариантов и все элементы остаточного списка характеристик той же характеристики вариантов "связаны" с характеристикой признака этой характеристики вариантов. Если некоторое значение типа признака,

содержащегося в некоторой характеристике вариантов, совпадает со значением, являющимся меткой некоторого варианта из той же характеристики вариантов, или находится в отрезке, задаваемом парой значений, являющейся такой меткой, то будем считать, что это значение "выбирает" данный вариант. Никакие два варианта из характеристики не могут быть выбраны одним и тем же значением. Если значение не выбирает ни один из вариантов и в данной характеристике вариантов имеется остаточный список характеристик, то значение выбирает этот остаточный список.

5.8.2. Явное представление типа-записи

```
явное представление типа-записи =
      "RECORD",список представлений полей,"END";
список представлений полей =
      представление полей,{";",представление полей};
представление полей =
      [представление фиксированных полей|
      представление вариантных полей];
представление фиксированных полей =
      идентификатор поля,
      {","идентификатор поля},":",
      представление типа поля;
идентификатор поля = простой идентификатор;
представление типа поля = представление типа;
представление вариантных полей =
```

```
        "CASE",представление признака,"OF",
        список представлений вариантов,"END";



представление признака =
        [идентификатор признака],":",
        представление типа признака;
идентификатор признака = простой идентификатор;
представление типа признака =
        идентификатор порядкового типа;
список представлений вариантов =
        представление варианта,
        {разделитель вариантов,
         представление варианта},
         ["ELSE",остаточный список представлений];
представление варианта =
        [список представлений меток варианта,":",
         список представлений полей];
список представлений меток варианта =
         представление метки варианта,
         {",",представление метки варианта};
представление метки варианта =
        константное выражение |
        константное выражение,
        ["..",константное выражение];
остаточный список представлений =
       список представлений полей;
```

5.8.3. Обработка явного представления типа-записи

Обработку явного представления типа-записи можно рассматривать как чисто формальное преобразование этого представления в результат обработки — тип-запись, заключающееся в следующих действиях (действия совершаются на разных уровнях вложенности в таком порядке, чтобы они были осмысленными на каждом шаге):

- каждое представление типа поля (не являющееся явным представлением типа-записи) заменяется на тип, представлением которого служит данное представление типа;
- каждый идентификатор поля заменяется на характеристику поля, состоящую из этого идентификатора поля и типа, полученного из представления типа поля, непосредственно содержащегося в том представлении фиксированных полей, которое непосредственно содержит данный идентификатор поля;
- каждое представление фиксированных полей заменяется на последовательность характеристик поля, полученных из непосредственно входящих в это представление последовательных идентификаторов поля;
- каждое представление признака, состоящее из возможного идентификатора признака и

идентификатора порядкового типа, заменяется на характеристику признака, состоящую из того же идентификатора признака (если он есть) и порядкового типа, представлением которого служит данный идентификатор порядкового типа;

- каждое представление метки варианта, являющееся константным выражением, заменяется на метку варианта, являющуюся значением этого выражения;
- каждое представление метки варианта, включающее в себя два константных выражения, заменяется на метку варианта, являющуюся упорядоченной парой значений соответственно первого и второго из этих константных выражений;
- каждый список представлений меток варианта заменяется на список меток варианта, полученных из соответствующих представлений меток варианта;
- каждое представление варианта, состоящее из списка представлений меток варианта и списка представлений полей, заменяется на вариант, состоящий из списка меток варианта, полученного из данного списка представлений меток варианта, и списка характеристик, полученного из данного списка представлений полей;

- каждый остаточный список представлений заменяется на остаточный список характеристик, являющийся списком характеристик, полученным из списка представлений полей, составляющего данный остаточный список представлений;
- каждое представление вариантных полей, содержащее в себе представление признака и список представлений вариантов, заменяется на характеристику вариантов, состоящую из характеристики признака, полученной из данного представления признака, и списка вариантов, состоящего из вариантов и возможного остаточного списка характеристик, полученных соответственно из представлений вариантов и возможного остаточного списка представлений, составляющих данный список представлений вариантов;
- каждый список представлений полей заменяется на список характеристик путём последовательной замены каждого очередного представления полей, являющегося представлением вариантных полей или представлением фиксированных полей, соответственно на один или несколько (в том числе, возможно, и на один) элементов списка характеристик; при этом очередной "пустой" (не являющийся ни представлением фиксированных,

ни представлением вариантных полей) элемент списка представлений остаётся без изменений;

- каждое явное представление типа-записи заменяется на тип-запись, характеризуемый списком характеристик, полученным из непосредственно входящего в это представление списка представлений полей.

5.8.4. Определение записи

Значение типа-записи называется *записью*. Пусть **L** есть копия списка характеристик, характеризующего некоторый тип-запись **R**, и пусть эта копия подвергнута "разметке", состоящей в том, что некоторые характеристики полей, вариантов и признаков, а также некоторые элементы списков вариантов становятся "выделенными", а некоторым выделенным характеристикам полей и характеристикам признаков сопоставляются некоторые значения. Разметка эта должна удовлетворять следующим условиям:

- все элементы списка **L** выделены;
- характеристика признака из выделенной характеристики вариантов выделена;
- каждой выделенной характеристике поля сопоставлено некоторое значение типа, идентичного с типом поля из этой характеристики поля;
- каждой выделенной характеристике признака сопоставлено некоторое значение типа,

идентичного с типом признака из этой характеристики признака;

- если характеристике признака из некоторой характеристики вариантов сопоставлено какое-то значение, выбирающее некоторый элемент списка вариантов из той же характеристики вариантов, то этот элемент списка вариантов выделен, а все остальные элементы списка вариантов не выделены;
- если некоторый вариант выделен, то выделены и все элементы списка характеристик из этого варианта;
- если некоторый остаточный список характеристик выделен, то выделены и все элементы этого списка.

Набор **Z** всех сопоставленных при такой разметке значений является по определению записью типа **R**, причём каждое сопоставленное значение является *полем* записи **Z**, а каждое значение, сопоставленное характеристике признака, является также *полем признака* записи **Z**.

5.8.5. Примеры на применение типов-записей

Пример:

Рассмотрим подробно следующее описание типа, содержащее явное представление типа-записи:

```
country =
        RECORD
            population_of_the_country_in_millions: REAL;
            CASE a_unitary_state_or_a_federation: BOOLEAN OF
                    TRUE: a_republic_or_a_monarchy: BOOLEAN
                   |FALSE: number_of_members_of_the_federation:
                                                        CARDINAL
            END
        END
```

Применив к данному явному представлению преобразование из 5.8.3, получим тип-запись, характеризующийся списком характеристик, состоящим из двух элементов. Первый элемент есть характеристика поля с идентификатором поля

population_of_the_country_in_millions

и типом поля REAL. Второй элемент есть характеристика вариантов. Характеристика признака этой характеристики вариантов состоит из идентификатора поля a_unitary_state_or_a_federation и типа признака BOOLEAN. Список вариантов характеристики вариантов содержит два варианта и не содержит остаточного списка. Первый вариант состоит из единственной метки варианта — значения ***истина*** типа BOOLEAN и единственной характеристики поля с идентификатором поля

a_republic_or_a_monarchy

и типом поля BOOLEAN. Второй вариант состоит из единственной метки варианта — значения ***ложь*** типа

BOOLEAN и единственной характеристики поля с идентификатором поля

number_of_members_of_the_federation

и типом поля CARDINAL.

Этот тип может иметь в качестве своих значений записи с такими, например, полями (расположенными в порядке встречаемости соответствующих идентификаторов полей в явном представлении типа — ср. 9.2; каждое поле представлено здесь соответствующим изображением константы или идентификатором константы):

(263.8, FALSE, 50),

(58.1, TRUE, TRUE),

(5.2, TRUE, FALSE) .

Убедимся в справедливости этого утверждения относительно первого из рассматриваемых значений, проведя нужную разметку в соответствии с условиями из 5.8.4:

а) оба элемента списка характеристик (характеристика поля и характеристика вариантов) становятся выделенными;

б) характеристика признака из характеристики вариантов также становится выделенной;

в) характеристике поля сопоставляется вещественное число 263.8;

г) характеристике признака сопоставляется значение ***ложь***;

д) поскольку характеристике признака сопоставлено значение, выбирающее второй вариант из списка вариантов, то второй вариант становится выделенным, а первый вариант — невыделенным;

е) характеристика поля из второго варианта становится выделенной;

ж) этой характеристике поля сопоставляется целое число 50.

Пример 2:

```
federation_member=RECORD
                    capital: ARRAY [1..25] OF CHAR;
                    total_population: REAL
                  END;
```

5.9. Тип защиты

В конкретной реализации может быть определён стандартный *тип защиты* PROTECTION. Реализация определяет этот тип либо как тип-перечисление, число элементов которого больше единицы, либо как тип-множество с базовым типом, определяющим непустое множество. Стандартные идентификаторы INTERRUPTIBLE и UNINTERRUPTIBLE обозначают при этом, соответственно, минимальное и максимальное значения типа защиты.

Иными словами, для любого значения **P** типа PROTECTION должны выполняться отношения (см. 12.7.2 и 12.7.3)

P ≥INTERRUPTIBLE

и

P ≤UNINTERRUPTIBLE .

(О типе защиты в реализации XDS см. Б.5.д.)

5.10. Определения совместимости

В Модуле-2 действует весьма "нелиберальный", "разрешительный" принцип: объекты разных типов могут встречаться в одной и той же конструкции, только если это явно разрешено некоторыми явными "правилами совместимости". Будем различать четыре вида совместимости: совместимость по выражениям, совместимость по присваиванию, совместимость по параметрам и компонентную совместимость.

5.10.1. Совместимость по выражениям

Два типа считаются *совместимыми по выражениям*, если выполняется хотя бы одно из следующих условий:

- они идентичны;
- один из них есть некоторый целочисленный, вещественный числовой или комплексный числовой тип, а другой — соответствующий общий тип;

- один из них есть литерный тип, а другой — общий строковый тип, характеризующийся длиной 0 или 1.

(О дополнениях, предоставляемых опцией M2ADDTYPES системы XDS, см. Б.2.)

5.10.2. Совместимость по присваиванию

Тип **T0** считается *совместимым по присваиванию* с типом **T1**, если выполняется хотя бы одно из следующих условий:

- тип **T0** идентичен типу **T1**, но не является типом открытого массива (см. 10.2);
- тип **T0** есть тип-отрезок типа **T1**;
- тип **T0** есть тип CARDINAL или тип-отрезок типа CARDINAL, а тип **T1** есть тип INTEGER или общий целочисленный тип;
- тип **T0** есть тип INTEGER или тип-отрезок типа INTEGER, а тип **T1** есть тип CARDINAL или общий целочисленный тип;
- тип **T0** есть некоторый вещественный числовой тип, а тип **T1** есть общий вещественный числовой тип;
- тип **T0** есть некоторый комплексный числовой тип, а тип **T1** есть общий комплексный числовой тип;
- тип **T0** есть некоторый строковый тип, а **T1** есть общий строковый тип, характеризующийся

длиной, не превышающей числа компонент в массивах типа **T0**;

- тип **T0** есть литерный тип или тип-отрезок литерного типа, а **T1** есть общий строковый тип, характеризующийся длиной 0 или 1;
- тип **T0** есть адресный тип (см. 16.1), а тип **T1** есть некоторый тип-указатель;
- тип **T0** есть некоторый тип-указатель, а тип **T1** есть адресный тип.

(Как мы увидим в дальнейшем, ни при каком реальном использовании понятия совместимости по присваиванию тип **T1** не может в силу правила из 12 быть типом-отрезком типа **T0**.)

(О дополнениях, предоставляемых опцией M2ADDTYPES системы XDS, см. Б.2.)

5.10.3. Совместимость по параметрам

Характеристика типа формального параметра *совместима по параметрам-значениям* с некоторым типом **T**, если выполняется хотя бы одно из следующих условий:

- характеристикой типа является тип, совместимый по присваиванию с типом **T**;
- характеристика типа (рассматриваемая как тип или как признаки открытого массива) системно совместима (см. 16.1.1) с типом **T**;

- ♦ характеристикой типа являются признаки открытого массива, состоящие из размерности 1 и литерного типа компоненты, а тип **T** есть общий строковый тип;
- ♦ характеристикой типа являются признаки открытого массива, компонентно совместимые (см. 5.10.4) с типом **T**.

Характеристика типа формального параметра *совместима по параметрам-переменным* с некоторым типом **T**, если выполняется хотя бы одно из следующих условий:

- ♦ характеристикой типа является тип, идентичный с типом **T**;
- ♦ характеристикой типа является адресный тип, а тип **T** есть некоторый тип-указатель;
- ♦ характеристика типа системно совместима с типом **T**;
- ♦ характеристикой типа являются признаки открытого массива, компонентно совместимые с типом **T**.

5.10.4. Компонентная совместимость

Обозначим через **F** признаки открытого массива, состоящие из размерности **n** и типа компоненты **C**. Если **n** есть 1, то обозначим через **F1** тип **C**, а в противном случае обозначим через **F1** признаки открытого массива, состоящие из соответственно **n** –

1 и **C**. Пусть далее тип **A** есть тип-массив с типом компоненты **A1**. Будем считать **F** *компонентно совместимым* с типом **A**, если выполняется хотя бы одно из следующих условий:

- **F1** есть тип, идентичный типу **A1**;
- **F1** системно совместим с **A1**;
- **F1** компонентно совместим с **A1**.

6. КОНСТАНТЫ

Константа есть объект, который имеет свой определённый тип и своё (постоянное) значение. Константа представляется либо изображением константы, либо обозначающим её (в соответствующей области видимости) идентификатором константы.

Типами изображения константы, представляемой изображением константы, являются соответственно тип и значение этого изображения.

Типами изображения целого числа и изображения вещественного числа являются соответственно общий целочисленный тип и общий вещественный числовой тип. Типами изображений строк, имеющих своими значениями строки определённой длины, являются общие строковые типы, характеризующиеся той же длиной.

набор описаний констант =

```
"CONST",{описание константы,";"};
описание константы =
    идентификатор константы,"=",
    константное выражение;
идентификатор константы = простой идентификатор;
```

Исполнение описания константы состоит в том, что создаётся (новая) константа, идентификатор константы начинает обозначать её, вычисляется константное выражение (см. 12), результат этого вычисления становится значением, а тип константного выражения — типом созданной константы.

Примеры описания констант:

```
N = 100
limit = 2*N-1
bound = MAX(INTEGER)-N
salutation = "Dear Sir!"
```

Константу целочисленного, вещественного числового, комплексного числового, логического, литерного или строкового типа будем называть соответственно *целочисленной*, *вещественной числовой*, *комплексной числовой*, *логической*, *литерной* или *строковой константой*.

Считается, что в блоке внешнего модуля "содержится" описание константы со (стандартным)

идентификатором константы NIL и "константным выражением", имеющим своим значением ***псевдоимя***.

7. ПЕРЕМЕННЫЕ

Переменная есть объект, который имеет определённый тип, существует в определённый период исполнения программы, для которого в момент его возникновения отводится определённое количество ячеек машинной памяти (см. 16.1), имеющих некоторый *логический адрес*, и который в ходе исполнения программы имеет своё текущее значение (внутреннее представление которого располагается в указанных ячейках).

набор описаний переменных =
"VAR",{описание переменных,";"};
описание переменных =
список идентификаторов переменных,
":",представление типа;
список идентификаторов переменных =
идентификатор переменной,[машинный адрес],
{",",идентификатор переменной,
[машинный адрес]};
идентификатор переменной =
простой идентификатор;
машинный адрес =

левая квадратная скобка,

константное выражение,
правая квадратная скобка;

В результате исполнения описания переменных каждый простой идентификатор, входящий в список идентификаторов переменных, начинает обозначать определённую, создаваемую в этот момент, переменную того типа, представлением которого служит данное представление типа.

Типом константного выражения, входящего в машинный адрес, должен быть адресный тип. Если за идентификатором переменной следует машинный адрес, то логическим адресом ячеек, отводимых для соответствующей переменной, должно быть значение указанного константного выражения.

О переменных, возникающих в ходе исполнения процедуры, см. 10.2. Переменные могут также в ходе исполнения программы создаваться и уничтожаться с помощью стандартных процедур NEW и DISPOSE (см. 15.1).

Если тип некоторой переменной может рассматриваться как тип-массив (тип – многомерный массив с типом компоненты **C** и типами индексов $\mathbf{I}_1$, ..., $\mathbf{I}_n$), то такая переменная называется *переменной-массивом* (*переменной – многомерным массивом* с *размерностью* **n**). Каждой последовательности

значений $\mathbf{i}_1$, ..., $\mathbf{i}_\mathbf{n}$, имеющих соответственно типы $\mathbf{I}_1$, ..., $\mathbf{I}_\mathbf{n}$, сопоставлена некоторая переменная типа **C**, называемая *компонентой* данной переменной с *индексами* $\mathbf{i}_1$, ..., $\mathbf{i}_\mathbf{n}$. Эта компонента в любой момент исполнения программы имеет своим значением значение той компоненты массива (многомерного массива), являющегося значением данной переменной, которая сопоставлена индексам $\mathbf{i}_1$, ..., $\mathbf{i}_\mathbf{n}$.

Если тип некоторой переменной есть тип-запись, характеризующийся некоторым списком характеристик **L**, то такая переменная называется *переменной-записью*. На любом уровне вложенности списка **L** каждой характеристике поля сопоставляется некоторая переменная типа, идентичного типу поля из данной характеристики поля, называемая *полем* переменной-записи. Точно так же каждой характеристике признака сопоставляется некоторая переменная типа, идентичного типу признака из данной характеристики признака, называемая *полем признака* данной переменной-записи. Поле признака переменной-записи есть по определению частный случай поля переменной-записи. Поле переменной-записи, сопоставленное характеристике поля, связанной (в смысле п. 5.8.1) с некоторой характеристикой признака, "связано" с полем признака, сопоставленным данной характеристике признака. Пусть поле **F** переменной-записи сопоставлено некоторой характеристике поля **H**. Если

в некоторый момент исполнения программы значение переменной-записи содержит поле **V**, сопоставленное характеристике **H** (и являющееся определённым значением), то в этот момент значение поля **F** и поле **V** совпадают, а иначе значение поля **F** не определено.

Если тип некоторой переменной является типом-указателем, то такая переменная называется *переменной-указателем*.

Если представление типа, входящее в некоторое описание переменных, есть идентификатор некоторого скрытого типа (см. 14.2), то все переменные, описанные в этом описании, называются *переменными* (данного) *скрытого типа*.

Если переменная не отождествляется (об отождествлении см. 7.1) с другой переменной, то считается, что она с момента своего создания и до момента первого присваивания (см. 7.2) ей какого-либо значения имеет некоторое (не определяемое программой) *начальное значение*.

Пример набора описаний переменных (см. примеры из 5.1.3, 5.4.1, 5.5, 5.6, 5.8.5):

```
VAR k, p, p0, p1: CARDINAL;
      length, width, area: REAL;
      b: BOOLEAN;
      jour_fixe: week;
      octal_digit: [0..7];
      A: ARRAY [1..100] OF INTEGER;
```

```
colour: spectrum;
cc: combination_of_colours;
pointer: pspectrum;
action: PROC;
a_procedure: PROCEDURE (spectrum,
                        VAR  spectrum);
a_function: next_colour_2;
USA, France, Denmark: country;
```

Идентификатор переменной является одним из видов обозначений переменных (см. 8).

7.1. Отождествление переменных

Когда осуществляется *отождествление* переменной **V1**, не являющейся ни переменной-массивом, ни переменной-записью, с переменной **V2**, переменной **V1** присваивается значение переменной **V2**, и в дальнейшем эти две переменные имеют в каждый момент одно и то же значение (так что присваивание одной из них некоторого нового значения есть в то же время и присваивание того же значения другой переменной).

При отождествлении переменной – **n**-мерного массива **V1** с переменной – **n**-мерным массивом **V2** каждая компонента **v1** переменной **V1** с индексами $\mathbf{i}_1$, ..., $\mathbf{i}_n$ отождествляется с такой компонентой **v2** переменной **V2**, что последовательность индексов, соответствующая компоненте **v1**, является

последовательностью, приведённой из последовательности индексов, соответствующей компоненте **v2**, относительно типа переменной **V2** и типа переменной **V1**.

При отождествлении переменной-записи **V1** некоторого типа-записи с переменной-записью **V2** того же типа-записи каждое поле переменной **V1**, сопоставленное некоторой характеристике поля (характеристике признака) этого типа-записи, отождествляется с полем переменной **V2**, сопоставленным той же характеристике поля (характеристике признака).

7.2. Присваивание значений переменным

Переменным в ходе исполнения программы *присваиваются* значения.

Чтобы переменной **V** типа **T0** можно было присвоить значение **P** типа **T1**, тип **T0** должен быть совместим по присваиванию с типом **T1**. (Эта совместимость является условием необходимым, но не всегда достаточным для реального осуществления присваивания при исполнении программы — см. примеры из 13.2.)

Если **T1** не есть структурный тип, то рассматриваемое присваивание состоит в указываемых ниже действиях:

- если **T1** есть общий вещественный числовой тип, то значение **P** округляется (в соответствии с

правилами, задаваемыми реализацией) до значения типа **T0**;

- если **T1** есть общий комплексный числовой тип, а тип **T0** есть тип COMPLEX (LONGCOMPLEX), то как действительная, так и мнимая часть значения **P** округляются до значения типа REAL (LONGREAL);
- **P** начинает в любом случае восприниматься как значение типа **T0** и становится (новым) значением переменной **V**.

Если при этом **T0** есть адресный тип, а **T1** есть некоторый другой тип-указатель или если **T0** есть тип-указатель, отличный от адресного типа, а **T1** есть адресный тип, то значением переменной **V** становится указатель на переменную, имеющую своим типом базовый тип типа **T0** и своим значением некоторое значение, внутреннее представление которого определяется (быть может, частично) внутренним представлением значения выражения **P** (и может, вообще говоря, оказаться недопустимым для значений данного базового типа).

Если **T1** есть тип-массив, то все последовательные компоненты массива, являющегося значением выражения **P**, присваиваются последовательным компонентам переменной **V**. (Если значение некоторой компоненты этого массива не определено, то значение соответствующей компоненты

переменной **V** тоже становится неопределённым.) В том частном случае, когда **P** есть строковая константа, после окончания указанных присваиваний очередной компоненте переменной **V** (если такая компонента имеется) присваивается ***признак конца строки***, а следующие компоненты (если они имеются) становятся неопределёнными.

Если **T1** (в данном случае обязательно совпадающий с **T0**) есть тип-запись, то процесс присваивания состоит в том, что каждое поле записи, являющейся значением выражения **P**, сопоставленное некоторой характеристике поля (характеристике признака) списка характеристик, характеризующего тип **T1** (и имеющее в данный момент определённое значение), присваивается тому полю переменной **V**, которое сопоставлено той же характеристике поля (характеристике признака), а все остальные поля переменной **V** становятся неопределёнными.

Если вне процесса, описанного в предыдущем абзаце, поле признака переменной-записи получает значение, не выбирающее (в смысле п. 5.8.1) тот вариант или тот остаточный список, который выбирало предшествующее значение поля-признака, то все значения полей переменной-записи, связанных (в смысле п. 7) с этим полем признака, становятся (или остаются) неопределёнными. Если при этом новое значение поля признака вообще не выбирает никакого варианта или остаточного списка, то может быть возбуждена исключительная ситуация.

возбуждена исключительная ситуация.

(О дополнениях, предоставляемых опцией M2ADDTYPES системы XDS, см. Б.2.)

8. ОБОЗНАЧЕНИЯ

Обозначение есть частный случай выражения (см. 12). Обозначение *обозначает* некоторую константу, некоторую процедуру или некоторую переменную и, в соответствии с этим, является *обозначением константы*, *обозначением процедуры* или *обозначением переменной*. Типом обозначения является тип обозначаемого им объекта.

обозначение =
    идентификатор |индексированное обозначение|
    обозначение-поле|разыменованное обозначение;

Если обозначение есть идентификатор, обозначающий некоторую константу, процедуру или переменную, то результат вычисления этого обозначения есть, соответственно, данная константа, данная процедура или значение (текущее) данной переменной (и обозначение обозначает ту же константу, процедуру или переменную).

8.1. Индексированные обозначения

индексированное обозначение =
    обозначение-массив, левая квадратная скобка,
    индексное выражение,
    правая квадратная скобка;
обозначение-массив = обозначение;
индексное выражение = выражение;

Под обозначением-массивом понимается такое обозначение, которое имеет своим типом некоторый тип-массив; под индексным выражением понимается такое выражение, с типом которого совместим по присваиванию тип индекса этого типа-массива.

Типом индексированного обозначения является тип компоненты данного типа-массива.

Если обозначение-массив обозначает массив (переменную-массив) **M**, а значение индексного выражения есть некоторое значение **I**, то индексированное обозначение обозначает ту компоненту массива (переменной-массива) **M**, которая сопоставлена индексу **I**, а результатом вычисления индексированного обозначения является данная компонента массива (значение данной компоненты переменной-массива).

Если в индексированном обозначении встречается пара символов "][", то она без изменения смысла может быть заменена на ",". Повторно применяя это правило нужное число раз, можно получить такое

индексированное обозначение, как

D[$\mathbf{I}_1$, $\mathbf{I}_2$, ..., $\mathbf{I}_\mathbf{n}$] ,

где обозначение-массив **D** обозначает некоторый **n**-мерный массив (переменную – **n**-мерный массив), а само индексированное обозначение — некоторую компоненту этого **n**-мерного массива (этой переменной – **n** -мерного массива), о которых можно сказать, что им сопоставлены индексы, являющиеся значениями выражений $\mathbf{I}_1$, $\mathbf{I}_2$, ..., $\mathbf{I}_\mathbf{n}$, а результатом вычисления индексированного обозначения является данная компонента **n**-мерного массива (значение данной компоненты переменной – **n**-мерного массива).

Примеры:

а) Пусть в программе имеются следующие описания типов и переменных:

```
TYPE week = (Sunday, Monday, Tuesday,
                Wednesday, Thursday, Friday,
                  Saturday);
        time_table =
              ARRAY week, [0 .. 23] OF BOOLEAN;
VAR Johnson: time_table;
```

Пусть далее в ходе исполнения программы переменная, обозначаемая идентификатором Johnson, получит в качестве своего значения массив **A** из

примера в 5.7.1. Тогда

переменная Johnson[Tuesday,10] будет иметь значение ***истина***,

переменная Johnson[Saturday,10] будет иметь значение ***ложь***,

переменная Johnson[Tuesday,18] будет также иметь значение ***ложь***.

Эти три переменные могут быть записаны также как

Johnson[Tuesday][10] ,
Johnson[Saturday][10] и
Johnson[Tuesday][18] .

б) TYPE t = ARRAY [1..2],[1..2] OF REAL;
VAR mx: ARRAY [1..2],[1..2] OF t;

В области видимости этих описаний значение переменой mx есть некоторый массив массивов значений типа t, то есть двумерный массив значений этого типа. Но значение типа t есть, в свою очередь, двумерный массив вещественных значений, так что значение переменной mx становится четырёхмерным массивом вещественных значений. Отсюда следует, что значения индексированных обозначений — переменных mx[1], mx[1,2], mx[1,2,2] и mx[1,2,2,1] — суть соответственно трёхмерный массив, двумерный массив, (одномерный) массив вещественных значений и просто вещественное значение.

8.2. Обозначения-поля

обозначение-поле =
обозначение-запись,".",идентификатор поля;
обозначение-запись = обозначение;

Под обозначением-записью понимается такое обозначение, которое имеет своим типом некоторый тип-запись; идентификатором поля может служить любой идентификатор поля, содержащийся на любом уровне вложенности в списке характеристик, характеризующем данный тип-запись. Типом, приданным обозначению-полю, будет тип поля из той характе
истики поля (характеристики признака), которая непосредственно содержит данный идентификатор (переменную-запись) **R**, а идентификатор поля есть некоторый идентификатор **F**, то обозначение-поле обозначает то поле записи (переменной-записи) **R**, которое сопоставлено характеристике поля (или характеристике признака), непосредственно содержащей идентификатор **F**, а результатом вычисления обозначения-поля является данное поле записи (значение данного поля переменной-записи).

Пример (см. примеры из п. 5.8.5 и 7):

В предположении, что переменные, обозначаемые идентификаторами USA, France и Denmark, имеют

соответственно значения, приведённые в Примере 1 из п. 5.8.5,
переменная
USA.number_of_members_of_the_federation
имеет значение 50,
переменная
France.population_of_the_country_in_millions
имеет значение 58.1,
переменная Denmark.a_unitary_state_or_a_federation
имеет значение ***истина***,
переменная Denmark.a_republic_or_a_monarchy
имеет значение ***ложь***.

8.3. Разыменованные обозначения

разыменованное обозначение =
обозначение-указатель,символ разыменования;
обозначение-указатель = обозначение;

Под обозначением-указателем понимается обозначение, имеющее своим типом некоторый тип-указатель, но не являющееся переменной скрытого типа. Типом, приданным разыменованному обозначению, будет связанный тип данного типа-указателя. Если значением обозначения-указателя является указатель, ссылающийся на некоторую переменную, то разыменованное обозначение обозначает эту переменную, а результатом вычисления

разыменованного обозначения является значение этой переменной. Если значение обозначения-указателя есть ***псевдоимя***, то возбуждается исключительная ситуация.

Пример (см. Пример 2 из 5.8.5):

Пусть в программе имеется следующее описание переменных:

```
VAR Russia:
ARRAY [1..89] OF POINTER TO
                                federation_member;
```

При некотором "разумном" придании значений полям соответствующих переменных-записей значение переменной Russia[16]^.capital будет совпадать со значением изображения строки "Kazan", а значение переменной Russia[16]^.total_population — с числом 3.7;

9. КОНСТРУКТОРЫ ЗНАЧЕНИЙ

Конструктор значения есть выражение (см. 12), значением которого может быть либо массив, либо запись, либо множество.

```
конструктор значения =
      конструктор массива | конструктор записи |
      конструктор множества;
```

Из 9.1—9.3 будет видно, что каждый конструктор значения начинается с некоторого идентификатора типа или же такой идентификатор типа подразумевается. Тип конструктора значения есть тип, представлением которого служит данный идентификатор. Этим типом должен быть соответственно некоторый тип-массив, тип-запись или тип-множество. Для того, чтобы значение конструктора было правильно построенным значением, на компоненты структуры, входящие в соответствующий конструктор значения, могут накладываться дополнительные ограничения.

9.1. Конструкторы массивов

```
конструктор массива =
        идентификатор типа,
       сконструированный массив;
сконструированный массив =
       левая фигурная скобка,
       повторяемая компонента структуры,
       {",",повторяемая компонента структуры},
      правая фигурная скобка;
повторяемая компонента структуры =
      компонента структуры,["BY",повторитель];
```

компонента структуры =
выражение |сконструированный массив |
сконструированная запись |
сконструированное множество;
повторитель = константное выражение;

Тип компоненты типа-массива, представлением которого служит данный идентификатор типа, должен должен быть совместим по присваиванию с типом каждого выражения, входящего в сконструированный массив в качестве компоненты структуры. Тип повторителя должен быть целочисленным типом, и значение повторителя не должно быть меньше 0.

Вычисление конструктора массива происходит следующим образом. Вычисляются все компоненты структуры и все повторители. Если повторяемая компонента структуры содержит повторитель, то значение соответствующей компоненты структуры повторяется столько раз, каково значение повторителя. Последовательность всех получившихся значений, сопоставленных последовательным значениям типа индекса данного типа-массива составляет массив, который и является искомым результатом вычисления.

Пример (см. примеры из пп. 5.7.1 и 8.1):

Двумерный массив **A** из примера в 5.7.1 может быть задан конструктором массива

```
time_table{
    {FALSE BY 24},
    {FALSE BY 9, TRUE BY 8, FALSE BY 7} BY 5,
    {FALSE BY 24} }  .
```

(О дополнении, предоставляемом опцией M2EXTENSIONS системы XDS, см. Б.1.2)

9.2. Конструкторы записей

```
конструктор записи =
    идентификатор типа,сконструированная запись;
сконструированная запись =
    левая фигурная скобка,
    [компонента структуры, {",",
    компонента структуры}],
    правая фигурная скобка;
```

Конструктор записи с идентификатором типа **T** построен правильно, если выполняются следующие условия:

- последовательность значений всех компонент структур совпадает с последовательностью значений всех полей некоторой возможной записи **R** типа **T**, расположенных в порядке расположения соответствующих

идентификаторов полей в явном представлении типа **T**;

- ♦ любая компонента структуры, соответствующая полю признака, является константным выражением.

В этом случае результатом вычисления данного конструктора записи является запись **R**.

Пример:

Значения переменных, обозначаемых в примере из п. 8.2 идентификаторами

USA, France и Denmark ,

могут быть заданы соответственно конструкторами записи

{263.8, FALSE, 50}
{58.1, TRUE, TRUE}
{5.2, TRUE, FALSE) .

9.3. Конструкторы множеств

конструктор множества =
[идентификатор типа],
сконструированное множество;
сконструированное множество =
левая фигурная скобка,
[член,{",",член}],правая фигурная скобка;
член = интервал|единичное;

интервал = выражение,"..",выражение;
единичное = выражение;

Если идентификатор типа в конструкторе множества отсутствует, то подразумевается, что идентификатором типа является BITSET.

Типы выражений, составляющих интервал, должны быть совместимы по выражениям, а базовый тип **B** того типа-множества **T**, который определяется идентификатором типа, должен быть совместим по присваиванию с типом каждого из выражений, содержащихся в сконструированном множестве. Если, однако, значение какого-либо из этих выражений не имеет тип **B**, то возбуждается исключительная ситуация.

Вычисление конструктора множества производится следующим образом.

- Каждое единичное вычисляется, и для него строится множество — значение типа **T** с единственным элементом, являющимся результатом этого вычисления.
- Для каждого интервала вычисляются оба входящие в него выражения, так что значениями первого и второго выражения оказываются некоторые значения $\mathbf{l}_1$ и $\mathbf{l}_2$, и для этого интервала строится множество — значение типа **T**,

элементы которого суть все значения **v**, удовлетворяющие неравенствам $\mathbf{l}_1 \le \mathbf{v} \le \mathbf{l}_2$.

- Результатом вычисления становится множество — объединение всех построенных выше множеств. (См., например, 12.5.)

Примеры:

а) (см. пример из п. 5.1.3)

Конструктор множества week{Sunday .. Thursday, Saturday} задает множество будних дней мусульманского календаря.

б) Значение конструктора множества {0,5,8} есть то значение типа BITSET, о котором шла речь в примере из 5.4.3. Аналогично, значение конструктора множества {7,8} следует представлять себе как шкалу двоичных значений, в которой только 7-е и 8-е значения суть единицы, а значение конструктора множества {} — как шкалу, содержащую только нулевые значения.

10. ПРОЦЕДУРЫ

(Об относящихся к процедурам дополнениях, предоставляемых опцией M2EXTENSIONS системы XDS, см. Б.1.3 и Б.1.4)

Процедура (*собственно процедура* или *процедура-*

функция) есть объект (который может также рассматриваться как значение), порождаемый некоторым определённым полным описанием процедуры. Процедура представляется *процедурным текстом*, то есть соответственно блоком собственно процедуры или блоком процедуры-функции из порождающего полного описания. Заголовок процедуры, содержащийся в порождающем описании, и компоненты этого заголовка являются по определению заголовком самой процедуры и её соответствующими компонентами. Типом процедуры является такой тип (независимо от того, обозначается ли он некоторым описанным в программе идентификатором или нет), с которым согласован заголовок процедуры.

описание процедуры =
предварительное описание процедуры|
    полное описание процедуры;
полное описание процедуры =
    полное описание собственно процедуры|
    полное описание процедуры-функции;
полное описание собственно процедуры =
    заголовок собственно процедуры,
  блок собственно процедуры,
    идентификатор процедуры;
полное описание процедуры-функции =
    заголовок процедуры-функции,
  блок процедуры-функции,
    идентификатор процедуры;
блок собственно процедуры =
    вступительная часть блока,

```
      ["BEGIN",тело блока],"END";
блок процедуры-функции =
        вступительная часть блока,"BEGIN",
      тело блока,"END";
вступительная часть блока =
        {набор описаний констант                    |
         набор описаний типов                       |
         набор описаний переменных                  |
         описание процедуры,";"                     |
         описание локального модуля,";"};
тело блока =
полное тело блока |
      неполное тело блока;
полное тело блока =
      нормальная часть,"EXCEPT",
      исключительная часть;
неполное тело блока = нормальная часть;
нормальная часть = последовательность операторов;
исключительная часть =
      последовательность операторов;
```

Идентификатор процедуры, расположенный в конце полного описания процедуры, должен совпадать с идентификатором процедуры, входящим в соответствующий заголовок процедуры. Если полное описание процедуры есть полное описание процедуры-функции, то нормальная часть тела блока должна содержать оператор возврата из функции (см. 13.2).

И2) Полное описание процедуры исполняется так, как исполнялось бы описание константы, в котором идентификатор константы совпадал бы с

идентификатором процедуры, а константное выражение имело бы своим значением процедуру, порождаемую этим полным описанием.

Если идентификатор процедуры описан во вступительной части внутреннего модуля (см. 14.7) или во вступительной части блока процедуры, то он не может выступать ни в качестве выражения, входящего в оператор присваивания (после символа :=, см. 13.2), ни в качестве фактического параметра вызова процедуры или вызова функции (см. 13.3, 11, 10.2).

10.1. Предварительные описания процедур

```
предварительное описание процедуры =
        предварительное описание собственно
          процедуры|
        предварительное описание процедуры-
          функции;
предварительное описание собственно процедуры =
       заголовок собственно процедуры,
       "FORWARD";
предварительное описание процедуры-функции =
        заголовок процедуры-функции,"FORWARD";
```

Каждому предварительному описанию процедуры, непосредственно содержащемуся в некотором модуле определений **D** или вступительной части некоторого модуля **M** или некоторой процедуры **P**, должно соответствовать некоторое (единственное) полное описание процедуры, причём должны выполняться следующие условия:

- идентификаторы из заголовков обоих описаний совпадают;
- заголовки процедур из обоих описаний согласованы;
- предварительное описание предшествует полному описанию;
- полное описание непосредственно содержится в том же модуле **D** или той же вступительной части модуля **M** или процедуры **P**, или же идентификатор из заголовка полного описания подвергается неквалифицируемому экспорту (см. 14.4) из некоторого локального модуля в модуль **M**.

В результате предстоящего исполнения указанного полного описания процедуры идентификатор процедуры из заголовков обоих описаний начнёт в обоих определяющих вхождениях обозначать одну и ту же процедуру. Все использующие вхождения данного идентификатора процедуры, расположенные между данными двумя описаниями, находятся в авансируемой позиции.

10.2. Исполнение процедуры

Исполнение собственно процедуры происходит при исполнении вызова процедуры (см. 13.3), а исполнение процедуры-функции — при вычислении значения вызова функции (см. 11). Каждая из этих конструкций содержит список выражений (см. 12), рассматриваемых как фактические параметры. Количество фактических параметров в вызове процедуры или вызове функции должно быть равно количеству формальных параметров соответствующей процедуры. Фактический и формальный параметры, находящиеся на одном и том же по счёту месте соответственно в списке фактических параметров и в заголовке описания процедуры, считаются соответствующими друг другу. Фактический параметр, соответствующий формальному параметру – значению, может быть любым выражением, а соответствующий формальному параметру – переменной может быть только обозначением переменной. Характеристика типа формального параметра – значения (соответственно формального параметра – переменной) должна быть совместима по параметрам – значениям (совместима по параметрам – переменным) с типом соответствующего фактического параметра.

Исполнение процедуры состоит из следующих этапов:

- исполнение списка спецификаций формальных параметров заголовка процедуры;
- исполнение представляющего процедуру процедурного текста.

Исполнение списка спецификаций формальных параметров состоит в обработке (в некотором порядке) всех входящих в этот список формальных параметров.

Рассмотрим обработку формального параметра – значения.

- Если характеристикой типа формального параметра является некоторый тип **T**, то создаётся новая переменная **V** типа **T**. Если **T** есть системный тип памяти (см. 16.1), то производится системное наложение (см. 16.1.2) соответствующего фактического параметра на переменную **V**, а в противном случае значение фактического параметра присваивается переменной **V**.
- Если характеристикой типа формального параметра являются признаки открытого массива, состоящие из размерности 1 и литерного типа компоненты, а фактический параметр есть выражение общего строкового типа, характеризующегося длиной **m**, то создаётся новая переменная **V**, тип которой есть новый тип-массив, тип индекса которого есть тип-отрезок [0.. **m** – 1], а тип компоненты есть

литерный тип. Значение фактического параметра присваивается переменной **V**.

- Иначе, если характеристикой типа формального параметра являются признаки открытого массива, состоящие из размерности 1 и типа компоненты — системного типа памяти **S**, а тип фактического параметра есть некоторый тип **D**, то создаётся новая переменная **V**, тип которой есть новый тип-массив с типом компоненты **S** и типом индекса [0..**l** – 1], где **l** = SIZE(**D**)/SIZE(**S**) (см. 15.2.4), и производится системное наложение фактического параметра на перемен-
- Иначе, если характеристикой типа формального параметра являются признаки открытого массива, состоящие из размерности **n** и типа компоненты — системного типа памяти **S**, а тип фактического параметра может рассматриваться как тип – (**n** – 1)-мерный массив с типом компоненты **D** и типами индексов $\mathbf{I}_1$, ..., $\mathbf{I}_{\mathbf{n}-1}$, то создаётся новая переменная **V**, тип которой есть новый тип – **n**-мерный массив с типом компоненты **S** и типами индексов

  $\mathbf{J}_1$, ..., $\mathbf{J}_{\mathbf{n}-1}$, [0.. **l** – 1] ,

  где каждый тип $\mathbf{J}_\mathbf{k}$ ($0 \le \mathbf{k} \le \mathbf{l} - 1$) есть приведённый тип-отрезок типа $\mathbf{I}_\mathbf{k}$, а **l** = SIZE(**D**) / SIZE(**S**), и производится системное наложение фактического параметра на переменную **V**.

- Иначе, если характеристикой типа формального параметра являются признаки открытого массива, состоящие из размерности **n** и типа компоненты **C**, а тип фактического параметра может рассматриваться как тип – **n**-мерный массив с типом компоненты **D** и типами индексов $\mathbf{I}_1$, ..., $\mathbf{I}_n$, то создаётся новая переменная **V**, тип которой есть новый тип – **n**-мерный массив с типом компоненты **C** и типами индексов $\mathbf{J}_1$, ..., $\mathbf{J}_n$, где каждый тип $\mathbf{J}_k$ есть приведённый тип-отрезок типа $\mathbf{I}_k$, и значение фактического параметра присваивается переменной **V**.

Рассмотрим обработку формального параметра – переменной. Пусть фактический параметр есть обозначение переменной **W**.

- Если характеристикой типа формального параметра является некоторый тип **T**, то создаётся новая переменная **V** типа **T**, отождествляемая с переменной **W**.
- Иначе, если характеристикой типа формального параметра являются признаки открытого массива, состоящие из размерности 1 и типа компоненты — системного типа памяти **S**, а тип фактического параметра есть некоторый тип **D**, то создаётся новая переменная **V**, тип которой есть новый тип-массив с типом компоненты **S** и типом индекса [0.. **l** – 1], где

$$\mathbf{l} = \mathrm{SIZE}(\mathbf{D}) / \mathrm{SIZE}(\mathbf{S}),$$

причём для этой переменной отводятся те же ячейки памяти, которые были отведены для переменной **W**.

- Иначе, если характеристикой типа формального параметра являются признаки открытого массива, состоящие из размерности **n** и типа компоненты — системного типа памяти **S**, а тип фактического параметра может рассматриваться как тип – (**n** – 1)-мерный массив с типом компоненты **D** и типами индексов $\mathbf{I}_1$, ..., $\mathbf{I}_{\mathbf{n}-1}$, то создаётся новая переменная **V**, тип которой есть новый тип – **n**-мерный массив с типом компоненты **S** и типами индексов $\mathbf{J}_1$, ..., $\mathbf{J}_{\mathbf{n}-1}$, [0 .. **l** – 1], где каждый тип $\mathbf{J}_\mathbf{k}$ (0 ≤ **k** ≤ **l** – 1) есть приведённый тип-отрезок типа $\mathbf{I}_\mathbf{k}$, а **l** = SIZE(**D**) / SIZE(**S**), причём для этой переменной отводятся те же ячейки памяти, которые были отведены для переменной **W**.
- Иначе, если характеристикой типа формального параметра являются признаки открытого массива, состоящие из размерности **n** и типа компоненты **C**, а тип фактического параметра может рассматриваться как тип – **n**-мерный массив с типом компоненты **D** и типами индексов $\mathbf{I}_1$, ..., $\mathbf{I}_\mathbf{n}$, то создаётся новая переменная **V**, тип которой есть новый тип – **n**-мерный

массив с типом компоненты **C** и типами индексов $\mathbf{J}_1$, ..., $\mathbf{J}_n$, где каждый тип $\mathbf{J}_k$ есть приведённый тип-отрезок типа $\mathbf{I}_k$, отождествляемая с переменной **W**.

Во всех перечисленных случаях тот простой идентификатор **P**, которым является данный формальный параметр, начинает обозначать переменную **V**. Переменная **V** существует до окончания исполнения процедуры. В тех случаях, когда переменная **V** возникала при обработке формального параметра с характеристикой типа — признаками открытого массива, тип переменной **V** будем называть *типом открытого массива*.

(В дальнейшем, рассказывая о работе той или иной процедуры, будем в целях удобства говорить о **V** как о "переменной **P**", а о значении переменной **V** — как о "значении параметра **P**".)

Исполнение процедурного текста, представляющего процедуру, состоит в следующем:

- если процедура защищена, то осуществляется подъём уровня защиты соответствующей сопрограммы (см. 19);
- исполняются описания из вступительной части блока процедуры;
- исполняется нормальная часть тела блока процедуры;

- все идентификаторы — формальные параметры процедуры и все идентификаторы из описаний вступительной части блока процедуры перестают обозначать те объекты, которые они до того обозначали;
- процедура *финализируется*;
- если процедура защищена, то осуществляется опускание уровня защиты соответствующей сопрограммы (см. 19).

Финализация процедуры состоит в финал зации блоков тех внутренних модулей этой процедуры, которые инициализированы при данном исполнении процедуры (см. 14.7). Финализация происходит в порядке, обратном порядку инициализации указанных модулей.

О стандартных процедурах, предусмотренных системой, и об особенностях их задания и использования см. 15.

11. ВЫЗОВЫ ФУНКЦИЙ

Вызов функции есть частный случай выражения (см. 12).

вызов функции =
    обозначение,

"(",[список фактических параметров],")";
список фактических параметров =
фактический параметр,
{",",фактический параметр};
фактический параметр = выражение;

Обозначение, входящее в вызов функции, должно быть либо обозначением процедуры, либо обозначением переменной, тип которой есть тип-процедура-функция. Значением этого обозначения должна быть некоторая процедура-функция. Типом вызова функции является тип результата этой процедуры-функции.

Вычисление вызова функции осуществляется следующим образом:

- данная процедура-функция исполняется;
- результатом вычисления становится значение, которое было "присвоено" при исполнении оператора возврата из функции (см. 13.4.2) "результату" процедуры-функции.

Примеры (см. примеры из 5.6.1 и 7):

what_next_2 (red)
func_without_parms ()
a_function (what_next_2 (red))

12. ВЫРАЖЕНИЯ

Выражение есть либо основа, либо произвольное выражение, взятое в скобки, либо два выражения (*операнда*), соединенные символом операции, либо один операнд, предшествуемый символом операции.

```
выражение =
        простое выражение,
        [символ отношения,простое выражение];

простое выражение =
        слагаемое|
        [знак],слагаемое|
        простое выражение,
        символ операции типа сложения,слагаемое;
слагаемое =
        множитель|
        слагаемое,
        символ операции типа умножения,множитель;
множитель =
        "(",выражение,")"|
        символ отрицания,множитель|
        основа;
основа =
        изображение константы | обозначение|
        конструктор значения | вызов функции;
```

Выражение имеет некоторый определённый тип. При исполнении программы может происходить *вычисление* выражения, причём значение, явившееся результатом этого вычисления, становится (текущим) значением выражения.

Если тип того обозначения или вызова функции, которым является основа, есть тип-отрезок, то типом данной основы является объемлющий тип этого типа-отрезка.

Если основа есть изображение константы, то результатом её вычисления является значение этого изображения константы.

Если множитель есть выражение, взятое в скобки, то тип и результат вычисления этого множителя совпадают соответственно с типом и результатом вычисления выражения, находящегося в скобках.

Если выражение содержит два операнда, соединённых символом операции, или один операнд, предшествуемый символом операции, то считается, что данный символ представляет некоторую *операцию — инфиксную* или соответственно *префиксную* (арифметическую, логическую или иную), зависящую от типов операндов и определяющую тип рассматриваемого выражения ("результирующий тип"). Вычисление выражения состоит в вычислении этих операндов и выполнении данной операции над значениями операндов, причём результат операции становится значением выражения.

(Для всех остальных случаев правила, определяющие типы и результаты вычисления выражений, здесь не приводятся ввиду их тривиальности.)

Если результат вычисления выражения типа **T** не оказывается значением типа **T**, то возбуждается исключительная ситуация.

Частным случаем выражения является константное выражение.

константное выражение = выражение;

Выражение является константным, если оно удовлетворяет следующим двум условиям:

- выражение не содержит идентификаторов переменных (в том числе в составе квалифицируемых идентификаторов — см. 14.4, 14.5);
- если выражение содержит вызов функции, то соответствующее обозначение процедуры есть либо стандартный идентификатор (ABS, CAP, CHR, CMPLX, FLOAT, HIGH, IM, INT, LENGTH, LFLOAT, MAX, MIN, ODD, ORD, RE, SIZE, TRUNC, VAL — см. 15.2), либо импортированный из модуля SYSTEM идентификатор CAST (см. 16.6), MAKEADR (см. 16.3), ROTATE, SHIFT (см.16.4) или TSIZE (см.16.5); при этом на фактический параметр

вызова функции SIZE(**P**) первое из данных условий не распространяется.

12.1. Целочисленные операции

Инфиксная *целочисленная операция* выполняется над парой операндов, типы которых являются совместимыми по выражениям целочисленными типами. Если один из операндов имеет общий целочисленный тип, а другой операнд — некоторый целочисленный тип **T**, то результирующим типом является тип **T**, а в противном случае результирующим типом является (совпадающий) тип двух операндов. При данных типах операндов символ "+" представляет арифметическое сложение, символ "-" — арифметическое вычитание, символ "*" — арифметическое умножение, символы "/" и "DIV" — разные формы деления нацело, а символы "REM" и "MOD" — разные формы взятия остатка при делении нацело. Операции, представляемые символами "/" и "REM", осуществляют такое деление произвольного делимого на ненулевой делитель, что остаток от этого деления имеет тот же знак, что и делимое, а операции, представляемые символами "DIV" и "MOD", осуществляют такое деление произвольного делимого на положительный делитель, что остаток от этого деления неотрицателен.

Примеры:

Делимое	Делитель	Результаты операций			
		/	REM	DIV	MOD
11	2	5	1	5	1
11	–2	–5	1	*	*
–11	2	–5	–1	–6	1
–11	–2	5	–1	*	*

* — *исключительная ситуация*)

Префиксная целочисленная операция выполняется над операндом целочисленного типа, причём результирующим типом является тип операнда. При данных типах операнда символ "+" представляет операцию, копирующую значение операнда, а символ "-" — операцию, изменяющую знак этого значения. Последняя операция не применима к операнду типа CARDINAL.

12.2. Вещественные числовые операции

Инфиксная *вещественная числовая операция* выполняется над парой операндов, типы которых являются совместимыми по выражениям вещественными числовыми типами. Если один из операндов имеет общий вещественный числовой тип, а другой операнд — некоторый вещественный числовой тип **T**, то результирующим типом является тип **T**, а в противном случае результирующим типом является (совпадающий) тип двух операндов. При

ется (совпадающий) тип двух операндов. При данных типах операндов символы "+", "-", "*" и "/" представляют соответственно арифметическое сложение, арифметическое вычитание, арифметическое умножение или арифметическое деление вещественных чисел.

Префиксная вещественная числовая операция выполняется над операндом вещественного числового типа, причём результирующим типом является тип операнда. При данных типах операнда символ "+" представляет операцию, копирующую значение операнда, а символ "-" — операцию, изменяющую знак этого значения.

Все рассматриваемые операции осуществляются с точностью, определяемой реализацией для операндов соответствующих типов.

12.3. Комплексные числовые операции

Инфиксная *комплексная числовая операция* выполняется над парой операндов, типы которых являются совместимыми по выражениям комплексными числовыми типами. Если один из операндов имеет общий комплексный числовой тип, а другой операнд — некоторый комплексный числовой тип **T**, то результирующим типом является тип **T**, а в противном случае результирующим типом является (совпадающий) тип двух операндов.

При данных типах операндов символы "+", "-", "*" и "/" представляют, соответственно, арифметическое сложение, арифметическое вычитание, арифметическое умножение или арифметическое деление комплексных чисел.

Префиксная комплексная числовая операция выполняется над операндом комплексного числового типа, причём результирующим типом является тип операнда. При данных типах операнда символ "+" представляет операцию, копирующую значение операнда, а символ "-" — операцию, изменяющую знаки действительной и мнимой частей этого значения.

Во всех рассматриваемых операциях действительная и мнимая части результирующего значения находятся с точностью, определяемой реализацией для значений соответствующих вещественных числовых типов.

12.4. Логические операции

Инфиксная и префиксная *логические операции* выполняются соответственно над парой операндов или одним операндом логического типа, причём результирующим типом является логический тип. Символ "OR" представляет операцию логического сложения, символ логического умножения — операцию логического умножения и символ отрицания — операцию логического отрицания.

При выполнении операции логического сложения

(логического умножения) вначале вычисляется левый операнд. Если результат вычисления есть ***истина*** (соответственно, ***ложь***), то результат операции также есть ***истина*** (***ложь***) и правый операнд не вычисляется. В противном случае вычисляется правый операнд и результат этого вычисления становится результатом операции.

Результат операции логического отрицания есть ***истина*** (соответственно, ***ложь***) в том случае, если значение операнда есть ***ложь*** (соответственно, ***истина***).

12.5. Операции над множествами

Инфиксная *операция над множествами* выполняется:

- над парой операндов одного и того же типа-множества, причём результирующим типом является тот же тип-множество;
- над парой операндов одного и того же типа-упакованного-множества, причём результирующим типом является тот же тип-упакованное-множество.

Будем обозначать множества, являющиеся значениями левого и правого операндов, соответственно как **L** и **R**. При данных типах операндов символ "+" представляет операцию объединения множеств, символ "-" — операцию

вычитания множеств, символ "*" — операцию пересечения множеств и символ "/" — операцию симметрического вычитания множеств.

Результаты этих операций суть:

- в случае операции объединения множеств — множество всех элементов, каждый из которых есть либо элемент множества **L**, либо элемент множества **R**, либо элемент того и другого;
- в случае операции вычитания множеств — множество всех элементов, каждый из которых есть элемент множества **L**, но не является элементом множества **R**;
- в случае операции пересечения множеств — множество всех элементов, каждый из которых есть одновременно элемент множества **L** и элемент множества **R**;
- в случае операции симметрического вычитания множеств — множество всех элементов, каждый из которых есть элемент одного и только одного из двух множеств **L** или **R**.

Пример:

В следующей таблице значения выражений, помещённых в левом столбце, совпадают со значениями соответствующих конструкторов множеств из правого столбца:

{0,5,8} + {7,8}	{0,5,7,8}

{0,5,8} + {}	{0,5,8}
{0,5,8} – {7,8}	{0,5}
{0,5,8} – {}	{0,5,8}
{0,5,8} * {7,8}	{8}
{0,5,8} * {}	{}
{0,5,8} / {7,8}	{0,5,7}
{0,5,8} / {}	{0,5,8}

(О дополнении, предоставляемом опцией M2EXTENSIONS системы XDS, см. Б.1.5)

12.6. Операция над строками

Единственной инфиксной *операцией над строками* является операция сцепления строк, представляемая символом "+". Она выполняется над парой операндов общего строкового типа (то есть строковых констант), причём результирующим типом является общий строковый тип. Результат этой операции есть строка, состоящая из последовательных литерных значений значения левого операнда и следующих за ними последовательных литерных значений значения правого операнда.

12.7. Операции сравнения

Для всех инфиксных *операций сравнения* результирующим типом является логический тип.

12.7.1. Операции сравнения комплексных чисел

Инфиксная операция сравнения комплексных чисел выполняется над парой операндов, типы которых суть совместимые по выражениям комплексные числовые типы. При данных типах операндов символ "=" представляет операцию равенства, а символ неравенства — операцию неравенства. Результат операции равенства есть ***истина***, а операции неравенства — ***ложь*** в том и только том случае, когда значения операндов (вычисленные с учётом свойственного данному типу приближения) совпадают.

12.7.2. Операции сравнения вещественных чисел и сравнения значений порядкового типа

Инфиксная операция сравнения вещественных чисел или сравнения значений порядкового типа выполняется над парой операндов, типы которых суть совместимые по выражениям вещественные числовые или, соответственно, порядковые типы. При данных типах операндов символ "=" представляет операцию равенства, символ неравенства — операцию неравенства, символ "<" — операцию "меньше", символ ">" — операцию "больше", символ "<=" — операцию "меньше или равно" и символ ">=" — операцию "больше или равно". Результат каждой такой операции есть ***истина*** в том и только том случае, когда для значения левого операнда и

значения правого операнда (в случае вещественных чисел, вычисленных с учётом свойственного данному типу приближения) выполняется соответствующее данной операции арифметическое отношение.

Примеры (см. пример из п. 5.1.3):

2 * 2 = 4
red < orange
Thursday > Monday
FALSE < TRUE

Результат каждой из этих операций есть ***истина***.

12.7.3. Операции сравнения множеств

Инфиксная операция сравнения множеств выполняется над парой операндов, типы которых суть

- идентичные типы-множества
- или идентичные типы-упакованные-множества.

При данных типах операндов символ "=" представляет операцию "совпадает с", символ неравенства — операцию "не совпадает с", символ "<=" — операцию "есть подмножество" и символ ">=" — операцию "есть надмножество".

Результат операции "совпадает с" есть ***истина***, а результат операции "не совпадает с" есть ***ложь*** в том и только том случае, когда оба множества — значения операндов состоят из одних и тех же элементов.

Результат операции "есть подмножество" есть ***истина*** в том и только том случае, когда каждый элемент множества — значения левого операнда является также элементом множества — значения правого операнда.

Результат операции "есть надмножество" есть ***истина*** в том и только том случае, когда каждый

элемент множества — значения правого операнда является также элементом множества — значения левого операнда.

Пример:

Результирующим значением всех следующих операций является ***истина***.

```
{8,5,0}          =   {0,5,8}
{0,5,8}          =   BITSET {0,5,8}
{0,5,8} + {7,8}=   {0,5,7,8}
{0,5,8}          #   {0,5,7}
{0,5,8}          #   {}
{0,5,8}          <= {0,5,7,8}
{}               <=  {0,5,8}
{0,5,8}          >= {0,8}
{0,5,8}          >= {}
```

12.7.4. Операция принадлежности множеству

Инфиксная операция принадлежности множеству представляется символом "IN". Она выполняется над такой парой операндов, в которой правый операнд имеет своим типом некоторый тип-множество, базовый тип которого совместим по присваиванию с типом левого операнда.

Результат этой операции есть ***истина*** в том и только том случае, когда значение левого операнда является элементом множества — значения правого операнда.

Пример:

Результатом операции

```
5 IN {0,5,8}
```

является ***истина***, а результатом каждой из операций

```
5 IN {0,8}
5 IN {}
```

является ***ложь***.

12.7.5. Операции сравнения процедур

Инфиксная операция сравнения процедур выполняется над парой операндов, типы которых суть идентичные типы-процедуры. При данных типах операндов символ "=" представляет операцию "совпадает с", а символ неравенства — операцию "не совпадает с". Результирующее значение операции "совпадает с" есть ***истина***, а операции "не совпадает с" есть ***ложь*** в том и только том случае, когда значения обоих её операндов являются одной и той же процедурой (процедурой, порождённой одним и тем же полным описанием).

Пример:

Пусть в области видимости описаний

```
VAR p: PROC;
PROCEDURE p1; BEGIN END p1;
PROCEDURE p2; BEGIN END p2;
```

исполняется оператор

p:=p1;

Тогда результатом операции

p=p1

будет ***истина***, а результатом операции

p1=p2

— ***ложь***.

12.7.6. Операции сравнения указателей

Инфиксная операция сравнения указателей выполняется над такой парой операндов, что:

- либо оба они суть переменные одного и того же скрытого типа,
- либо ни один из них не есть переменная скрытого типа, причём
  - типы их суть идентичные типы-указатели,
  - или каждый из них имеет своим значением ***псевдоимя***,
  - или тип одного из них есть тип-указатель, а другой имеет своим значением ***псевдоимя***.

При данных типах операндов символ "=" представляет операцию "совпадает с", а символ неравенства — операцию "не совпадает с". Результат операции "совпадает с" есть ***истина***, а операции "не совпадает с" есть ***ложь*** в том и только том случае, когда либо значения обоих её операндов являются указателями, ссылающимися на одну и ту же переменную, либо каждое из этих значений есть ***псевдоимя***.

Пример:

В области видимости описаний

VAR p: POINTER TO REAL; b1,b2: BOOLEAN;

после исполнения операторов

p := NIL; b1:= NIL=NIL; b2 := NIL=p;

каждая из переменных b1 и b2 имеет значение ***истина***.

См. также 15.1 (пример к процедурам NEW и DISPOSE, п. д).

13. ОПЕРАТОРЫ

оператор =
пустой оператор | оператор присваивания |
вызов процедуры | оператор возврата |
оператор присоединения |
условный оператор | оператор выбора |
цикл пока | цикл до | безусловный цикл |
оператор выхода | цикл с шагом
|оператор повтора
;

последовательность операторов =
оператор,{";",оператор};

Исполнение последовательности операторов, входящей в какую-либо конструкцию языка, состоит в последовательном исполнении операторов этой последовательности.

13.1. Пустые операторы

Пустой оператор не содержит символов. При его исполнении не совершается никаких действий.

пустой оператор = ;

13.2. Операторы присваивания

оператор присваивания =
обозначение,":=",выражение;

Обозначение в операторе присваивания должно быть обозначением некоторой переменной **V** типа, совместимого по присваиванию с типом (одним из типов) выражения (будем его называть **E**). При этом

- либо **V** и **E** являются переменными одного и того же скрытого типа,
- либо ни **V**, ни **E** не является переменной скрытого типа.

В этих условиях оператор присваивания присваивает значение выражения **E** переменной **V**.

Примеры:

а) Пусть имеется следующее описание переменных:

VAR ten:[1..10]; hundred:[1..100];

Оба оператора присваивания

```
ten:=hundred
hundred:=ten
```

законны, поскольку входящие в них выражения являются обозначениями, имеющими (согласно правилу из 12) тип CARDINAL.

Если текущее значение переменной hundred есть 10, то исполнение первого из этих двух операторов пройдет нормальным образом; если же значение это есть 11, то при попытке исполнения данного оператора возбуждается исключительная ситуация.

б) Пусть имеются описания:

```
VAR   i: INTEGER; ri: [-100..100]; c: CARDINAL;
      rc: [0..100];r: REAL; lr: LONGREAL;
```

Тогда

- каждый из типов CARDINAL, INTEGER, [0..100], [-100..100] совместим по присваиванию с типами выражений i, ri, c, rc, -1, 10 и не совместим по присваиванию с типами выражений r, lr и 10.0;
- тип REAL совместим по присваиванию с типами выражений r, 10.0 и не совместим по присваиванию с типами выражений i, ri, c, rc, lr, -1 и 10;
- тип LONGREAL совместим по присваиванию с типами выражений lr, 10.0 и не совместим по присваиванию с типами выражений i, ri, c, rc, r, -1 и 10.

Пусть исполнены далее операторы присваивания i:=-10 и ri:=-100. Тогда при попытке исполнения каждого из операторов присваивания

c:=i; c:=ri; c:=-1; rc:=i; rc:=ri; rc:=-1

возбуждается исключительная ситуация.

в) Тип ARRAY [0..3] OF CHAR совместим по присваиванию с типами выражений "abc" и "abcd" и не совместим по присваиванию с типом выражения "abcde".

13.3. Вызовы процедур

вызов процедуры =
обозначение,
["(",[список фактических параметров],")"];

Обозначение, входящее в вызов процедуры, должно быть либо обозначением процедуры, либо обозначением переменной, тип которой есть тип-собственно-процедура. Значением этого обозначения должна быть некоторая собственно процедура. Исполнение вызова процедуры состоит в исполнении этой собственно процедуры.

Примеры (см. примеры из пп. 5.1.3, 5.6.1 и 7):

proc_without_parms
a_procedure (orange, colour)

13.4. Операторы возврата

оператор возврата =
    простой оператор возврата|
    оператор возврата из функции;

13.4.1. Простые операторы возврата

простой оператор возврата = "RETURN";

Простой оператор возврата может служить лишь оператором, содержащимся в теле блока собственно процедуры или теле блока модуля (см. 14). Исполнение простого оператора возврата состоит в завершении исполнения данного тела блока.

13.4.2. Операторы возврата из функций

оператор возврата из функции =
        "RETURN", выражение;

Оператор возврата из функции может служить лишь оператором, содержащимся в теле блока процедуры-функции. Тип результата соответствующей процедуры-функции должен быть совместим по присваиванию с типом выражения. Исполнение оператора возврата из функции состоит в вычислении значения выражения, "присваивании" этого значения (по правилам осуществления присваивания) "результату" процедуры-функции и завершении исполнения данного тела блока.

исполнения данного тела блока.

Пример описания простейшей процедуры-функции, содержащего оператор возврата из функции (см. пример из п. 5.6.1):

```
PROCEDURE func_without_parms (): INTEGER;
BEGIN RETURN 72*12345679
END func_without_parms;
```

Значение вызова функции func_without_parms () есть число 888888888.

13.5. Операторы присоединения

оператор присоединения =
"WITH",обозначение-запись,
"DO",последовательность операторов,"END";

Пусть обозначение-запись есть **R**. Тогда исполнение оператора присоединения состоит в исполнении содержащейся в нём последовательности операторов при подразумеваемой текстуальной замене любого вхождения идентификатора **f** в позиции, которую может занимать обозначение-поле **R.f**, на это **R.f**.

Пример:

В условиях примера из 8.2 провозглашение Голливуда отдельным штатом в составе Соединённых Штатов Америки может

сопровождаться выполнением следующего оператора:

```
WITH USA
DO  number_of_members_of_the_federation:=
       number_of_members_of_the_federation + 1
END     ,
```

равносильным оператору

```
USA.number_of_members_of_the_federation:=
USA.number_of_members_of_the_federation + 1
```

13.6. Условные операторы

```
условный оператор =
       начало условного,[остаточное],"END";
начало условного =
       "IF",условие,
       "THEN",последовательность операторов,
       {"ELSIF",условие,
        "THEN",последовательность операторов};
условие = выражение;
остаточное =
       "ELSE",последовательность операторов;
```

Условие есть выражение логического типа. Исполнение условного выражения состоит в том, что входящие в него условия последовательно вычисляются до тех пор, пока значением очередного

условия не окажется ***истина***, после чего исполняется последовательность операторов, отделяемая от этого условия ключевым словом THEN. Если ни одно из условий не имеет значения ***истина*** и при этом присутствует остаточное, то исполняется последовательность операторов из остаточного, а иначе никакие дальнейшие действия не совершаются.

Пример (см. пример из 7):

```
IF length<0.0 OR width<0.0
THEN area:=-1.0
ELSIF length>1000.0 OR width>1000.0
THEN area:=-1000000.0
ELSE area:=length∗width
END
```

13.7. Операторы выбора

```
оператор выбора =
      "CASE",селектор,
      "OF",список альтернатив,"END";
селектор = выражение;
список альтернатив =
       альтернатива,
      {разделитель вариантов,альтернатива},
```

```
        [остаточное];
альтернатива =
        [список меток альтернативы,":",
         последовательность операторов];
список меток альтернативы =
        метка альтернативы,{",",метка альтернативы};
метка альтернативы =
        константное выражение,
        ["..",константное выражение];
```

Селектор есть выражение некоторого порядкового типа, совместимого по значениям с типами константных выражений, содержащихся в метках альтернатив. Если метка альтернативы содержит два константных выражения, то значения этих, взятых по порядку, выражений задают — в качестве соответственно нижней и верхней границы — некоторый, относящийся к этой метке альтернативы, отрезок значений данного порядкового типа. Значение любого константного выражения, совпадающего с некоторой меткой альтернативы, и любое значение, принадлежащее какому-нибудь из отрезков, относящихся к некоторой метке альтернативы, будем считать значением, связанным с этой меткой альтернативы, причём никакое значение не может быть одновременно связано с двумя разными метками альтернатив.

В предположении, что все значения, связанные с

метками альтернатив, уже определены, дальнейшее исполнение оператора выбора происходит следующим образом:

- вычисляется значение селектора;
- если это значение есть значение, связанное с некоторой меткой альтернативы, то оно начинает "выбирать" соответствующую альтернативу;
- иначе, если оператор выбора содержит остаточное, то это значение начинает выбирать остаточное, а в противном случае возбуждается исключительная ситуация;
- исполняется последовательность операторов из выбираемых этим значением альтернативы или остаточного.

Пример:

Допустим, что к описаниям, о которых говорилось в примере 8.2, добавлены следующие описания переменных:

```
VAR unit_or_feder, rep_or_mon: BOOLEAN;
        number_of_members: CARDINAL;
```

Тогда приводимый ниже оператор осуществляет перенос значений полей переменной-записи USA в вышеописанные переменные.

```
WITH USA
DO unit_or_feder:=a_unitary_state_or_a_federation;
    CASE unit_or_feder OF
        TRUE:rep_or_mon:=a_republic_or_a_monarchy
```

```
        |FALSE:
          number_of_members:=
                  number_of_members_of_the_federation
      END
  END
```

13.8. Циклы пока

```
цикл пока =
      "WHILE",выражение,
      "DO",последовательность операторов,"END";
```

Тип выражения должен быть логическим типом. Исполнение цикла пока осуществляется следующим образом:

- вычисляется значение выражения;
- если это значение есть ***ложь***, то исполнение оператора завершается, а если оно есть ***истина***, то исполняется расположенная после "DO" последовательность операторов и процесс исполнения начинается заново.

(Таким образом, последовательность операторов может быть исполнена несколько раз, не исполнена ни одного раза, а также исполняться неограниченное число раз.)

Пример (см. примеры из 7):

```
k:=0; p:=p0;
```

```
WHILE p<=p1
DO IF A[p]=0 THEN k:=k+1 END; p:=p+1
END
```

(Предполагается, что 1≤p0≤100 и 1≤p1≤100.)

В результате выполнения этих операторов значением переменной k станет число нулей, содержащихся в интервале компонент массива A от компоненты A[p0] до компоненты A[p1].

13.9. Циклы до

```
цикл до = "REPEAT",
      последовательностьоператоров,
      "UNTIL",выражение;
```

Тип выражения должен быть логическим типом. Исполнение цикла до осуществляется следующим образом:

- исполняется расположенная после "REPEAT" последовательность операторов;
- вычисляется значение выражения;
- если это значение есть ***истина***, то исполнение оператора завершается, а если оно есть ***ложь***, то процесс исполнения начинается заново.

(Таким образом, последовательность операторов может быть исполнена один или большее число раз, а также исполняться неограниченное число раз.)

Пример (см. примеры из 7):

```
k:=0; p:=p0;
IF p1>=p0
THEN REPEAT IF A[p]=0 THEN k:=k+1 END;
                    p:=p+1
        UNTIL p>p1
END
```

(Предполагается то же условие, что и в предыдущем примере.)
Этот фрагмент программы эквивалентен по своему эффекту фрагменту из предыдущего примера.

13.10. Безусловные циклы

безусловный цикл =
"LOOP",последовательность операторов,
"END";

Исполнение безусловного цикла состоит в последовательном исполнении содержащейся в нем последовательности операторов. (Исполнение это может быть незавершаемым, но может и завершиться при исполнении какого-либо содержащегося в безусловном цикле оператора выхода — см. 13.11.)

Пример:

```
LOOP END
```

Этот оператор бесконечно долго ничего не делает.

13.11. Операторы выхода

оператор выхода = "EXIT";

Оператор выхода может находиться только внутри безусловного цикла (на любом уровне вложенности операторов). Исполнение оператора выхода состоит в завершении исполнения ближайшего охватывающего его безусловного цикла.

Пример (см. примеры из 7):

```
k:=0; p:=p0;
LOOP IF p>p1 THEN EXIT
        ELSIF A[p]=0 THEN k:=k+1
        END;
        p:=p+1
END
```

Условия и эффект исполнения — такие же, как и в примерах из 13.8 и 13.9.

13.12. Циклы с шагом

цикл с шагом =
    "FOR",простой идентификатор,":=",

выражение для начала,
"TO",выражение для конца,
["BY",величина шага],
"DO",последовательность операторов,"END";
выражение для начала = выражение;
выражение для конца = выражение;
величина шага = константное выражение;

Идентификатор, расположенный после "FOR", должен быть идентификатором некоторой переменной, называемой *параметром цикла*. Типами параметра цикла, выражения для начала и выражения для конца должны быть порядковые типы. Величина шага должна иметь целочисленный тип. Тип параметра цикла должен быть совместим по присваиванию с типами выражения для начала и выражения для конца. Кроме того, на параметр цикла накладываются следующие ограничения:

- идентификатор **I** параметра цикла должен быть описан в списке идентификаторов переменных некоторого описания переменных того блока модуля или блока процедуры, который непосредственно охватывает данный цикл с шагом;
- идентификатор **I** не может подвергаться ни экспорту, ни импорту (см. 14.4, 14.5);
- ни в последовательности операторов цикла с шагом, ни в каком-либо описании процедуры,

содержащемся во вступительной части блока, тело которого непосредственно содержит цикл с шагом, идентификатор **I** не может быть использован ни как обозначение в операторе присваивания, ни как фактический параметр, соответствующий формальному параметру – переменной, ни как параметр цикла некоторого другого цикла с шагом.

(Цель этих ограничений состоит в том, чтобы ни при каком возможном исполнении цикла с шагом не могло произойти изменение значения параметра цикла.)

Считается, что перед началом исполнения цикла с шагом происходит вычисление величины шага, если последняя явно присутствует в цикле. Полученное значение не должно быть равно 0. При дальнейшем определении исполнения цикла с шагом будем использовать стандартную процедуру-функцию VAL (см. 15.2.1), так что вызов функции VAL(INTEGER,**N**), где **N** есть выражение некоторого порядкового типа с представлением **T**, будет обозначать порядковый номер значения **N** в множестве, определяемом этим порядковым типом, а VAL(**T**,**n**) будет обозначать то значение порядкового типа, порядковый номер которого есть **n**. При подразумеваемом наличии описания

VAR **p**,**q**: INTEGER

исполнение цикла с шагом

FOR **I**:=**A** TO **B** BY **C** DO **S** END ,

где тип параметра цикла **I** имеет представление **T**, равносильно исполнению следующего фрагмента программы:

p:=VAL(INTEGER, **A**); **q**:=VAL(INTEGER, **B**);
WHILE (**C**>0) & (**p** <= **q**) OR (**C**<0) & (**p** >= **q**)
DO **I**:=VAL(**T**,**p**); **S**; **p**:= **p** + **C**
END ,

а исполнение цикла с шагом

FOR **I**:= **A** TO **B** DO **S** END

равносильно исполнению цикла с шагом

FOR **I**:= **A** TO **B** BY 1 DO **S** END .

После завершения исполнения значение переменной **I** становится неопределённым. Из данного выше описания следует, что последовательность операторов **S** может быть исполнена нуль, один или большее число раз.

Примеры:

а) (См. примеры из 7)
Оба оператора

FOR k:=1 TO 100 DO A[k]:=0 END и
FOR k:=100 TO 1 BY -1 DO A[k]:=0 END

полностью заполняют массив A нулями. Оператор

FOR k:=3 TO 100 BY 3 DO A[k]:=0 END

заполняет нулями каждую третью компоненту

массива (с индексами 3, 6, ..., 99), а операторы

```
FOR k:=10 TO 1 DO A[k]:=0 END    и
FOR k:=10 TO 100 BY -1 DO A[k]:=0 END
```

не вносят в массив никаких изменений.

б) В условиях примера а) из 8.1 и при наличии описания переменных

```
VAR d: week
```

оператор

```
FOR d:=Monday TO Friday
DO Johnson[d,12]:=FALSE
END
```

вводит для нашего гипотетического "служащего" ежедневный обеденный перерыв.

13.13. Операторы повтора

оператор повтора = RETRY;

Операторы повтора встречаются в исключительных частях тел блоков. Об их использовании см. в 18.

14. МОДУЛИ

Программный модуль, модуль определений, модуль реализации, а также описание локального модуля содержат повторенный дважды идентификатор

соответствующего модуля. Каждая из этих конструкций может содержать списки импорта, а описание локального модуля — также и список экспорта. Каждая из этих конструкций, кроме модуля определений, содержит также блок модуля. Будем, кроме того, считать, что внешний модуль также содержит неявный блок модуля.

Каждая из этих конструкций, кроме модуля определений, может содержать также задание уровня защиты, то есть константное выражение типа защиты.

```
блок модуля =
   вступительная часть блока,
        "BEGIN",[инициализирующее тело блока,]
        "FINALLY",
        финализирующее тело блока,
        "END"|
         вступительная часть блока,
         ["BEGIN",инициализирующее тело блока,]
         "END";
инициализирующее тело блока = тело блока;
 финализирующее тело блока = тело блока;
```

14.1. Локальные модули

```
описание локального модуля =
        "MODULE",
        простой идентификатор модуля,
        возможная защита,
```

```
        {список импорта},[список экспорта],
        блок модуля,
        простой идентификатор модуля;
простой идентификатор модуля =
        простой идентификатор;
возможная защита =
        левая квадратная скобка,
        задание уровня защиты,
        правая квадратная скобка,";"|
        ";";
задание уровня защиты = константное выражение;
 список импорта =
        список простого импорта|
        список неквалифицируемого импорта;
список простого импорта =
        "IMPORT",список идентификаторов,";";
список неквалифицируемого импорта =
        "FROM",идентификатор модуля,
        "IMPORT",список идентификаторов,";";
список экспорта =
        список неквалифицируемого экспорта|
        список квалифицируемого экспорта;
список неквалифицируемого экспорта =
        "EXPORT",список идентификаторов,";";
список квалифицируемого экспорта =
        "EXPORT","QUALIFIED",
        список идентификаторов,";";
```

В результате исполнения описания локального модуля идентификатор модуля начинает обозначать *локальный модуль*, совпадающий с этим описанием, после чего этот локальный модуль исполняется (см. 14.7). Локальный модуль считается вложенным в некоторый модуль или некоторую процедуру, если он непосредственно содержится во вступительной части блока этого модуля или этой процедуры.

14.2. Раздельные модули

Раздельные модули входят в программу парами, состоящими каждая из модуля определений и модуля реализации с одним и тем же идентификатором (так что этот идентификатор обозначает оба данные модуля). Цель такой парности состоит в том, чтобы разработчик раздельных модулей (и в частности, системных и стандартных модулей — см. 16, 17) мог вносить скрытые изменения в реализацию этих модулей, не беспокоя их пользователей и будучи сам защищён от вмешательства со стороны последних. Для пользователя доступна (через импорт) информация, содержащаяся в модуле определений, и недоступна та информация, которая содержится в модуле реализации. Детальней об этом будет сказано ниже.

В программу входят только те раздельные модули, из которых производится импорт в какой-либо входящий в программу модуль.

```
модуль определений =
        "DEFINITION","MODULE",
         простой идентификатор модуля,";",
        {список импорта},
        {набор описаний констант |
          набор описаний и определений типов |
          набор описаний переменных |
          определение процедуры,";"},
          "END",простой идентификатор модуля,".";
набор описаний и определений типов =
        "TYPE",{описание или определение типа,";"};
описание или определение типа =
        описание типа|определение скрытого типа;
определение скрытого типа =
        простой идентификатор;
определение процедуры = заголовок процедуры;


модуль реализации =
        "IMPLEMENTATION","MODULE",
        идентификатор модуля,
        возможная защита,{список импорта},
        блок модуля,идентификатор модуля,".";
```

Если простой идентификатор описан в некотором описании модуля определений, то соответствующая область видимости считается расширенной за счёт блока модуля реализации.

Идентификатор процедуры, входящий в определение процедуры модуля определений, должен быть описан в полном описании процедуры из вступительной части блока модуля реализации, причём оба заголовка процедур, входящие в это определение и это описание, должны быть согласованы друг с другом. Считается при этом, что данный идентификатор процедуры описывается также в определении процедуры, обозначая в соответствующей области видимости ту же процедуру, которую он обозначает в модуле реализации.

Пример

Здесь этой идентификатор, входящий в определение скрытого типа модуля определений, должен быть описан в описании типа из вступительной части блока модуля реализации или же описан в описании типа из вступительной части блока локального модуля и подвергнут неквалифицируемому экспорту (см. 14.4) из этого модуля в модуль реализации. Тип, обозначаемый указанными идентификаторами типа, может быть только типом-указателем (в частности, адресным типом). При этом считается, что данный идентификатор описывается также в определении скрытого типа, обозначая в соответствующей области видимости тот же тип-указатель, который он обозначает в модуле реализации. В указанной области видимости этот тип-указатель рассматривается как *скрытый тип*, а данный идентификатор (или соответствующий квалифицируемый идентификатор)

рассматривается как *идентификатор* (данного) *скрытого типа*. Эта "скрытость" означает, что в данной области видимости связанный тип соответствующего типа-указателя "не известен" и что на использование переменных, тип которым в той же области придан с помощью этого идентификатора, накладываются вытекающие отсюда ограничения (см. 8.3, 13.2). Реальная работа с такими переменными может осуществляться только с помощью процедур, описанных в модуле определений с помощью определений процедур и реализованных в модуле реализации с помощью внутренних средств этого модуля.

14.3. Программный модуль

программный модуль =
    "MODULE",идентификатор модуля,
    возможная защита,{список импорта},
    блок модуля,идентификатор модуля,".";

14.4. Экспорт из модулей

Если при исполнении списка неквалифицируемого экспорта некоторого локального модуля, обозначаемого идентификатором **M**, некоторый простой идентификатор **x** подвергается *неквалифицируемому экспорту* из данного модуля, то это означает, что область видимости этого идентификатора расширяется путём включения в неё

тификатора расширяется путём включения в неё блока того модуля или той процедуры, в которые вложен данный локальный модуль, и что в этом блоке использование как простого идентификатора **x**, так и квалифицируемого идентификатора **M**.**x** становится равносильным использованию идентификатора **x** в локальном модуле.

Если при исполнении списка квалифицируемого экспорта некоторого локального модуля, обозначаемого идентификатором **M**, или при исполнении модуля определений, обозначаемого идентификатором **M**, некоторый простой идентификатор **x** подвергается *квалифицируемому экспорту* из данного модуля, то это означает, что область видимости этого идентификатора расширяется путём включения в неё блока того модуля или той процедуры, в которые вложен данный модуль, и что в этом блоке использование квалифицируемого идентификатора **M**.**x** становится равносильным использованию идентификатора **x** в локальном модуле.

(© дополнении, предоставляемом опцией M2EXTENSIONS системы XDS, см. Б.1.6)

14.5. Импорт в модули

Если локальный или раздельный модуль, обозначаемый идентификатором **M**, вложен в некоторый модуль (или — в случае локального

модуля — в некоторую процедуру) **P** и при исполнении списка простого импорта этого локального или раздельного модуля некоторый простой идентификатор подвергается *простому импорту* в данный модуль, то это означает следующее:

- если простому импорту подвергается простой идентификатор **x**, то соответствующая область видимости расширяется путём включения в неё блока данного локального или раздельного модуля (так что в этом блоке использование идентификатора **x** оказывается равносильным использованию того же идентификатора в блоке модуля или процедуры **P**);
- если (независимо от предыдущего) простому импорту подвергается идентификатор модуля **L**, а некоторый квалифицируемый идентификатор **L.x** обозначает в блоке модуля (процедуры) **P** некоторый объект, то область видимости простого идентификатора **x**, обозначающего этот объект, расширяется путём включения в неё блока данного локального (раздельного) модуля и в этом блоке использование квалифицируемого идентификатора **L.x** становится равносильным использованию того же квалифицируемого идентификатора в блоке модуля или процедуры **P**.

Если некоторый локальный или раздельный модуль

вложен в модуль (или процедуру) **P** и при исполнении списка неквалифицируемого импорта этого локального (раздельного) модуля некоторый простой идентификатор **x** подвергается *неквалифицируемому импорту* из некоторого модуля **L** в данный локальный (раздельный) модуль, то это означает, что область видимости простого идентификатора **x** расширяется путём включения в неё блока данного модуля и в этом блоке использование этого идентификатора становится равносильным использованию квалифицируемого идентификатора **L**.**x** в блоке модуля (процедуры) **P**.

(О дополнении, предоставляемом опцией M2EXTENSIONS системы XDS, см. Б.1.7)

14.6. Неявный экспорт и импорт

Если идентификатор типа-перечисления подвергается неквалифицируемому экспорту, квалифицируемому экспорту, простому импорту или неквалифицируемому импорту, то одновременно с ним подвергаются соответственно неквалифицируемому экспорту, квалифицируемому экспорту, простому импорту или неквалифицируемому импорту описанные одновременно с ним идентификаторы (то есть, идентификаторы, входящие в список идентификаторов явного представления данного типа-перечисления).

14.7. Исполнение модулей

Локальный модуль считается *внутренним модулем* некоторой процедуры, если он содержится в блоке этой процедуры или во внутреннем модуле этой процедуры. Остальные исполняемые модули называются *статическими*.

Исполнение модуля состоит из следующих этапов:

- модуль инициализируется;
- если модуль является программным, то он *финализируется*.

Инициализация модуля выполняется следующим образом:

- *исполняются* в порядке их встречаемости списки импорта данного модуля;
- если имеется блок данного модуля, то этот блок *инициализируется*;
- если данный модуль есть локальный модуль, имеющий список экспорта, то исполняется этот список;
- если данный модуль есть модуль определений, то идентификатор этого модуля и каждый идентификатор, описанный в описаниях и определениях модуля, подвергается квалифицируемому экспорту из данного модуля.

Исполнение списка простого импорта некоторого модуля **Q** заключается в выполнении следующих действий для каждого очередного идентификатора из

этого списка:

- если данный идентификатор есть идентификатор некоторых ранее не инициализированных и не инициализирующихся в данный момент раздельных модулей, то эти модули инициализируются (вначале модуль определений, а затем модуль реализации);
- если данный идентификатор не был подвергнут простому импорту в модуль **Q**, то он подвергается этому импорту.

Исполнение списка неквалифицируемого импорта некоторого модуля **Q**, непосредственно содержащего в себе идентификатор модуля **R** и некоторый список идентификаторов **L**, заключается в следующем:

- если **R** есть идентификатор некоторых ранее не инициализированных и не инициализирующихся в данный момент раздельных модулей, то эти модули инициализируются (как сказано выше);
- все не подвергнутые простому импорту в модуль **Q** идентификаторы из списка **L** подвергаются этому импорту.

Инициализация блока модуля состоит в следующем:

- исполняются — в порядке, зависящем от применяемой модели исполнения программ — описания из вступительной части блока модуля;

- при наличии нормальной части инициализирующего тела данного блока эта нормальная часть исполняется;

Исполнение списка неквалифицируемого или квалифицируемого экспорта некоторого локального модуля заключается в том, что все идентификаторы из данного списка подвергаются соответственно неквалифицируемому или квалифицируемому экспорту из данного локального модуля.

Финализация программного модуля выполняется следующим образом:

- *финализируются* все блоки статических модулей, инициализированных в ходе исполнения программного модуля; финализация блоков происходит в порядке, обратном порядку инициализации соответствующих модулей;

Финализация блока модуля состоит в исполнении нормальной части финализирующего тела этого блока, если таковое имеется.

Если модуль защищён, то перед инициализацией и перед финализацией блока этого модуля осуществляется подъём, а после инициализации и после финализации — опускание уровня защиты соответствующей сопрограммы (см 19).

14.8. Примеры на использование модулей

Пример 1 (схематический):

```
MODULE program;
  FROM M IMPORT sex, trans, invert, note, note1,
                label;
  VAR i,j,k: INTEGER; John: sex;
      comp1, comp2: BOOLEAN;
  MODULE loc1;
    EXPORT QUALIFIED p1,p2;
    PROCEDURE p1(): INTEGER;
    BEGIN RETURN 2*2
    END p1;
    PROCEDURE p2(): INTEGER;
    BEGIN RETURN 3*3
    END p2;
  BEGIN
  END loc1;
  MODULE loc2;
    IMPORT i,loc1;
  BEGIN i:=loc1.p1()
  END loc2;
  MODULE loc3;
    FROM loc1 IMPORT p2;
    IMPORT j;
    EXPORT a;
```

```
    VAR a: ARRAY [1..100] OF INTEGER;
  BEGIN j:=p2()
  END loc3;
  MODULE loc4;
    IMPORT a;
    VAR n: INTEGER;
  BEGIN FOR n:=1 TO 100 DO a[n]:=0 END
  END loc4;
BEGIN FOR k:=10 TO 100 BY 10 DO a[k]:=1 END;
      John:=male;
      trans (note,label); invert (note);
      comp1 := note=note1; comp2 := note=label
END program.

DEFINITION  MODULE  M;
  TYPE sex = (male, female); pair;
  PROCEDURE trans (r: pair; VAR s: pair);
  PROCEDURE invert(VAR r: pair);
  VAR note, note1, label: pair;
END M.

IMPLEMENTATION  MODULE  M;
  FROM N IMPORT address1, address2;
  TYPE pair =
         POINTER TO RECORD x,y: BOOLEAN END;
  PROCEDURE trans(r: pair; VAR s: pair);
  BEGIN s^.x:=r^.x; s^.y:=r^.y
  END trans;
```

```
    PROCEDURE invert(VAR r: pair);
      VAR z: BOOLEAN;
    BEGIN z:=r^.x; r^.x:=r^.y; r^.y:=z
    END invert;
BEGIN note:=address1; note1:=address1; label:=address2
END M.
```

Рассмотрим поочерёдно встречающиеся в программе объекты.

1) Идентификаторы i и j, описанные в программном модуле program, подвергаются простому импорту соответственно в локальные модули loc2 и loc3. Использование в этих локальных модулях идентификаторов i и j равносильно использованию тех же модулей в программном модуле; иными словами, каждый из этих идентификаторов обозначает в обоих модулях одну и ту же (свою) переменную.

2) Идентификатор p1, описанный в loc1, подвергается квалифицируемому экспорту в программный модуль, в силу чего процедура-функция, обозначаемая в loc1 этим идентификатором, обозначается в программном модуле квалифицируемым идентификатором loc1.p1. Локальный модуль loc1 описан в программном модуле и подвергается простому импорту в локальный модуль loc2. Поэтому та же процедура-функция обозначается в модуле loc2 тем же квалифицируемым идентификатором loc1.p1.

3) Идентификатор p2, описанный в loc1,

подвергается квалифицируемому экспорту в программный модуль, в силу чего процедура-функция, обозначаемая в loc1 этим идентификатором, обозначается в программном модуле квалифицируемым идентификатором loc1.p2. Простой идентификатор p2 подвергается неквалифицируемому импорту из модуля loc1 в модуль loc3, и поэтому ту же процедуру-функцию в модуле loc3 обозначает снова идентификатор p2.

4) Идентификатор a, описанный в локальном модуле loc3, подвергается неквалифицируемому экспорту из этого модуля в охватывающий его программный модуль, а затем простому импорту из программного модуля в локальный модуль loc4. Переменная-массив, обозначаемая в loc3 этим идентификатором, сохраняется в программном модуле и в модуле loc4 и обозначается там тем же простым идентификатором a.

5) В модуле определений M описаны идентификатор типа sex, идентификаторы процедур trans и invert, идентификаторы переменных note, note1 и label. Эти идентификаторы подвергаются квалифицируемому экспорту в модуль, непосредственно охватывающий модуль M, то есть во внешний модуль, а затем — неквалифицируемому импорту в программный модуль. В программном модуле соответствующие объекты обозначаются теми же простыми идентификаторами, которыми они

обозначались в модуле M. Вместе с идентификатором типа-перечисления sex неявно экспортируются, а затем импортируются идентификаторы male и female.

6) Из некоторого, пока явно не выписанного, модуля определений N в модуль реализации M импортируются идентификаторы address1 и address2. Тексты раздельных модулей N будут приведены в одном из примеров п. 16.3.

Рассмотрим далее ход исполнения программы.

а) Начинает исполняться программный модуль. Это означает, что начинает исполняться импортируемый в него модуль определений M, а также модуль реализации M. Это, в свою очередь, означает, что начинают исполняться модули N. Как будет ясно из примера к п. 16.3, исполнение этих модулей приведёт к созданию двух переменных, каждая из которых содержит в себе пару логических значений, к заполнению первой из этих пар значениями ***истина*** и ***ложь*** и к вычислению адресов этих переменных. Эти адреса (указатели) становятся соответственно значениями переменных address1 и address2.

б) После исполнения модулей N начинают исполняться операторы блока модуля реализации M. Переменные note и note1 скрытого типа pair получают в качестве своих значений указатели на переменную, значение которой есть запись с полями ***истина*** и ***ложь***. Переменная label того же скрытого типа pair получает в качестве своего значения указатель на

переменную, значение которой есть запись с неопределёнными пока полями. (Того, что note, note1 и label связаны именно с записями, пользователь может не знать. От разработчиков модулей M и N он знает лишь, что эти переменные указывают на "каким-то образом" хранимую пару логических значений, причём первая переменная хранит именно пару ***истина*** и ***ложь***, и что он с помощью процедуры trans может пересылать эти значения, а с помощью процедуры invert — менять их на противоположные.)

в) Затем исполняется локальный модуль loc2. Процедура-функция loc1.p1 исполняется и полученное значение 4 присваивается переменной i (из программного модуля).

г) Исполняется локальный модуль loc3. Переменная j получает аналогичным образом значение 9.

д) Исполняется локальный модуль loc4. Все 100 компонент переменной-массива a получают значение 0.

е) Начинает исполняться последовательность операторов из инициализирующего тела блока программного модуля. Первый оператор этой последовательности (цикл с шагом) присваивает каждой десятой компоненте переменной-массива a значение 1. Переменная John импортированного типа sex получает значение, обозначаемое идентификатором male. Разыменованная переменная, на которую указывает значение импортированной

переменной label, получает с помощью импортированной процедуры trans в качестве своего значения ту пару логических значений, на которые указывает значение импортированной переменной note. К переменной note применяется импортированная процедура invert, меняющая каждое значение соответствующей пары на противоположное. Переменная comp1 получает значение ***истина***, а переменная comp2 — значение ***ложь***, поскольку значения переменных note и note1 являются указателями на одну и ту же переменную, а значения переменных note и label — указателями на разные переменные.

В результате всего этого к концу исполнения рассматриваемой программы описанные в её программном модуле переменные i, j, k, John, comp1, comp2 и импортированные в программный модуль переменные note, note1, label и a будут иметь такие значения:

- i — 4
- j — 9
- k — (не определено)
- John — (значение, обозначаемое идентификатором male)
- note — (скрытая упорядоченная пара значений ***ложь*** и ***истина***)
- note1 — (скрытая упорядоченная пара значений ***ложь*** и ***истина***)
- label — (скрытая упорядоченная пара значений

истина и ***ложь***)

comp1 — ***истина***
comp2 — ***ложь***
a — 0 0 0 0 0 0 0 0 0 1
.....
0 0 0 0 0 0 0 0 0 1

Пример 2 (несколько более содержательный):

В примерах из пп. 5.7.1, 8.1, 9.1 шла речь о некотором "учреждении", "служащим" которого сопоставлены некие таблицы (массивы), задающие рабочее расписание этих служащих в течение недели. Опишем в модуле определений types типы week и time_table, взятые из п. 8.1. Будем считать, что в модуле определений persons описаны и заполнены для каждого служащего переменные-массивы типа time_table (в частности, переменные с весьма нам понятными идентификаторами Johnson, Peterson, Isidorson). Раздельные модули persons здесь подразумеваются, но не выписываются. В модулях определений procs опишем служебные процедуры onward и backward, сдвигающие сетку рабочих часов на один час вперёд или назад (в том числе и через границу суток и недели). Наконец, программный модуль alter будет осуществлять с помощью процедур onward и backward, импортированных из procs, желаемое для администрации изменение рабочей сетки тех или иных служащих.

(Разумеется, приводимая здесь программа не будет

иметь никакого практического значения до тех пор, пока в неё не будут включены средства, позволяющие выдавать результаты её исполнения — новые значения массивов — на внешние носители. Эти средства будут подробно разобраны далее, в 17.4, а пока нам придётся примириться с заведомой неполнотой этой программы.)

```
MODULE alter;
       IMPORT persons;
       FROM procs IMPORT onward, backward;
       VAR k: CARDINAL;
BEGIN  onward (persons.Johnson);
           backward (persons.Peterson);
           FOR k:=1 TO 3 DO onward (persons.Isidorson)
          END
END alter.

DEFINITION MODULE procs;
      FROM types IMPORT time_table;
      PROCEDURE onward (VAR x: time_table);
      PROCEDURE backward (VAR x: time_table);
END procs.

IMPLEMENTATION MODULE procs;
       FROM types IMPORT week, time_table;
       PROCEDURE next_day (x: week): week;
           VAR z: week;
       BEGIN CASE x OF
                  Sunday:       z:=Monday
```

```
            |Monday:      z:=Tuesday
            |Tuesday:     z:=Wednesday
            |Wednesday:   z:=Thursday
            |Thursday:    z:=Friday
            |Friday:      z:=Saturday
            END;
            RETURN z
END next_day;
PROCEDURE prev_day (x: week): week;
    VAR z: week;
BEGIN CASE x OF
             Monday:      z:=Sunday
            |Tuesday:     z:=Monday
            |Wednesday:   z:=Tuesday
            |Thursday:    z:=Wednesday
            |Friday:      z:=Thursday
            |Saturday:    z:=Friday
            END;
            RETURN z
END prev_day;
PROCEDURE onward (VAR x: time_table);
      VAR w: week; h: CARDINAL;
           carry: BOOLEAN;
BEGIN carry:=x[Saturday,23];
          FOR w:=Saturday TO Sunday BY –1
          DO FOR h:=23 TO 1 BY -1
              DO x[w,h]:=x[w,h-1]
              END;
              IF w = Sunday
```

```
                    THEN x[w,0]:=carry
                    ELSE x[w,0]:=x[prev_day(w),23]
                    END
                END
        END onward;
        PROCEDURE backward (VAR x: time_table);
            VAR w: week; h: CARDINAL;
                carry: BOOLEAN;
        BEGIN carry:=x[Sunday,0];
                FOR w:=Sunday TO Saturday
                DO FOR h:=0 TO 22
                    DO x[w,h]:=x[w,h+1]
                    END;
                    IF w = Saturday
                    THEN x[w,23]:=carry
                    ELSE x[w,23]:=x[next_day(w),0]
                    END
                END
        END backward;
BEGIN
END procs.


DEFINITION MODULE types;
    TYPE week = (Sunday, Monday, Tuesday,
                    Wednesday, Thursday, Friday,
                    Saturday);
            time_table =
                    ARRAY week, [0 .. 23] OF BOOLEAN;
END types.
```

```
IMPLEMENTATION MODULE types;
BEGIN
END types.
```

От громоздких процедур next_day и prev_day из модуля procs мы постараемся избавиться, используя стандартные процедуры INC и DEC (см. п. 15.1).

Пример 3 (на использование рекурсивного вызова процедур):

Мольеровский "мещанин во дворянстве" раздумывал над тем, какие перестановки слов можно сделать во фразе "Madame, vos beaux yeux me font mourir d'amour" ("Сударыня, ваши прекрасные глаза заставляют меня умирать от любви"). Приводимая ниже программа заполняет двумерный массив permuted всеми 120 осмысленными перестановками этой фразы.

```
MODULE permute;
   FROM source IMPORT n, m, A, W, T, a, words,
                                 lengths_of_words;
   FROM word IMPORT next_word;
   VAR permuted: T;
   PROCEDURE f(k: CARDINAL; model: A);
      VAR i,j: CARDINAL;
   BEGIN
```

```
        FOR i:=k TO n
        DO a[k]:=model[i];
             FOR j:=k TO i-1 DO a[j+1]:=model[j] END;
             FOR j:=i+1 TO n DO a[j]:=model[j] END;
             IF k=n
            THEN FOR j:=1 TO n
                    DO next_word(permuted,words[a[j]],j)
                    END
            ELSE f(k+1,a)
            END
        END
   END f;
BEGIN f(1,a)
END permute.


DEFINITION MODULE source;
   CONST n=5; m=15; fc=120;
   TYPE  A = ARRAY [1..n] OF CARDINAL;
           W = ARRAY [1..n],[1..m] OF CHAR;
           T = ARRAY [1..fc],[1..70] OF CHAR;
   VAR words: W; a,lengths_of_words: A;
END source.

IMPLEMENTATION MODULE source;
   VAR i: CARDINAL;
BEGIN FOR i:=1 TO n DO a[i]:=i END;
      words:=W{"MADAME ", "VOS BEAUX YEUX ",
                "ME FONT ", "MOURIR ",
```

```
                    “D'AMOUR "};
      lengths_of_words:=A{6,14,7,6,7}
END source.

DEFINITION MODULE word;
    FROM source IMPORT T;
    PROCEDURE next_word
      (VAR p: T; w: ARRAY OF CHAR; j: CARDINAL);
END word.

IMPLEMENTATION MODULE word;
   FROM source IMPORT n,m,fc,T,lengths_of_words,a;
   VAR start: BOOLEAN; I,J: CARDINAL;

PROCEDURE next_word
     (VAR p: T; w:  ARRAY OF CHAR; j: CARDINAL);
  VAR r: CARDINAL;
BEGIN IF start THEN I:=1; J:=1; start:=FALSE END;
          FOR r:=0 TO lengths_of_words[a[j]]
          DO p[I,J]:=w[r]; J:=J+1
          END;
          IF j=n THEN I:=I+1; J:=1 END
END next_word;
BEGIN start:=TRUE
END word.
```

Для того, чтобы применить данный алгоритм к какой-либо другой фразе, достаточно заменить значения констант n, m и fc в модуле определений source и изменить конструкторы множеств в правых

изменить конструкторы множеств в правых частях операторов присваивания для переменных words и lengths_of_words в соответствующем модуле реализации. (Константа fc есть, как легко понять, факториал числа n.)

(При наличии средств, упомянутых в примечании к предыдущему примеру, можно было бы выдавать очередные перестановки по мере их получения и не было бы необходимости ни в типе T и массиве permuted, ни в массиве lengths_of_words, ни вообще в раздельном модуле word.)

15. СТАНДАРТНЫЕ ПРОЦЕДУРЫ

Данный раздел знакомит с существующими в языке *стандартными* (предопределенными) *процедурами*, обозначаемыми стандартными идентификаторами процедур.

Стандартные процедуры, как будет видно в дальнейшем, могут не укладываться в рамки, определённые для процедур Модулы-2, составляемых программистом: стандартная процедура может иметь переменное число параметров; один и тот же параметр может принимать разные типы; то же относится к типу результата; фактическим параметром процедуры может служить не выражение, а представление типа.

15.1. Стандартные собственно процедуры

Процедуры INC и DEC

Вызовы процедуры INC и процедуры DEC должны иметь один или два фактических параметра. Первый параметр (соответствующий формальному параметру – переменной) должен быть переменной порядкового типа, а второй параметр — выражением целочисленного типа. Если объемлющий тип типа первого параметра не есть INTEGER, то значение второго параметра не может быть отрицательным числом.

Вызовы INC(**x**,**N**) и DEC(**x**, **N**) присваивают переменной **x** значение, являющееся соответственно **N**-м преемником или **N**-м предшественником текущего значения этой переменной относительно типа первого параметра.

При отсутствии среди значений данного типа такого преемника (предшественника) возбуждается исключительная ситуация. Вызовы INC(**x**) и DEC(**x**) равносильны соответственно вызовам INC(**x**,1) и DEC(**x**,1).

Примеры:

а) Предположим, что q есть переменная типа CHAR со значением "C", а i — переменная типа INTEGER со значением 1. В этом случае имеет место следующее:

Исполнение оператора процедуры	придаёт переменной q значение	придаёт переменной i значение
INC(q,0) INC(q,2) INC(q,-2) DEC(q,2) DEC(q,3)	"C" "E" *исключит. ситуация* "A" *исключит. ситуация*	
INC(i,2) INC(i,-2) DEC(i,2) DEC(i,-2)		3 –1 –1 3

б) Дадим новый вариант модуля определений procs из примера 2 в п. 14.8.

```
IMPLEMENTATION MODULE procs;
   FROM types IMPORT week, time_table;
   PROCEDURE onward (VAR x: time_table);
      VAR w,w1: week; h: CARDINAL;
            carry: BOOLEAN;
   BEGIN carry:=x[Saturday,23];
            FOR w:=Saturday TO Sunday BY –1
            DO FOR h:=23 TO 1 BY -1
                 DO x[w,h]:=x[w,h-1]
```

```
                END;
            IF w = Sunday
            THEN x[w,0]:=carry
            ELSE w1:=w; DEC(w1); x[w,0]:=x[w1,23]
            END
        END
    END onward;
    PROCEDURE backward (VAR x: time_table);
      VAR w,w1: week; h: CARDINAL;
          carry: BOOLEAN;
    BEGIN carry:=x[Sunday,0];
          FOR w:=Sunday TO Saturday
          DO FOR h:=0 TO 22
              DO x[w,h]:=x[w,h+1]
              END;
              IF w = Saturday
              THEN x[w,23]:=carry
              ELSE w1:=w; INC(w1);
                   x[w,23]:=x[w1,0]
              END
        END
    END backward;
BEGIN
END procs.
```

в) В следующих двух процедурах заголовки взяты из примеров в п. 5.6.1. Процедуры находят (циклически) следующий "цвет" в спектре. В собственно процедуре what_next_1 полученный "цвет" остаётся значением параметра – переменной

x, а в случае процедуры-функции what_next_2 этот "цвет" становится значением вызова функции.

```
PROCEDURE what_next_1 (VAR x: spectrum);
BEGIN IF x = violet THEN x:=red
        ELSE INC(x)
        END
END what_next_1;

PROCEDURE what_next_2 (x: spectrum): spectrum;
BEGIN IF x = violet THEN RETURN red
        ELSE INC(x); RETURN x
        END
END what_next_2;
```

Процедуры INCL и EXCL

Вызовы процедур INCL и EXCL должны иметь два фактических параметра. Первый из них (соответствующий формальному параметру – переменной) должен быть переменной, имеющей своим типом некоторый тип-множество, а второй — выражение, тип которого совместим по выражениям с объемлющим типом базового типа данного типа-множества. Если значение второго параметра не имеет своим типом указанный базовый тип, то при исполнении вызовов процедур INCL(**s**,**x**) и EXCL(**s**,**x**) возбуждается исключительная ситуация. В противном случае исполнение вызова процедуры INCL(**s**,**x**)

равносильно исполнению оператора присваивания **s** := **s** + **T**{**x**}, а исполнение вызова процедуры EXCL(**s**,**x**) равносильно исполнению оператора присваивания **s** := **s** – **T**{**x**}, где **T** есть тип параметра **s**.

Пример:

Пусть в программе имеются описания

```
TYPE set = SET OF [0..1];
VAR s: set;
VAR k: CARDINAL;
```

Тогда

При последовательном исполнении операторов	значение s будет последовательно совпадать со значениями конструктора
s:=set{1};	set{1}
k:=1; INCL(s,k);	set{1}
k:=0; EXCL(s,k);	set{0,1}
INCL(s,k);	set{0}
k:=1; EXCL(s,k);	

При значении k, отличном от 0 и 1, исполнение вызовов процедуры INCL(s,k) и EXCL(s,k) будет приводить к возбуждению исключительной ситуации.

Процедуры NEW и DISPOSE

Фактический параметр (соответствующий формальному параметру – переменной) вызова процедуры NEW(**P**) или вызова процедуры DISPOSE(**P**) должен быть переменной-указателем некоторого типа **T**, имеющего своим связанным типом некоторый тип **T1**. При указанных условиях исполнение этого вызова равносильно исполнению вызова ALLOCATE(**P**,SIZE(**T**)) или, соответственно, вызова DEALLOCATE(**P**, SIZE(**T**)), где

- ALLOCATE (DEALLOCATE) есть идентификатор процедуры, область видимости которого охватывает место вызова процедуры NEW(**P**) (DISPOSE(**P**));
- заголовок процедуры, обозначаемой данным идентификатором, согласован с заголовком процедуры ALLOCATE (DEALLOCATE), импортированной из стандартного модуля Storage (см. 17.1.1).

(О стандартной функции SIZE см. в 15.2.4.)

Если данная процедура ALLOCATE есть процедура, импортируемая из модуля Storage, то исполнение рассматриваемого вызова означает практически создание новой переменной типа **T**, отведение для неё нужного количества ячеек памяти и помещение в **P** начального адреса выделенной памяти, так что значением переменной **P** становится

указатель на эту новую переменную. В свою очередь, если данная процедура DEALLOCATE импортируется из модуля Storage, то исполнение рассматриваемого вызова означает уничтожение той переменной, указателем на которую является значение **P**, освобождение отведённых ячеек памяти и присвоение **P** значения ***псевдоимя***. В противных случаях рассматриваемый вызов может осуществлять некоторый другой способ отведения (освобождения) памяти или же какие-либо другие действия по усмотрению программиста.

Имеется также и другой вариант вызова каждой из этих процедур, а именно NEW(**P**,$\mathbf{x}_1$,...,$\mathbf{x}_n$) и DISPOSE(**P**,$\mathbf{x}_1$,...,$\mathbf{x}_n$), где $\mathbf{n} \geq 1$. Фактический параметр **P** в этом варианте должен быть переменной-указателем типа **T**, имеющего своим связанным типом некоторый тип-запись **T1**. Такой вызов равносилен вызову

ALLOCATE(**P**,SYSTEM.TSIZE(**T1**,$\mathbf{x}_1$,...,$\mathbf{x}_n$))

или, соответственно,

DEALLOCATE(**P**,SYSTEM.TSIZE(**T1**,$\mathbf{x}_1$,...,$\mathbf{x}_n$))

(см. 16.5).

Пример:

Рассмотрим таблицу размерами 100×100, каждый элемент которой либо пуст, либо содержит вектор, состоящий из 10 целочисленных значений. В интересах экономии памяти эту таблицу будем представлять как двумерный массив, каждый

элемент которого есть либо ***псевдоимя***, либо указатель на переменную – одномерный массив. Значение каждой компоненты любой такой переменной – одномерного массива будет зависеть от значений всех трёх задействованных индексов и определяться по некоторым правилам, скрытым от программиста в модуле реализации R.

а) Модуль определений R может иметь такой вид:

```
DEFINITION MODULE R;
PROCEDURE fill (x,y,z: INTEGER): INTEGER;
END R.
```

б) Предполагается, что программный модуль для нашей задачи содержит в своей вступительной части списки импорта

```
FROM R IMPORT fill;
FROM Storage IMPORT ALLOCATE,
                    DEALLOCATE;
```

и описания

```
TYPE T  =
       POINTER TO ARRAY [1..10] OF INTEGER;
VAR matrix: ARRAY [1..100], [1..100] OF T;
       i,j,k: INTEGER;
```

в) Пусть требуется, чтобы наша таблица содержала

пустые элементы выше своей главной диагонали и чтобы остальные элементы были заполнены с помощью импортируемой процедуры-функции fill. Это может быть осуществлено следующим оператором:

```
FOR i:=1 TO 100
DO FOR j:=1 TO i
     DO NEW(matrix[i,j]);
          FOR k:=1 TO 10
          DO matrix[i,j]^[k]:=fill(i,j,k)
          END
     END;
     FOR j:=i+1 TO 100 DO matrix[i,j]:=NIL END
END
```

г) Если в ходе дальнейшего выполнения программы нужно будет очистить элементы, находящиеся на главной диагонали, то это можно осуществить с помощью оператора

```
FOR i:=1 TO 100 DO DISPOSE(matrix[i,i]) END
```

д) Если же нужно очистить некоторый произвольный элемент с индексами i и j, то для этого придётся сперва выяснить с помощью операции сравнения отношений, не пуст ли он, и в случае отрицательного ответа применить процедуру DISPOSE

```
IF matrix[i,j] # NIL
THEN DISPOSE(matrix[i,j])
```

```
    END
```

(О дополнении, предоставляемом опцией M2EXTENSIONS системы XDS, см. Б.1.8.)

Процедура HALT

Вызов процедуры HALT не имеет фактических параметров. Если этот вызов исполняется на этапе инициализации программного модуля, то завершается исполнение этого этапа.

В противном случае завершается происходящая в данный момент финализация блока модуля.

(О новых стандартных процедурах, предоставляемых опцией M2EXTENSIONS системы XDS, см. Б.1.8)

15.2. Стандартные процедуры-функции

15.2.1. Функции для преобразования типов

Под *преобразованием типов* понимается превращение значения одного типа в определенное значение некоторого (как правило, другого) типа в соответствии с некоторыми, заданными языком, соотношениями между этими типами. Функция VAL осуществляет такое преобразование в общем случае. Некоторые частные случаи преобразований можно производить с помощью остальных приводимых ниже функций.

Функция VAL

Вызов функции VAL имеет два фактических параметра. Первый параметр либо является представлением некоторого порядкового типа, и в этом случае типом вызова функции служит объемлющий тип этого порядкового типа, либо является представлением некоторого вещественного числового типа, и в этом случае типом вызова служит сам этот вещественный тип. Второй параметр является выражением некоторого порядкового или некоторого вещественного числового типа. Ниже приводятся допустимые комбинации фактических параметров и соответствующие этим комбинациям значения вызова VAL(**T**,**x**):

а) **T** есть представление некоторого вещественного числового типа, **x** есть выражение некоторого вещественного числового типа или вложенного типа некоторого целочисленного типа. Значение вызова VAL(**T**,**x**) есть то значение типа, представляемого параметром **T**, которое является определяемым реализацией приближением к значению выражения **x**.

б) **T** есть представление вложенного типа некоторого целочисленного типа, **x** есть выражение вложенного типа некоторого целочисленного типа или выражение некоторого вещественного числового типа. Значение вызова VAL(**T**,**x**) есть целая часть значения выражения **x**. Если последняя не является значением типа **T**, то возбуждается исключительная ситуация.

в) Как тип, представляемый параметром **T**, так и тип выражения **x** являются порядковыми типами, и

либо в точности один из них есть вложенный тип некоторого целочисленного типа, либо объемлющие типы обоих этих типов идентичны. Значение вызова VAL(**T**,**x**) есть то значение типа, представляемого параметром **T**, порядковый номер которого в определяемом этим типом множестве равен порядковому номеру значения выражения **x** в множестве, определяемом типом данного выражения. Если значение выражения **x** является численным значением, меньшим нуля, или если порядковому номеру значения выражения **x** не соответствует никакое значение типа **T**, то возбуждается исключительная ситуация.

Примеры (используется пример из 5.1.4):

Вызов функции	Значение вызова (возможное представление)	Тип вызова
VAL(REAL,5.2) VAL(REAL,-5)	5.2 -5.0	REAL REAL
VAL(INTEGER,-5) VAL(INTEGER,-5.2) VAL(CARDINAL,5) VAL(CARDINAL,5.2) VAL(CARDINAL,-5)	-5 -5 5 5 *исключит. ситуация*	INTEGER INTEGER CARDINAL CARDINAL
VAL(CARDINAL,FALSE) VAL(INTEGER,Wednesday) VAL(CHAR,8) VAL(CHAR,"8") VAL(BOOLEAN,1) VAL(week_days,3) VAL(week_days,Wednesday) VAL(week_days,Sunday)	0 3 10C "8" ***истина*** Wednes-day Wednes-day *исключит. ситуация*	CARDINAL INTEGER CHAR CHAR BOOLEAN week week

Функции FLOAT и LFLOAT

Вызов функции FLOAT и вызов функции LFLOAT имеют один фактический параметр, являющийся выражением некоторого вещественного числового типа или некоторого целочисленного типа. Типы вызова этих функций суть соответственно REAL и LONGREAL.

Значение вызова FLOAT(**x**) и, соответственно, значение вызова LFLOAT(**x**) есть значение типа REAL (LONGREAL), являющееся определяемым реализацией приближением к значению выражения **x**.

Вызов FLOAT(**x**) эквивалентен вызову

VAL(REAL,**x**) ,

а вызов LFLOAT(**x**) — вызову

VAL(LONGREAL,**x**) .

Функция INT

Вызов функции INT имеет один фактический параметр, являющийся выражением некоторого порядкового типа или некоторого вещественного числового типа. Тип вызова функции — INTEGER.

Значением вызова INT(**x**), в котором **x** есть выражение порядкового типа, является порядковый

номер значения выражения **x** в множестве, определяемом этим типом. Значением вызова INT(**x**), в котором **x** есть значение вещественного числового типа является целая часть значения выражения **x**.

Вызов INT(**x**) эквивалентен вызову

VAL(INTEGER,**x**) .

Функция TRUNC

Вызов функции TRUNC имеет один фактический параметр, являющийся выражением некоторого вещественного числового типа. Тип вызова функции — CARDINAL.

Значение вызова TRUNC(**x**) есть целая часть значения выражения **x**. Если это значение отрицательно, то возбуждается исключительная ситуация.

Вызов TRUNC(**x**) эквивалентен вызову

VAL(CARDINAL,**x**) .

Функция ORD

Вызов функции ORD имеет один фактический параметр, являющийся выражением некоторого порядкового типа. Тип вызова функции — CARDINAL.

Значением вызова ORD(**x**) является число, являющееся порядковым номером значения

выражения **x** в множестве, определяемом соответствующим порядковым типом. Если это число отрицательно, то возбуждается исключительная ситуация.

Вызов ORD(**x**) эквивалентен вызову

VAL(CARDINAL,**x**) .

Пример:

Если **A** есть выражение типа **T**, то INC(**A**) и DEC(**A**) равносильны соответственно выражениям VAL(**T**,ORD(**A**)+1) и VAL(**T**,ORD(**A**)-1).

Функция CHR

Вызов функции CHR должен иметь один фактический параметр, являющийся выражением некоторого целочисленного типа. Тип вызова функции — литерный тип.

Значением вызова CHR(**x**) является литерное значение, имеющее в множестве, определяемом литерным типом, порядковый номер, равный значению выражения **x**.

Вызов CHR(**x**) эквивалентен вызову

VAL(CHAR,**x**) .

15.2.2. Функции над числовыми значениями

Функция ABS

Вызов функции ABS имеет один фактический параметр, являющийся выражением вещественного числового типа или целочисленного типа, отличного от типа CARDINAL. Тип вызова функции совпадает с типом параметра.

Значением вызова ABS(**x**) является абсолютная величина значения параметра **x**.

Функция ODD

Вызов функции ODD имеет один фактический параметр, являющийся выражением некоторого целочисленного типа. Тип вызова функции — логический. Значение вызова ODD(**x**) есть ***истина***, если значение выражения **x** нечетно, и ***ложь*** в противном случае.

Функции RE и IM

Вызов функции RE и вызов функции IM имеют один фактический параметр, являющийся выражением типа COMPL, типа LONGCOMPL либо общего комплексного числового типа. Типом вызова функции является соответственно тип REAL, тип LONGREAL либо общий вещественный числовой тип. Значениями вызова RE(**z**) и вызова IM(**z**) являются те значения соответствующего вещественного

числового типа, которые служат определяемыми реализацией приближениями к вещественной (соответственно мнимой) части параметра **z**.

Функция CMPLX

Вызов функции CMPLX имеет два фактических параметра, являющихся выражениями некоторых вещественных числовых типов. Если оба выражения имеют тип REAL или если одно из них имеет тип REAL, а другое — общий вещественный числовой тип, то типом вызова является тип COMPLEX. Если оба выражения имеют тип LONGREAL или если одно из них имеет тип LONGREAL, а другое — общий вещественный числовой тип, то типом вызова является тип LONGCOMPLEX. Если оба выражения имеют общий вещественный числовой тип, то типом вызова является общий комплексный числовой тип. (Остальные комбинации типов параметров недопустимы.)

Значением вызова CMPLX(**x**,**y**) является определяемое реализацией приближение такого комплексного числа, действительная часть которого есть значение параметра **x**, а комплексная часть — значение параметра **y**.

15.2.3. Функции над строковыми и литерными значениями

Функция LENGTH

Вызов функции LENGTH имеет один фактический параметр, являющийся выражением строкового типа. Тип вызова функции — тип CARDINAL.

Значение вызова LENGTH(**s**) есть длина строки, абстрагированной из значения выражения **s**.

Функция CAP

Вызов функции CAP имеет один фактический параметр, являющийся выражением литерного типа. Тип вызова функции — тип CHAR.

Если значением параметра **c** является литерное значение, соответствующее некоторой, **n**-й в алфавите, строчной букве, то значением вызова CAP(**c**) является литерное значение, соответствующее **n**-й в алфавите заглавной букве; в противном случае значением вызова является само значение параметра **c**.

15.2.4. Функции, определяющие размеры объектов

Функция HIGH

Вызов функции HIGH может содержаться лишь в блоке процедуры и имеет один фактический параметр, являющийся либо некоторым формальным параметром **A** соответствующей процедуры, характеризующимся признаками открытого массива — размерностью **n** и некоторым типом компоненты, либо индексированным обозначением, в котором

обозначение массива есть **A**, а число индексных выражений меньше, чем **n**. Тип вызова функции — CARDINAL.

Значение вызова HIGH(**x**) вычисляется следующим образом.

- Если **x** есть **A**, то рассматривается сама переменная-массив **A**.
- Если **x** есть индексированное обозначение $\mathbf{A}[\mathbf{i}_1, ..., \mathbf{i}_\mathbf{m}]$, то рассматривается переменная-массив — индексированное обозначение с **m** индексами **A**[0, ..., 0].
- Верхний порядковый номер типа индекса типа рассматриваемой переменной-массива (иными словами, уменьшенное на единицу число значений этого типа-индекса) становится искомым значением вызова.

Примеры:

а) (См. пример из п. 5.6.1)

```
PROCEDURE init_matrix
        (VAR matrix: ARRAY OF ARRAY OF REAL);
    VAR i,j: CARDINAL;
BEGIN FOR i:=0 TO HIGH(matrix)
          DO FOR j:=0 TO HIGH(matrix[10])
               DO matrix[i,j]:=0.0
               END
          END
END init_matrix;
```

Процедура придаёт всем компонентам переменной — двумерного массива, являющейся фактическим параметром, нулевые значения.

б) В области видимости описаний

```
TYPE ABC = ("A","B","C");
        U = RECORD c0,c1,c2: CARDINAL END;
VAR a:
      ARRAY [-7 .. -2], BOOLEAN, ABC  OF REAL;
PROCEDURE p
  (f: ARRAY OF ARRAY OF ARRAY OF REAL): U;
BEGIN RETURN
U{HIGH(f),HIGH(f[0]),HIGH(f[0,0])}
END p;
```

значением вызова функции p(a) будет запись с полями 5, 1, 2.

Функции MIN и MAX

Вызов функции MIN и вызов функции MAX имеют один фактический параметр, являющийся идентификатором, представляющим некоторый порядковый тип или некоторый вещественный числовой тип. Если тип, представляемый параметром, есть вещественный числовой тип, либо вложенный тип целочисленного типа, то типом вызова функции является соответственно общий вещественный числовой тип или общий целочисленный тип; в противном случае тип вызова функции есть

объемлющий тип типа, представляемого параметром.

Значениями вызова MIN(**T**) и MAX(**T**) являются соответственно наименьшее и наибольшее значения типа, представляемого параметром **T**.

(Значения вызовов этих функций для стандартных типов в реализации XDS см. в Б.6.)

Примеры:

В области видимости описаний

```
TYPE months = (Jan,Feb,Mar,Apr,May,Jun,
               Jul,Aug,Sep,Oct,Nov,Dec);
     summer = months[Jun..Aug];
     Rome = [-753..476];
```

значения вызовов функций MIN и MAX для приведённых выше параметров будут следующими:

Параметр	MIN	MAX
month	Jan	Dec
summer	Jun	Aug
Rome	-753	476

Функция SIZE

Вызов функции SIZE имеет один фактический параметр, являющийся либо идентификатором типа, либо обозначением — идентификатором. В последнем случае указанный идентификатор не может быть формальным параметром, характеризующимся признаками открытого массива. Тип вызова функции

— общий целочисленный тип.

Значение вызова SIZE(**T**), где **T** есть идентификатор типа, должно быть числом ячеек памяти, отводимых для переменной представляемого им типа. Значение вызова SIZE(**v**), где **v** есть обозначение типа, представляемого идентификатором **T**, совпадает со значением вызова SIZE(**T**).

(Значения вызовов функции SIZE для стандартных типов в реализации XDS см. в Б.6.)

(О новых стандартных процедурах-функциях, предоставляемых опцией M2EXTENSIONS системы XDS, см. Б.1.9)

16. СИСТЕМНЫЕ МОДУЛИ. СИСТЕМНЫЙ МОДУЛЬ SYSTEM

С точки зрения пользователя, *системные модули* могут рассматриваться как частные случаи стандартных модулей, хотя и обладающие определёнными особенностями. В частности, процедура, описанная в системном модуле (формально говоря, в соответствующем модуле определений), может подобно стандартной процедуре иметь переменное число параметров и разные типы одного и того же формального параметра и результата. Кроме того, использование системного модуля должно подчиняться следующим ограничениям:

- процедура, описанная в системном модуле, не может быть присвоена никакой переменной-процедуре;
- идентификатор типа, описанный в определении скрытого типа, содержащемся в системном модуле, не может быть использован в качестве представления типа в описании типа, содержащемся во вступительной части блока модуля реализации и соответствующем описанию скрытого типа из модуля определений.

В этом разделе мы рассмотрим системный модуль SYSTEM, задающий средства низкого уровня для доступа к машинной памяти, выделяемой для переменных, и к адресам, служащим для доступа к этой памяти, а также для выполнения адресной арифметики и для манипулирования с битовыми представлениями значений.

В следующих пунктах раздела приводятся описанные для этих целей в модуле SYSTEM типы, константы и процедуры.

16.1. Системные типы памяти и адресный тип

Системными типами для работы с памятью (*системными типами памяти*) являются типы LOC, BYTE и WORD.

Значением типа LOC является неинтерпретируемое битовое содержимое *ячейки*, то есть наименьшей

адресуемой единицы памяти. Число битов в ячейке определяется значением константы BITSPERLOC.

Значение типа BYTE есть неинтерпретируемое битовое содержимое *байта*, рассматриваемого как сплошной участок памяти, содержащий LOCSPERBYTE ячеек. Тип BYTE описан как массив с типом компоненты LOC и типом индекса [0 .. LOCSPERBYTE - 1]. Конкретная реализация может — с учётом особенностей архитектуры соответствующей машины — отказаться от заведения типа BYTE и константы LOCSPERBYTE.

Значение типа WORD есть неинтерпретируемое битовое содержимое *слова*, рассматриваемого как сплошной участок памяти, содержащий LOCSPERWORD ячеек. Тип WORD описан как массив с типом компоненты LOC и типом индекса [0 .. LOCSPERWORD-1].

(О типе BYTE и константе LOCSPERBYTE в реализации XDS см. Б.4, а значения остальных упомянутых здесь констант см. в Б.6. О дополнении, предоставляемом опцией M2EXTENSIONS системы XDS, см. Б.1.10.)

Операции сравнения для основных типов системной памяти (ср. 12.7)

Операция сравнения для основных типов системной памяти выполняется над парой операндов, типы которых суть идентичные основные типы системной памяти. При данных типах операндов

темной памяти. При данных типах операндов символ "=" представляет операцию "совпадает с", а символ неравенства — операцию "не совпадает с". Результирующее значение операции "совпадает с" есть ***истина***, а операции "не совпадает с" есть ***ложь*** в том и только том случае, когда значения её операндов одинаковые.

Тип ADDRESS

Адресный тип ADDRESS описан как POINTER TO LOC и значение этого типа есть логический адрес соответствующей ячейки в памяти машины.

16.1.1. Системная совместимость

Системно совместимыми с некоторым типом могут быть либо тип, либо признаки открытого массива.Тип **F** системно совместим с типом **T**, если **F** есть системный тип памяти и SIZE(**T**)=SIZE(**F**).

Признаки открытого массива системно совместимы с типом **T**, если они состоят из размерности, равной 1, и типа компоненты — системного типа памяти **F** и если значение SIZE(**T**) есть целое кратное от значения SIZE(**F**).

16.1.2. Системное наложение

Если **V** есть переменная системного типа памяти **S** или переменная – многомерный массив с типом компоненты **S**, а **E** есть выражение, то *системное*

наложение **E** на **V** означает некоторое (определяемое реализацией) размещение битового представления значения выражения **E** в ячейках, отведённых для **V**, или, соответственно, в ячейках, отведённых для последовательных компонент **V**.

16.2. Функции адресной арифметики

Функции ADDADR и SUBADR

Вызов функции ADDADR и вызов функции SUBADR имеют два фактических параметра. Первый параметр является выражением, тип которого есть некоторый тип-указатель (в частности, адресный тип), а второй параметр является выражением типа CARDINAL или неотрицательной целочисленной константой. Тип вызова функций есть адресный тип.

Значением вызова ADDADR(**addr**,**offset**) и, соответственно, значением вызова SUBADR(**addr**,**offset**) является логический адрес, получаемый в результате определяемого реализацией прибавления значения выражения **offset** к значению выражения **addr** (или, соответственно, вычитания значения выражения **offset** от значения выражения **addr**). Если значение выражения **addr** есть ***псевдоимя***, то возбуждается исключительная ситуация.

Функция DIFADR

Вызов функции DIFADR имеет два фактических параметра, являющихся выражениями, имеющими

своими типами некоторые типы-указатели. Тип вызова функции — INTEGER.

Значением вызова DIFADR(**addr1**,**addr2**) должно быть целое число, получаемое в результате определяемого реализацией вычитания значения выражения **addr2** из значения выражения **addr1**. Если значение хотя бы одного из параметров есть ***псевдоимя***, то возбуждается исключительная ситуация.

Свойства функций для адресной арифметики

Функции для адресной арифметики должны быть такими, чтобы были справедливы следующие соотношения (здесь тип переменной **adr** есть адресный тип, а тип переменной **n** есть стандартный тип CARDINAL):

SUBADR(ADDADR(**adr**,**n**), **n**) = **adr**
ADDADR(SUBADR(**adr**,**n**), **n**) = **adr**
DIFADR(ADDADR(**adr**,**n**), **adr**) = **n**
DIFADR(**adr**,SUBADR(**adr**,**n**)) = **n**
ADDADR(**adr**, 0) = **adr**

16.3. Функции построения и определения адресов

Функция MAKEADR

Вызов функции MAKEADR имеет один фактический параметр, являющийся выражением типа CARDINAL или неотрицательной целочисленной константой. Тип вызова функции есть адресный тип.

Значением вызова MAKEADR(**val**) является

логический адрес, полученный в результате определяемого реализацией преобразования, примененного к **val**.

Значение, построенное с помощью функции MAKEADR из константного параметра, может быть использовано в описании переменных (см. 7) для фиксирования предназначенного той или иной переменной адреса памяти.

Функция ADR

Вызов функции ADR имеет один фактический параметр, являющийся обозначением переменной. Тип вызова функции есть адресный тип. Значением вызова ADR(**v**) должен быть указатель на переменную **v**.

Примеры:

а) Ниже приводятся раздельные модули N для примера 1 из п. 14.8.

```
DEFINITION MODULE N;
   IMPORT SYSTEM;
   VAR address1, address2: SYSTEM.ADDRESS;
END N.


IMPLEMENTATION MODULE N;
   IMPORT SYSTEM;
   FROM Storage IMPORT ALLOCATE;
   VAR inner1, inner2:
      POINTER TO RECORD a, b: BOOLEAN END;
```

```
BEGIN NEW (inner1);
        address1:=SYSTEM.ADR(inner1^);
        inner1^.a:=TRUE; inner1^.b:=FALSE;
        NEW (inner2);
        address2:=SYSTEM.ADR(inner2^)
END N.
```

б) Применим адресные типы для работы со "списками". Создадим и будем наращивать некоторый список, "элементы" которого суть вещественные значения. Каждый очередной элемент будет порождаться функцией elem, а процесс наращивания списка будет продолжаться, пока функция on будет выдавать значение ***истина***. Эти две функции импортируются из модуля TAKE.

```
DEFINITION MODULE TAKE;
   PROCEDURE on(): BOOLEAN;
   PROCEDURE elem(): REAL;
END TAKE.
```

В программном модуле для нашей задачи будем иметь списки импорта

```
IMPORT SYSTEM;
FROM Storage IMPORT ALLOCATE;
FROM TAKE IMPORT elem, on;
```

и описания

```
TYPE P = POINTER TO LIST;
TYPE LIST = POINTER TO
                    RECORD element: REAL; next: LIST
                    END;
VAR list: LIST; pntr: P;
```

Требуемую задачу выполняют операторы

```
pntr:=SYSTEM.ADR(list);
WHILE on()
DO NEW(pntr^); pntr^ ^.element:=elem();
      pntr^ ^.next:=NIL;
       pntr:=SYSTEM.ADR(pntr^^.next)
END;
```

в) (См. Пример 1 из 5.8.5)

Пусть имеются два списка, отличающихся от списка из предыдущего примера тем, что элементы их имеют тип POINTER TO country. Это означает наличие таких, например, описаний:

```
TYPE PP = POINTER TO LLIST;
TYPE
   LLIST = POINTER TO
              RECORD element: POINTER TO country;
                        next: LLIST
              END;
VAR list1,list2: LLIST; pntr1,pntr2: PP;
```

Ставим задачу объединить эти два списка в один, но так, чтобы возможные указатели на одну и ту же переменную типа country, имеющиеся в разных списках, сохранились лишь в одном экземпляре.

сках, сохранились лишь в одном экземпляре. Заводим с этой целью список импорта

```
IMPORT SYSTEM;
```

и исполняем операторы

```
pntr1:=SYSTEM.ADR(list1);
WHILE pntr1^ # NIL
DO pntr2:=SYSTEM.ADR(list2);
    WHILE pntr2^ # NIL
    DO IF pntr1^ ^.element = pntr2^ ^.element
        THEN pntr2^:=pntr2^ ^.next
        END;
        IF pntr2^ # NIL
        THEN pntr2:=SYSTEM.ADR(pntr2^ ^.next)
        END
    END;
    pntr1:=SYSTEM.ADR(pntr1^^.next)
END;
pntr1^:=list2; list2:=NIL;
```

16.4. Функции для преобразования битовых представлений множеств

Битовое представление множества **S**, являющегося значением некоторого типа-множества, есть последовательность битов, расположенных в примыкающих друг к другу ячейках памяти и перенумерованных порядковыми номерами — начиная с некоторого **A** и кончая некоторым **Z** —

элементов множества, определяемого базовым типом данного типа-множества. (Эта последовательность битов может и не занимать полностью целое число ячеек.) Бит с номером **n** имеет значение 1, если **n** является порядковым номером некоторого элемента множества **S**, и имеет значение 0, если **n** не является порядковым номером элемента множества **S**.

Положительный (соответственно отрицательный) *единичный сдвиг* битового представления множества есть такое преобразование этого представления, при котором каждый **n**-й его бит, кроме **A**-го (соответственно **Z**-го), получает значение, которое имел (**n** – 1)-й (соответственно (**n** + 1)-й) бит, а **A**-й (соответственно **Z**-й) бит получает значение 0.

Положительный (соответственно отрицательный) *единичный циклический сдвиг* битового представления множества есть такое преобразование этого представления, при котором каждый **n**-й его бит, кроме **A**-го (соответственно **Z**-го), получает значение, которое имел (**n** – 1)-й (соответственно (**n** + 1)-й) бит, а **A**-й (соответственно **Z**-й) бит получает значение, которое имел **Z**-й (соответственно **A**-й) бит.

Функции SHIFT и ROTATE

Вызов функции SHIFT и вызов функции ROTATE имеют два фактических параметра. Первый параметр является выражением, имеющим своим типом некоторый тип-множество, а второй параметр —

выражением типа INTEGER или общего целочисленного типа. Тип вызова функции совпадает с типом первого параметра. Реализация может накладывать ограничения на типы, допустимые в качестве первого параметра, но в любом случае должны восприниматься значения типа BITSET.

Значение вызова SHIFT(**val**, **num**) есть множество, битовое представление которого получено из битового представления значения выражения **val** **n**-кратным применением положительного единичного сдвига, если значение выражения **num** неотрицательно, или отрицательного единичного сдвига, если это значение отрицательно, где **n** есть абсолютная величина значения выражения **num**.

Значение вызова ROTATE(**val**, **num**) есть множество, битовое представление которого получено из битового представления значения выражения **val** **n**-кратным применением положительного единичного циклического сдвига, если значение выражения **num** неотрицательно, или отрицательного единичного циклического сдвига, если это значение отрицательно, где **n** есть абсолютная величина значения выражения **num**.

Пример (см. пример из п. 5.1.3):

Пусть при наличии описаний

```
TYPE setweek = SET OF week;
VAR N: setweek;
```

переменной N присвоено значение конструктора множества

week{Monday,Wednesday,Saturday} .

Тогда каждое из следующих выражений будет иметь значение ***истина***:

```
SHIFT(N,1) = setweek{Tuesday,Thursday}
SHIFT(N,-3) = setweek{Sunday,Wednesday}
ROTATE(N,1) = setweek{Sunday,Tuesday,Thursday}
ROTATE(N,-3) = setweek{Sunday,Wednesday,
                       Friday}
ROTATE(N,7) = N
```

Всё, сказанное в настоящем разделе и относящееся к типам-множествам, относится также к типам-упакованным-множествам.

16.5. Функция для определения размера значений данного типа

Функция TSIZE

Фактический параметр вызова функции TSIZE(**t**) должен быть идентификатором, представляющим некоторый тип, а значение этого вызова есть (максимальное) число ячеек памяти, отводимых для значений данного типа. Рассматриваемый вызов равносилен, таким образом, вызову функции SIZE(**t**) и тип его есть общий целочисленный тип.

Имеется также и другой вариант вызова этой функции, а именно TSIZE(**t**,$\mathbf{x}_1$,...,$\mathbf{x}_n$), где $\mathbf{n} \geq 1$. Фактический параметр **t** в этом варианте должен быть идентификатором, представляющим некоторый тип-запись, характеризуемый списком характеристик, содержащим характеристики признаков. Значение рассматриваемого вызова есть число ячеек памяти, используемых таким значением данного типа, у которого последовательные поля признаков суть значения второго и дальнейших фактических параметров вызова.

16.6. Функция для перевода типов

Перевод типов есть интерпретация битового представления значения одного типа как значения некоторого другого типа.

Функция CAST

Вызов функции CAST имеет два фактических параметра. Первый параметр есть идентификатор, представляющий некоторый тип. Второй параметр есть выражение, не являющееся изображением числа. Тип вызова функции есть тип, представляемый первым параметром.

Значение вызова CAST(**Type**,**val**) есть преобразование значения выражения **val** к типу, представляемому идентификатором **Type**, осуществляемое в соответствии со следующими правилами, в которых **N1** обозначает число ячеек,

занимаемых значением выражения **val**, а **N2** — число ячеек, используемых переменной типа, представляемого идентификатором **Type**:

- если **N1** ≥ **N2**, то битовое представление значения вызова совпадает с первыми **N2** ячейками битового представления значения выражения **val**;
- иначе первые **N1** ячеек битового представления значения вызова совпадают с битовым представлением значения выражения **val**, а содержимое остальных ячеек неопределено.

17. СТАНДАРТНЫЕ МОДУЛИ

В настоящем разделе будут приведены стандартные модули, применяемые в усечённом варианте Модулы-2.

Ряд других стандартных модулей языка будет приведён в разделе 22.

17.1. Стандартные модули для базовых операций

Средства, предоставляемые рассматриваемыми здесь модулями, относятся к числу средств низкого уровня, то есть базовых средств языка.

17.1.1. Стандартный модуль Storage

Данный модуль даёт средства для создания и уничтожения переменных в ходе исполнения

программы.

В модуле описаний Storage имеются следующие определения процедур (здесь и во многих случаях в дальнейшем после определения процедуры даются в скобках пояснения к ней):

```
PROCEDURE ALLOCATE
                (VAR addr: SYSTEM.ADDRESS;
                 amount: CARDINAL);
```

(При исполнении данной процедуры выделяется участок свободной, то есть не занятой какой-либо переменной, памяти, размер которой в ячейках равен значению параметра amount, и адрес первой из этих ячеек присваивается переменной addr. Если выделить такой участок невозможно, то переменной addr присваивается значение ***псевдоимя***.)

```
PROCEDURE DEALLOCATE
                (VAR addr: SYSTEM.ADDRESS;
                 amount: CARDINAL);
```

(При исполнении данной процедуры освобождается участок памяти, у которого адрес первой ячейки есть значение переменной addr, а размер в ячейках равен значению параметра amount, при условии, что этот участок был ранее выделен процедурой ALLOCATE; после этого переменной addr присваивается значение ***псевдоимя***. Если участок не был выделен, то возбуждается исключительная ситуация.)

Кроме того, в модуле имеются описания (определения) для идентификаторов StorageExceptions, IsStorageException и StorageException, о чём речь будет идти в п. 18.2.

17.1.2. Стандартные модули LowReal и LowLong

Модули LowReal и LowLong предоставляют базовые средства для операций над вещественными числовыми значениями стандартных типов REAL и LONGREAL. Идентификаторы, описываемые в данных двух модулях, попарно лексически совпадают, так что пояснения к ним будут даваться одновременно, причём соответствующий (рассматриваемый) модуль будет обозначаться как **L**, а соответствующий стандартный тип — как **R**.

С каждым типом **R** реализация связывает два — связанных с определением диапазона значений данного типа — целых числа: *основание* **r** и *разрядность* **p**. Считается, что любое значение **x** переменной типа **R** имеет **r**-ичное *нормированное представление* $\mathbf{s} \times \mathbf{m} \times \mathbf{r}^{\mathbf{e}}$, где целочисленное значение **e** и вещественные числовые значения **m** и **s** суть соответственно *порядок*, *мантисса* и *знак* (+1.0 или –1.0) числа **x**, а мантисса, в свою очередь, представляется **r**-ичной дробью

$$0.a_1\ a_2\ \ldots\ a_{\mathbf{p}}\ ,$$

причём $0 \le a_{\mathbf{i}} < \mathbf{r}$ и либо все $a_{\mathbf{i}}$ равны нулю, либо a_1 не

равно нулю.

Для удобства дальнейшего изложения будем через c_i ($\mathbf{i} \leq \mathbf{p}$) обозначать значение выражения $\mathbf{s} \times \mathbf{m'} \times \mathbf{r^e}$, где мантисса **m'** представляется дробью

$$0.0 \ldots 0\ a_i\ 0 \ldots 0.$$

Если $\mathbf{i} > \mathbf{p}$, то c_i равно нулю.

а) Константы

В модулях определений LowReal и LowLong имеется ряд описаний констант с определяемыми реализацией константными значениями. Ниже приводятся идентификаторы соответствующих констант и указывается смысл значений каждой из них.

аа) Целочисленные константы

radix — основание
places — разрядность
expoMin — наименьшее значение порядка
expoMax — наибольшее значение порядка
nModes — число элементов множества, определяемого типом **L**.Modes (см. ниже)

аб) Вещественные числовые константы

large — наибольшее значение типа **R**
small — некоторое малое положительное значение типа **R** (со значением порядка, равным

expoMin)

ав) Логические константы

Значение логической константы есть ***истина*** в том и только том случае, если выполняется указываемое условие.

IEEE — представление с плавающей точкой вещественных чисел типа **R** соответствует требованиям стандарта IEEE754 или стандарта IEEE854

ISO — представление вещественных чисел типа **R** соответствует требованиям стандарта ISO / IEC 10967-1

rounds — каждая операция над значениями типа **R** производит результат, ближайший к математическому результату этой операции

gUnderflow — имеются значения типа **R**, расположенные между 0.0 и small

extend — реализация может допускатью использование вещественных чисел, больших по абсолютной величине чем MAX(**R**), в качестве значений типа **R**, но с тем, чтобы они не могли быть значениями выражений из операторов присваивания или

значениями фактических параметров

exception — каждая операция, пытающаяся выработать значение, находящееся вне диапазона чисел типа **R**, возбуждает исключительную ситуацию (см. 18)

(Значения констант типов LowReal и LongReal для реализации XDS см. в Б.7.)

б) Тип Modes

В модулях определений LowReal и LowLong имеется описание типамножества Modes, базовый тип которого есть тип-отрезок стандартного типа CARDINAL с границами 0 и nModes – 1.

Реализация определяет некоторые "особенности" выполнения операций над значениями типа **R** и нумеруют эти особенности некоторыми целыми числами из множества, определяемого базовым типом типа Modes. (К числу этих особенностей могут, например, относиться возбуждение исключительных ситуаций в случае определённых событий, управление округлением и т.д.) В каждый данный момент исполнения программы каждая такая особенность может быть задействована или не задействована. Начальное множество задействованных особенностей определяется реализацией.

в) Процедуры

В модулях определений LowReal и LowLong

имеются следующие определения процедур (в пояснениях исходим из того, что значение параметра x имеет нормированное представление, указанное в начале этого пункта):

PROCEDURE exponent (x: **R**): INTEGER;

(Значение вызова этой функции есть **e**. Если значение параметра x равно нулю, то имеет место исключительная ситуация.)

PROCEDURE fraction (x: **R**): **R**;

(Значение вызова этой функции есть **s** × **m**.)

PROCEDURE sign (x: **R**): **R**;

(Значение вызова этой функции при положительном или отрицательном значении параметра x есть **s**. При нулевом значении параметра x значением вызова может быть любое из двух возможных значений для **s**.)

PROCEDURE ulp (x: **R**): **R**;

(Значение вызова этой функции есть $c_{\mathbf{p}}$.)

PROCEDURE succ (x: **R**): **R**;

(Значением вызова этой функции является значение типа **R**, "следующее" за значением параметра x, то есть значение выражения x + ulp(x), если такое значение существует. В противном случае имеет место исключительная ситуация.)

PROCEDURE pred (x: **R**): **R**;

(Значением вызова этой функции является значение типа **R**, "предшествующее" значению параметра x, то есть значение выражения x – ulp(x), если такое значение существует. В противном случае имеет место исключительная ситуация.)

PROCEDURE scale (x: **R**, n: INTEGER): **R**;

PROCEDURE synthesise (n: INTEGER, x: **R**): **R**;

(Значение вызова каждой из этих функций есть $x \times \mathbf{r}^{\mathbf{n}}$, если такое значение существует. В противном случае имеет место исключительная ситуация.)

PROCEDURE intpart (x:**R**): **R**;

PROCEDURE fractpart (x:**R**): **R**;

(Значением вызова функции intpart(x) или вызова функции fractpart(x) является соответственно целая или дробная часть значения параметра x с учётом знака этого значения.)

Пример:

Значением вызова функции LowReal.intpart(-3.14) является число –3.0, а значением вызова функции LowReal.fractpart(-3.14) — число –0.14.

PROCEDURE trunc (x: **R**, n: INTEGER): **R**;

(Если значение **n** параметра n меньше или равно 0,

то имеет место исключительная ситуация. В противном случае значение вызова функции trunc(x,n) есть $\mathbf{s} \times \mathbf{m'} \times \mathbf{r}^{\mathbf{e}}$, где мантисса **m'** представляется дробью $0.a_1 \ldots a_{\mathbf{n}}\ 0 \ldots 0$, если $\mathbf{n} \leq \mathbf{p}$, а иначе совпадает с мантиссой **m**.)

PROCEDURE round (x: **R**, n: INTEGER): **R**;

(Если значение **n** параметра n меньше или равно 0, то имеет место исключительная ситуация. В противном случае значение вызова функции round(x,n) есть такое из двух значений выражений trunc(x,n) и $\text{trunc}(x,n) + \mathbf{s} \times c_{\mathbf{n}}$, абсолютная величина разности которого со значением параметра x ми-
нимальна.)

PROCEDURE setMode (m: Modes);

(При исполнении вызова процедуры setMode(m) каждая особенность выполнения операций над значениями типа **R**, нумеруемая некоторым целым числом — элементом значения параметра m, или, соответственно, не нумеруемая никаким таким целым числом, становится задействованной или, соответственно, не задействованной.

Если, однако, текущая сопрограмма — о сопрограммах см. в 20 — отлична от главной сопрограммы, то эффект исполнения этого вызова не определён.
)

PROCEDURE currentMode (): Modes;

(Значением вызова этой функции является такое

множество целых чисел, что каждая задействованная или, соответственно, не задействованная особенность выполнения операций над значениями типа **R** нумеруется некоторым целым числом — элементом этого множества или, соответственно, не нумеруется никаким таким целым числом.)

PROCEDURE IsLowException (): BOOLEAN;

(Значение вызова этой функции есть ***истина*** в том и только том случае, если текущая сопрограмма находится в исключительном состоянии, возникшем при исполнении процедуры, описанной в некотором определении из модуля определений **L**.)

17.1.3. Стандартный модуль CharClass

С помощью средств данного модуля можно определить, к какому "классу" литерных значений относится то или иное конкретное литерное значение.

В модуле определений CharClass имеются следующие определения процедур. В пояснениях к этим процедурам-функциям даются те необходимые и достаточные условия, при которых значение вызова соответствующей функции есть ***истина***.

PROCEDURE IsNumeric (ch: CHAR): BOOLEAN;

(Значение параметра ch соответствует цифре.)

PROCEDURE IsLetter (ch: CHAR): BOOLEAN;

(Значение параметра ch соответствует букве.)

PROCEDURE IsUpper (ch: CHAR): BOOLEAN;

(Значение параметра ch соответствует заглавной букве.)

PROCEDURE IsLower (ch: CHAR): BOOLEAN;

(Значение параметра ch соответствует строчной букве.)

PROCEDURE IsControl (ch: CHAR): BOOLEAN;

(Значение параметра ch есть управляющее литерное значение.)

PROCEDURE IsWhiteSpace (ch: CHAR): BOOLEAN;

(Значение параметра ch соответствует пробелу или является форматирующим литерным значением.)

17.2. Стандартные модули для вещественной математики

Стандартные модули RealMath и LongMath задают константы и функции, позволяющие осуществлять математические действия над значениями соответственно типов REAL и LONGREAL. Идентификаторы, описываемые в этих двух модулях, попарно лексически совпадают.

а) Константы

pi (число пи)

exp1 (число e)

б) Функции

(Даны заголовки соответствующих описаний, в

которых **R** означает соответственно REAL или LONGREAL, и математический смысл функций, если он не вполне очевиден.)

PROCEDURE sqrt(x: **R**): **R**
(получение квадратного корня)
PROCEDURE exp(x: **R**): **R**
(показательная функция)
PROCEDURE ln(x: **R**): **R**
(получение натурального логарифма)
PROCEDURE sin(x: **R**): **R**
PROCEDURE cos(x: **R**): **R**
PROCEDURE tan(x: **R**): **R**
PROCEDURE arcsin(x: **R**): **R**
PROCEDURE arccos(x: **R**): **R**
PROCEDURE arctan(x: **R**): **R**
PROCEDURE power(x,y: **R**): **R** (x в степени y)
PROCEDURE round(x: **R**): INTEGER
(округление значения параметра до ближайшего целого числа)

17.3. Стандартные модули для комплексной математики

Стандартные модули ComplexMath и LongComplexMath задают константы и функции, позволяющие осуществлять математические действия над значениями соответственно типов COMPLEX и LONGCOMPLEX. Идентификаторы, описываемые в

этих двух модулях, попарно лексически совпадают.

а) Константы

i (мнимая единица)
one (вещественная единица)
zero (нуль)

б) Функции

(Даны определения процедур-функций из модулей определения ComplexMath и LongComplexMath, причём **C** означает, соответственно, COMPLEX либо LONGCOMPLEX, а **R** — REAL либо LONGREAL; в скобках указан математический смысл функций, если он не вполне очевиден.)

PROCEDURE abs(z: **C**): **R**;
(модуль комплексной величины z)
PROCEDURE arg(z: **C**): **R**;
(аргумент комплексной величины z)
PROCEDURE conj(z: **C**): **C**;
(сопряжённая комплексная величина)
PROCEDURE power(u: **C**; v: **R**): **C**; (u в степени v)
PROCEDURE sqrt(z: **C**): **C**;
(главное значение квадратного корня)
PROCEDURE exp(z: **C**): **C**;
(комплексная показательная функция)
PROCEDURE ln(z: **C**): **C**;
(главное значение натурального логарифма)

```
PROCEDURE sin(z: C): C;
PROCEDURE cos(z: C): C;
PROCEDURE tan(z: C): C;
PROCEDURE arcsin(z: C): C;
PROCEDURE arccos(z: C): C;
PROCEDURE arctan(z: C): C;
PROCEDURE polarToComplex(u,v: R): C;
        (получение комплексного значения по
         модулю u и аргументу v)
PROCEDURE scalarMult(u: R; v: C): C;
        (произведение вещественной величины и
         комплексной величины)
```

17.4. Стандартные модули для ввода-вывода

В настоящем пособии не будет приводиться вся система стандартных модулей и содержащихся в них процедур для ввода-вывода, развёрнутая в Стандарте. Ограничимся в нашем изложении самым необходимым.

Процедуры ввода и вывода имеют дело с некоторыми *потоками данных*, соответственно *входными* и *выходными*. Элементами потоков данных являются литеры, пробел и *переход на новую строчку*. Можно говорить о двух уровнях представления элемента потока — внешнем (представление на устройствах ввода/вывода) и внутреннем (машинном). Внешним представлением элемента потока — литеры или пробела служит сама эта литера или сам пробел, а

внутренним представлением такого элемента потока служит внутреннее представление литерного значения, соответствующего этой литере или пробелу. Внешнее представление перехода на новую строчку определяется реализацией для разных устройств ввода/вывода как какой-то физический способ осуществления такого перехода (см. 17.4.2, 17.4.3), а внутреннее представление перехода на новую строчку есть определяемая реализацией комбинация битов. (Там, где это не приводит к двусмысленности, мы для простоты изложения не будем различать внутреннее и внешнее представления элемента потока, называя их просто "элемент потока", "литера", "пробел", "переход на новую строчку".)

Типичные действия процедуры ввода ("чтение") состоят в том, что она выбирает из того или иного входного потока несколько "очередных" элементов, преобразует их в некоторое значение определённого типа и передаёт это значение некоторой переменной программы. Процедура вывода совершает, как правило, обратный процесс ("запись"): преобразует некоторое значение определённого типа в последовательность очередных элементов выходного потока.

Считается, что входные и выходные потоки идут через некоторые *каналы*, *открываемые* и *закрываемые* на тех или иных физических устройствах. (Каналы идентифицируются некоторыми значениями — *идентификациями* канала.) Имеются определяемые реализацией и всегда открытые

стандартные каналы — стандартный канал ввода и стандартный канал вывода. Имеется также единственный *фиктивный канал*, никогда не открываемый. Остальные используемые в программе каналы создаются пользователем, открываются и закрываются в ходе исполнения программы. Процедуры ввода и вывода исполняются на каналах. А именно, они могут исполняться либо непосредственно на созданных и открытых пользователем каналах, либо через *подразумеваемые каналы* — подразумеваемый канал ввода или подразумеваемый канал вывода. В начале исполнения программы подразумеваемый канал ввода (вывода) считается совмещённым с соответствующим стандартным каналом. В дальнейшем подразумеваемые каналы могут с помощью специальных процедур совмещаться с любым существующим каналом.

Мы будем далее рассматривать только два типа физических устройств, на которых могут открываться каналы: интерактивное (диалоговое) терминальное устройство компьютера и файлы прямого доступа. Но предварительно познакомимся с некоторыми вспомогательными стандартными модулями.

(О стандартных каналах в реализации XDS см. Б.5.е.)

17.4.1. Стандартные модули IOChan, StdChans, IOResult, SIOResult, ChanConsts

Для удобства изложения мы будем считать, что все приводимые в этом пункте и описываемые в модулях IOChan, StdChans, IOResult, SIOResult и ChanConsts идентификаторы подвергаются неквалифицируемому импорту во все рассматриваемые в этом и дальнейших пунктах стандартные модули ввода-вывода.

а) Модуль IOChan

В модуле определений IOChan описан, в частности, тип ChanId, значениями которого являются идентификации каналов.

б) Модуль StdChans

Модуль определений StdChans содержит, в частности, следующие описания процедур:

```
PROCEDURE StdInChan (): ChanId;
PROCEDURE StdOutChan (): ChanId;
PROCEDURE NullChan (): ChanId;
PROCEDURE InChan (): ChanId;
PROCEDURE OutChan (): ChanId;
PROCEDURE SetInChan (cid: ChanId);
PROCEDURE SetOutChan (cid: ChanId);
```

Значениями вызовов функций StdInChan, StdOutChan и NullChan являются соответственно идентификации стандартного канала ввода, стандартного канала вывода и фиктивного канала.

Значениями вызовов функций InChan и OutChan являются идентификации текущих подразумеваемых каналов ввода и вывода. Вызовы процедур SetInChan(cid) и SetOutChan(cid) совмещают подразумеваемый канал ввода и подразумеваемый канал вывода с каналом, идентификация которого есть значение параметра cid. (Поскольку такой канал в пояснениях к дальнейшим процедурам будет упоминаться неоднократно, мы будем в этих пояснениях говорить о нём как о "канале cid".)

в) Модули IOResult и SIOResult

Модуль определений IOResult содержит следующие описание типа и определение процедуры:

```
TYPE ReadResults =
   (notKnown, allRight, outOfRange, wrongFormat,
    endOfLine, endOfInput);
PROCEDURE ReadResult (cid: ChanId): ReadResults;
```

Константы, обозначаемые идентификаторами, входящими в описание типа-перечисления ReadResults, выступают в качестве *результатов чтения*, устанавливаемых на конкретных каналах при исполнении на этих каналах процедур ввода. Считается, что до того, как на канале исполнена хотя бы одна процедура ввода, на данном канале установлен результат чтения notKnown. После исполнения очередной процедуры ввода результатом чтения становится — в зависимости от содержания входного потока — некоторая другая из числа данных констант. Значением вызова функции ReadResult

Значением вызова функции ReadResult является текущее значение результата чтения, установленного на канале cid.

Модуль определений SIOResult содержит такое же описание типа ReadResults, что и модуль IOResult, а также определение процедуры:

```
PROCEDURE ReadResult (): ReadResults;
```

Вызов процедуры из модуля SIOResult	равносилен вызову процедуры из модуля IOResult
ReadResult()	ReadResult(InChan())

г) Модуль ChanConsts

Модуль определений ChanConsts содержит описание типа-перечисления ChanFlag, явное представление которого содержит в своём списке идентификаторов идентификаторы readFlag, writeFlag, oldFlag, textFlag, rawFlag, echoFlag. Этот модуль содержит также следующие описания:

```
TYPE FlagSet = SET OF ChanFlags;
CONST read = FlagSet{readFlag};
      write = FlagSet{writeFlag};
      old = FlagSet{oldFlag};
      text = FlagSet{textFlag};
```

```
        raw = FlagSet{rawFlag};
        echo = FlagSet{echoFlag};

TYPE OpenResults =
      (opened, wrongNameFormat, wrongFlags,
       tooManyOpen, outOfChans,
        wrongPermissions, noRoomOnDevice,
       noSuchFile, fileExists, wrongFileType,
       noTextOperations, noRawOperations,
        noMixedOperations, alreadyOpen, otherProblem);
```

Значения типа ChanFlags суть некоторые признаки, которыми снабжаются открываемые каналы. Значения типа FlagSet суть комбинации таких признаков. С помощью описанных в модуле констант типа FlagSet можно легко задавать любые требуемые комбинации признаков. Так вместо

```
FlagSet{readFlag,writeFlag}
```

можно писать просто read+write. Те же константы будут дальше в нашем тексте просто обозначать соответствующие признаки. Зададим их смысл:

read — допустимы операции ввода
write — допустимы операции вывода
old — до открытия канала мог (должен) существовать некоторый файл
text — допустимы операции, предусмотренные в 17.4.4.1 — 17.4.4.3 (то есть операции над интерпретированными двоичными кодами)

raw — допустимы операции, предусмотренные в 17.4.4.4 (то есть операции над неинтерпретированными двоичными кодами)

echo — на интерактивном устройстве установлен режим единичной литеры; этот признак используется только при работе со стандартным модулем TermFile (см. 17.4.2)

Идентификаторы констант, входящие в описание типа OpenResults, представляют собой английские выражения, мнемонически характеризующие возможные результаты попытки открыть канал. (Значение константы opened сигнализирует об успешном открытии канала, значения остальных констант — о той или иной ошибке.)

17.4.2. Стандартный модуль TermFile

Данный модуль предоставляет средства для задания канала, связанного с интерактивным терминальным устройством компьютера.

Входной поток, идущий через этот канал, считается потенциально бесконечным (так как в любой момент в него может поступить новый элемент потока от устройства). Поэтому работающая на этом канале процедура ввода не воспринимает отсутствие очередного элемента потока как его исчерпание, а переходит в режим ожидания нового элемента. Движение по входному и выходному потокам, связанным с терминальным устройством, с целью

чтения и записи может происходить только в одну сторону, то есть без возврата к пройденным элементам.

Интерактивное устройство может находиться в одном из двух режимов: *режиме единичной литеры* и *строчечном режиме*. В режиме единичной литеры каждый элемент потока, поданный с устройства, сразу же включается во входной поток (а выдача на устройство обратного сигнала происходит при выборке этого элемента процедурой ввода). В строчечном режиме подаваемые с устройства элементы потока накапливаются отдельно и включаются в поток только при получении перехода на новую строчку (а выдача обратного сигнала может происходить до начала выборки этих элементов процедурой ввода).

Переход на новую строчку, осуществляемый при выборке соответствующего элемента потока процедурой ввода или при занесении его процедурой вывода, определяется обычно реализацией для терминального устройства как перевод курсора в начало следующей строчки экрана.

В модуле определений TermFile имеются следующие определения процедур:

```
PROCEDURE Open (VAR cid: ChanId; flags: FlagSet;
    VAR res: OpenResults);
```

(Процедура делает попытку открыть на терминальном устройстве канал, снабжённый признаками, определяемыми значением параметра

flags. Отсутствие задания признака raw влечёт неявное наличие признака text. Признак old является избыточным. Если канал оказывается успешно открытым, то переменной cid присваивается идентификация этого открытого канала, а в противном случае — идентификация фиктивного канала. Переменной res присваивается в любом случае значение, сигнализирующее о результате работы процедуры.)

PROCEDURE IsTermFile (cid: ChanId): BOOLEAN;

(Значение вызова этой процедуры-функции есть ***истина*** в том и только том случае, если значением параметра cid является идентификация некоторого канала, открытого на терминальном устройстве.)

PROCEDURE Close (VAR cid: ChanId);

(Процедура закрывает открытый на терминальном устройстве канал cid и присваивает переменной cid идентификацию фиктивного канала. Если открытого канала cid не было, то возбуждается исключительная ситуация.)

17.4.3. Стандартный модуль RndFile

Данный модуль предоставляет средства для задания каналов, связанных с именованными файлами прямого доступа (обычно, библиотечными файлами).

Процедура открытия канала связывает его с некоторым именованным файлом, либо уже существующим, либо вновь создаваемым, и

устанавливает на этом файле три "позиции": начальную позицию, конечную позицию и текущую позицию (позицию чтения/записи). Позиции явно не представимы и доступны для программиста только через посредство специальных процедур. Начальная позиция в дальнейшем не меняется, а две другие позиции могут меняться в ходе исполнения программы. Поток данных на файле располагается в каждый момент между начальной и конечной позициями, а текущая позиция определяет положение очередного элемента потока. Имеется возможность запоминать в любой момент текущую позицию и затем возвращаться к ней, продолжая чтение или запись с этого места на файле. При открытии канала текущая позиция совпадает с начальной позицией.

Переход на новую строчку, осуществляемый при выборке соответствующего элемента потока процедурой ввода, означает для файла прямого доступа переход текущей позиции в начало следующей строчки файла. Достижение в ходе чтения конечной позиции рассматривается как исчерпание потока. При записи происходит размещение очередных элементов выходного потока последовательно по файлу (с заменой ранее размещённых на тех же местах элементов, если таковые были). Переход на новую строчку, осуществляемый при занесении соответствующего элемента потока процедурой вывода, означает для файла прямого доступа переход

текущей позиции в начало следующей строчки файла независимо от того, остались ли прежние элементы в текущей строчке файла. Достижение в ходе записи конечной позиции не приводит к концу записи, а конечная позиция после этого сдвигается вместе с текущей позицией.

В модуле определений описан тип FilePos, значения которого суть позиции. Приведём далее следующие имеющиеся в модуле определения процедур:

```
PROCEDURE OpenOld (VAR cid: ChanId;
                   name: ARRAY OF CHAR;
                   flags: FlagSet; VAR res: OpenResults);
```

(Процедура делает попытку найти существующий файл прямого доступа, имя которого есть значение параметра name, и открыть на нём канал, снабжённый признаками, определяемыми значением параметра flags. Отсутствие задания признака write влечёт неявное наличие признака read. Признак old является избыточным. Если канал оказывается успешно открытым, то переменной cid присваивается идентификация этого открытого канала, а в противном случае — идентификация фиктивного канала. Переменной res присваивается в любом случае значение, сигнализирующее о результате работы процедуры. При успешном открытии канала процедура устанавливает начальную позицию в начале файла,

а конечную позицию — после последней заполняющей файл литеры.)

```
PROCEDURE OpenClean (VAR cid: ChanId;
                     name: ARRAY OF CHAR;
                     flags: FlagSet; VAR res: OpenResults);
```

(Процедура делает попытку открыть на некотором файле канал, снабжённый признаками, определяемыми значением параметра flags. Этот файл есть либо существующий файл прямого доступа, имя которого есть значение параметра name, либо — при отсутствии такового — новый, создаваемый процедурой файл с тем же именем. Если файл с этим именем существовал, но среди упомянутых признаков не было признака old, то такая попытка является заведомо безуспешной. Признак write является избыточным. Если канал оказывается успешно открытым, то переменной cid присваивается идентификация этого открытого канала, а в противном случае — идентификация фиктивного канала. Переменной res присваивается в любом случае значение, сигнализирующее о результате работы процедуры.

При успешном открытии канала процедура устанавливает начальную позицию в начале файла и совмещает с ней конечную позицию, так что всё прежнее содержимое файла, если оно и было, становится несуществующим.)

```
PROCEDURE StartPos(cid: ChanId): FilePos;
```

PROCEDURE EndPos(cid: ChanId): FilePos;
PROCEDURE CurrentPos(cid: ChanId): FilePos;
(Если на некотором файле прямого доступа имеется открытый канал cid, то значением вызова каждой из этих функций является соответственно начальная позиция, конечная позиция и текущая позиция этого файла. В противном случае возбуждается исключительная ситуация.)
PROCEDURE SetPos(cid: ChanId; pos: FilePos);
(Если на некотором файле прямого доступа имеется открытый канал cid, то текущей позицией этого файла становится значение параметра pos. В противном случае возбуждается исключительная ситуация.)
PROCEDURE Close (VAR cid: ChanId);
(Процедура закрывает открытый на файле прямого доступа канал cid и присваивает переменной cid идентификацию фиктивного канала. Если открытого канала cid не было, то возбуждается исключительная ситуация.)

17.4.4. Ввод и вывод текстовой, числовой и двоичной информации

Все процедуры, приведённые в подпунктах данного пункта, осуществляют ввод и вывод через канал cid.

17.4.4.1. Стандартные модули TextIO и STextIO

С помощью этих модулей осуществляется ввод и

вывод литерных и строковых значений.

Из модуля определений TextIO возьмём следующие определения процедур:

а) Процедуры ввода

```
PROCEDURE ReadChar (cid: ChanId; VAR ch: CHAR);
```

(Процедура предназначена для чтения литерного значения из входного потока.

Если к моменту исполнения процедуры входной поток исчерпан, то значение переменной ch становится неопределённым и на канале устанавливается результат чтения endOfInput. Иначе, если очередной элемент потока есть переход на новую строчку, то значение переменной ch также становится неопределённым и устанавливается результат чтения endOfLine. В противном случае из потока выбирается его очередной элемент, переменной ch присваивается соответствующее литерное значение, и устанавливается результат чтения allRight.)

```
PROCEDURE ReadString (cid: ChanId;
                      VAR s: ARRAY OF CHAR);
```

(Процедура предназначена для чтения из входного потока последовательности литерных значений, воспринимаемой как строка. Пусть n есть число компонент переменной s. Если к моменту исполнения процедуры входной поток исчерпан, то на канале устанавливается результат чтения endOfInput. Иначе, если очередной элемент потока

есть переход на новую строчку, то устанавливается результат чтения endOfLine. Наконец, если во входном потоке имеются очередные литеры и/или пробелы, то из него выбирается максимальное, но не превосходящее n, количество m этих элементов потока, литерные значения, соответствующие указанным элементам, последовательно присваиваются компонентам s[0], ..., s[m – 1] переменной s, а если к тому же m < n, то компоненте s[m] присваивается ***признак конца строки***. При этих условиях на канале устанавливается результат чтения allRight.)

```
PROCEDURE ReadToken (cid: ChanId;
                     VAR s: ARRAY OF CHAR);
```

(Процедура предназначена для чтения из входного потока "слова", то есть максимального набора последовательных литерных значений, не соответствующих пробелу.

Пусть n есть число компонент переменной s. Вначале процедура пропускает, то есть выбирает без дальнейшей обработки и пересылки все имеющиеся очередные пробелы потока. Если после этого входной поток оказывается исчерпанным, то на канале устанавливается результат чтения endOfInput. Иначе, если очередной элемент потока есть переход на новую строчку, то устанавливается результат чтения endOfLine. Иначе, если во входном потоке имеются очередные литеры в количестве, большем

литеры в количестве, большем чем n, то первые n последовательно выбираются, соответствующие литерные значения присваиваются компонентам s[0], ..., s[n – 1] переменной s, и на канале устанавливается результат чтения outOfRange. Наконец, если таких литер оказалось m ($0 < m \leq n$), то эти m элементов последовательно выбираются, соответствующие литерные значения присваиваются компонентам s[0], ..., s[m – 1] переменной s, а если к тому же $m < n$, то компоненте s[m] присваивается ***признак конца строки***. При этих условиях на канале устанавливается результат чтения allRight.)

PROCEDURE SkipLine(cid: ChanId);

(Процедура пропускает все очередные литеры, пока следующим элементом потока не окажется переход на новую строчку или пока поток не будет исчерпан. В первом случае осуществляется переход на новую строчку, и на канале устанавливается результат чтения allRight. Во втором случае устанавливается результат чтения endOfInput.)

б) Процедуры вывода

PROCEDURE WriteChar (cid: ChanId; ch: CHAR);

(Если значение переменной ch есть литерное значение, соответствующее некоторой литере или пробелу, то эта литера или пробел заносится в выходной поток.

Иначе, если значение данной переменной есть управляющее или форматирующее значение, то эффект исполнения данной процедуры определяется реализацией в зависимости от данного значения. В частности, могут быть осуществлены некоторые специальные действия над выходным потоком, как, например, переход на новую строчку, возврат к началу строчки, занесение некоторого количества пробелов и т.д.

В иных случаях эффект исполнения данной процедуры определяется реализацией.)

```
PROCEDURE WriteString (cid: ChanId;
                       s: ARRAY OF CHAR);
```

(Исполнение вызова данной процедуры состоит в последовательном исполнении вызова процедуры WriteChar(cid,s[**i**]), где **i** принимает последовательные целочисленные значения, начиная с 0 и вплоть до исчерпания массива — значения параметра s или же до тех пор, пока очередным значением для s[**i**] не окажется ***признак конца строки***.)

```
PROCEDURE WriteLn (cid: ChanId);
```

(Процедура осуществляет переход на новую строчку.)

Из модуля определений STextIO возьмём следующие определения процедур ввода и вывода:

```
PROCEDURE ReadChar (VAR ch: CHAR);
```

```
PROCEDURE ReadString (VAR s:
                        ARRAY OF CHAR);
PROCEDURE ReadToken (VAR s:
                        ARRAY OF CHAR);
PROCEDURE SkipLine;
PROCEDURE WriteChar (ch: CHAR);
PROCEDURE WriteString (s: ARRAY OF CHAR);
PROCEDURE WriteLn;
```

Вызовы процедур из модуля STextIO	равносильны вызовам процедур из модуля TextIO
ReadChar(ch)	ReadChar(InChan(),ch)
ReadString(s)	ReadString(InChan(),s)
ReadToken(s)	ReadToken(InChan(),s)
SkipLine	SkipLine(InChan())
WriteChar(ch)	WriteChar(OutChan(),ch)
WriteString(s)	
WriteString(OutChan(),s)	
WriteLn	WriteLn(OutChan())

17.4.4.2. Стандартные модули WholeIO и SWholeIO

С помощью этих модулей осуществляется ввод и вывод целочисленных значений, внешними

представлениями которых во входном и выходном потоках служат последовательности литер, имеющие либо форму, синтаксически выражаемую как

десятичное число

(*форма целого без знака*), либо форму, выражаемую как

знак,десятичное число

(*форма целого со знаком*).

Определения, данные в предыдущем абзаце, будут использованы также в пояснениях к процедурам модуля WholeStr (см. 22.3.3.2).

В модуле определений WholeIO имеются следующие определения процедур:

```
PROCEDURE ReadCard (cid: ChanId;
                    VAR card: CARDINAL);
```

(Процедура предназначена для чтения из входного потока целого положительного числа.

Вначале процедура пропускает все имеющиеся очередные пробелы потока. Если после этого поток оказался исчерпанным, то на канале устанавливается результат чтения endOfInput. Иначе, если очередной элемент есть конец строчки, то устанавливается результат чтения endOfLine. В противном случае процедура, выбирая очередные литеры потока, пытается получить такую, максимальную по длине, последовательность литер, которая представляла

бы в форме целого без знака некоторое целое число. Если такой последовательности не существует, то устанавливается результат чтения wrongFormat. В трёх перечисленных выше случаях значение переменной card становится неопределённым. Если же такая последовательность существует, то процедура обращает её в соответствующее целое число. Если это число находится в диапазоне типа CARDINAL, то оно присваивается переменной card, а на канале устанавливается результат чтения allRight; в противном случае переменной card присваивается значение выражения MAX(CARDINAL), а на канале устанавливается результат чтения outOfRange.)

```
PROCEDURE ReadInt (cid: ChanId;
                   VAR int: INTEGER);
```

(Процедура предназначена для чтения из входного потока целого числа любого знака.

Вначале процедура пропускает все имеющиеся очередные пробелы потока. Если после этого поток оказался исчерпанным, то на канале устанавливается результат чтения endOfInput. Иначе, если очередной элемент есть конец строчки, то устанавливается результат чтения endOfLine. В противном случае процедура, выбирая очередные литеры потока, пытается получить такую, максимальную по длине,

последовательность литер, которая представляла бы в форме целого без знака или в форме целого со знаком некоторое целое число. Если такой последовательности не существует, то устанавливается результат чтения wrongFormat. В трёх перечисленных выше случаях значение переменной int становится неопределённым. Если же такая последовательность существует, то процедура обращает её в соответствующее целое число. Если это число находится в диапазоне типа INTEGER, то оно присваивается переменной int, а на канале устанавливается результат чтения allRight; в противном случае переменной int присваивается значение выражения MAX(INTEGER) или выражения -MAX(INTEGER) в зависимости от того, является ли полученное число положительным или отрицательным, а на канале устанавливается результат чтения outOfRange.)

```
PROCEDURE WriteCard (cid: ChanId;
                     card: CARDINAL; width: CARDINAL);
```

(Процедура преобразует значение параметра card в представляющую это значение последовательность литер формы целого без знака. Если значение параметра width есть 0, то перед литерами заносится один пробел. Иначе, если число n этих литер меньше значения параметра width, то перед ними заносятся width – n пробелов. Полученная

последовательность пробелов и литер заносится в выходной поток.)

```
PROCEDURE WriteInt (cid: ChanId; int: INTEGER;
                    width: CARDINAL);
```

(Процедура преобразует значение параметра int в представляющую это значение последовательность литер формы целого без знака, если это значение неотрицательно, или формы целого со знаком, если оно отрицательно. Если значение параметра width есть 0, то перед литерами заносится один пробел. Иначе, если число n этих литер меньше значения параметра width, то перед ними заносятся width – n пробелов. Полученная последовательность пробелов и литер заносится в выходной поток.)

В модуле определений SWholeIO имеются следующие определения процедур:

```
PROCEDURE ReadCard (VAR card: CARDINAL);
PROCEDURE ReadInt (VAR int: INTEGER);
PROCEDURE WriteCard (card: CARDINAL;
                     width: CARDINAL);
PROCEDURE WriteInt (int: INTEGER;
                    width: CARDINAL);
```

Вызовы процедур из модуля SWholeIO	равносильны вызовам процедур из модуля WholeIO
ReadCard(card)	ReadCard(InChan(),card)
ReadInt(int)	ReadInt(InChan(),int)
WriteCard(card)	WriteCard(OutChan(),card)
WriteInt(int)	WriteInt(OutChan(),int)

17.4.4.3. Стандартные модули RealIO, SReal, LongIO и SLongIO

С помощью модулей RealIO и SRealIO осуществляется ввод и вывод значений типа REAL, а с помощью модулей LongIO и SLongIO осуществляется ввод и вывод значений типа LONGREAL. Внешними представлениями этих значений во входном и выходном потоках служат последовательности литер, имеющие либо форму, синтаксически выражаемую как

[знак],целая часть,[дробная часть]
(*форма вещественного с фиксированной точкой*),

либо форму, выражаемую как

[знак],целая часть,[дробная часть],"E",показатель
(*форма вещественного с плавающей точкой*).

Исходным строковым представлением вещественного числа **R**, настроенным на "позицию" **P**, где **P** есть некоторое целое число, будем называть такую последовательность литер, которая представляет в форме с фиксированной точкой число **R**, округлённое до десяти в степени –**P**, если **P** неотрицательно, и до десяти в степени –**P** – 1 в противном случае, и удовлетворяет следующим условиям:

- в последовательности целая часть либо начинается с цифры, отличной от цифры "0", либо состоит из единственной цифры "0";
- последовательность не содержит знака "+";
- последовательность содержит дробную часть в том и только том случае, если **P** неположительно.

Исходным строковым представлением вещественного числа **R**, настроенным на "число значащих цифр" **F**, где **F** есть некоторое целое неотрицательное число, будем называть такую последовательность литер, которая представляет в форме с плавающей запятой в необходимой мере округлённое число **R** и удовлетворяет следующим условиям:

- если **F** положительно, то последовательность содержит **F** цифр (а если **F** равно нулю, то количество цифр определяется реализацией);
- последовательность содержит единственную цифру в целой части;

- либо эта единственная цифра не есть цифра "0", либо каждая цифра в целой и дробной частях есть цифра "0";
- последовательность не содержит знака "+";
- последовательность не содержит дробной части, состоящей лишь из литеры ".".

Пусть показатель в исходном строковом представлении числа **R**, настроенном на число значащих цифр **F**, представляет целое число $3 \times \mathbf{n} + \mathbf{p}$, где **n** — произвольное целое число, а **p** есть 0, 1 или 2. Назовём тогда *техническим строковым представлением* числа **R**, настроенным на число значащих цифр **F**, указанное исходное представление, подвергнутое следующим преобразованиям:

- показатель изменяется так, чтобы представлять число $3 \times \mathbf{n}$;
- если представление не содержит дробной части, то в конец целой части приписываются **p** цифр "0";
- иначе, если представление содержит дробную часть, содержащую **q** ($\mathbf{q} \leq \mathbf{p}$) цифр, то в конец дробной части приписываются $\mathbf{p} - \mathbf{q}$ цифр "0", а литера "." удаляется;
- иначе **p** первых цифр дробной части переносятся в конец целой части.

Конкретная реализация может наложить ограничения на значения "позиции" **P** и "числа

значащих цифр" **F**.

(Об эффекте использования в реализации XDS величин **P** и **F**, превосходящих допустимые, см. Б.5.6.)

Определения, данные в предыдущих абзацах, будут использованы также в пояснениях к процедурам модулей RealStr и LongStr (см. 22.3.3.3).

В модулях определений RealIO и LongIO имеются следующие определения процедур, где **R** обозначает — в процедурах из первого или второго модуля — соответственно REAL или LONGREAL:

PROCEDURE ReadReal (cid: ChanId; VAR real: **R**);

(Процедура предназначена для чтения из входного потока вещественного числа.

Вначале процедура пропускает все имеющиеся очередные пробелы потока. Если после этого поток оказался исчерпанным, то на канале устанавливается результат чтения endOfInput. Иначе, если очередной элемент есть конец строчки, то устанавливается результат чтения endOfLine. В противном случае процедура, выбирая очередные литеры потока, пытается получить такую, максимальную по длине, последовательность литер, которая представляла бы в форме вещественного с фиксированной точкой или форме вещественного с плавающей точкой некоторое вещественное числовое значение. Если такой последовательности не

существует, то устанавливается результат чтения wrongFormat. В трёх перечисленных выше случаях значение переменной real становится неопределённым. Если же такая последовательность существует, то процедура обращает её в соответствующее вещественное числовое значение. Если это значение находится в диапазоне типа **R**, то оно присваивается переменной real, а на канале устанавливается результат чтения allRight; в противном случае переменной real присваивается значение выражения MAX(**R**) или выражения –MAX(**R**) в зависимости от того, является ли полученное значение положительным или отрицательным, а на канале устанавливается результат чтения outOfRange.)

PROCEDURE WriteFixed (cid: ChanId; real: **R**;
place: INTEGER; width: CARDINAL);

(Процедура строит последовательность литер, являющуюся исходным строковым представлением значения параметра real, настроенным на позицию, равную значению параметра place. Если значение параметра width равно 0, то перед этой последовательностью помещается пробел. Иначе, если число n литер последовательности меньше значения параметра width, то перед последовательностью помещаются width – n пробелов. Полученная

последовательность пробелов и литер заносится в выходной поток.)

```
PROCEDURE WriteFloat (cid: ChanId; real: R;
                      sigFigs,width: CARDINAL);
PROCEDURE WriteEng (cid: ChanId; real: R;
                    sigFigs,width: CARDINAL);
```

(Процедуры строят последовательность литер, являющуюся, соответственно, исходным или техническим строковым представлением значения параметра real, настроенным на число значащих цифр, равное значению параметра sigFigs. Если в этой последовательности дробная часть не содержит цифры, отличной от 0, то дробная часть удаляется. Также, если в последовательности показатель представляет число 0, то он удаляется вместе с литерой "E". Если значение параметра width равно 0, то перед последовательностью, полученной после указанных преобразований, помещается пробел. Иначе, если число n литер полученной последовательности меньше значения параметра width, то перед ней помещаются width – n пробелов. Полученная после всего этого последовательность пробелов и литер заносится в выходной поток.)

PROCEDURE WriteReal (cid: ChanId; real: **R**;
width: CARDINAL);

(Если при некотором, по возможности наибольшем, целочисленном значении **P** исходное строковое представление значения параметра real, настроенное на позицию **P**, состоит из стольких литер, каково значение параметра width, то вызов данной процедуры равносилен вызову процедуры WriteFixed(cid,real,**P**,width). В противном случае вызов данной процедуры равносилен вызову процедуры WriteFloat(cid,real,sigFigs,width) с таким ненулевым значением параметра sigFigs, чтобы количество литер в вырабатываемой этой процедурой последовательности по возможности не превосходило значения параметра width. Особым случаем является вызов процедуры WriteReal(cid,real,width), когда значение параметра width равно 0. Такой вызов равносилен вызову WriteFloat(cid,real,0,0).)

В модулях определения SRealIO и SLongIO имеются следующие определения процедур, где смысл обозначения **R** тот же:

PROCEDURE ReadReal (VAR real: **R**);

```
PROCEDURE WriteFixed (real: R; place: INTEGER;
                      width: CARDINAL);
 PROCEDURE WriteFloat (real: R;
                       sigFigs,width: CARDINAL);
PROCEDURE WriteEng (real: R;
                    sigFigs,width: CARDINAL);
PROCEDURE WriteReal (real: R; width: CARDINAL);
```

Вызовы процедур из модулей SRealIO и SLongIO	равносильны вызовам процедур из модулей RealIO и LongIO
ReadReal(real)	ReadReal(InChan(),real)
WriteFixed(real,place,width)	WriteFixed(OutChan(), real,place,width)
WriteFloat(real,sigFigs,width)	WriteFloat(OutChan(), real,sigFigs,width)
WriteEng(real,sigFigs,width)	WriteEng(OutChan(), real,sigFigs,width)
WriteReal(real,width)	WriteReal(OutChan(), real,width)

17.4.4.4. Стандартные модули RawIO и SRawIO

С помощью процедур этих модулей производится

ввод и вывод не интерпретируемых и не преобразуемых двоичных кодов (наборов битов), представляющих собой последовательности значений типа SYSTEM.LOC.

В модуле определений RawIO имеются следующие определения процедур (в определениях и пояснениях к ним исходим из предположения, что идентификатор LOC подвергнут неквалифицируемому импорту из модуля SYSTEM в рассматриваемые нами модули):

```
PROCEDURE Read (cid: ChanId;
                VAR to: ARRAY OF LOC);
```

(Пусть n есть число компонент переменной to. Если к моменту исполнения процедуры входной поток исчерпан, то на канале устанавливается результат чтения endOfInput. Иначе, если во входном потоке имеются очередные элементы потока, то из него выбирается максимальное, но не превосходящее n, количество m этих элементов. Внутренние представления этих элементов становятся внутренними представлениями значений последовательных компонент to[0], ..., to[m-1] переменной to. Если m<n, то устанавливается результат чтения wrongFormat, а если m=n, то устанавливается результат чтения allRight.)

```
PROCEDURE Write (cid: ChanId;
                 from: ARRAY OF LOC);
```

(Процедура заносит в выходной поток внутренние

представления последовательных компонент массива from.)

В модуле определений SRawIO имеются следующие определения процедур:

```
PROCEDURE Read (VAR to: ARRAY OF LOC);
PROCEDURE Write (from: ARRAY OF LOC);
```

Вызовы процедур из модуля SRawIO	равносильны вызовам процедур из модуля RawIO
Read(to)	Read(InChan(),to)
Write(from)	Write(OutChan(),from)

17.4.5. Примеры на ввод и вывод

Пример 1:

Пусть через терминальное устройство поступает информация о новых пациентах больницы: фамилия (не более 15 букв), имя (не более 10 букв), пол (M или F), год рождения, номер палаты, рост (в см как целое число), вес (в кг как вещественное число). Сведения относительно каждого пациента вводятся, начиная с новой строчки. Эта информация пересылается сразу же в библиотечный файл в табулированной форме. Программу, предназначенную для этой цели, считаем работающей неопределённо долго. В случае

нарушения формата ввода программа посылает в соответствующую строчку таблицы последовательность символов #.

```
MODULE entry;
      FROM IOChan IMPORT ChanId;
      FROM IOResult
               IMPORT ReadResults, ReadResult;
      FROM ChanConsts
              IMPORT read, text, OpenResults;
      FROM TermFile IMPORT Open;
      FROM RndFile IMPORT OpenClean;
      FROM TextIO IMPORT ReadChar, SkipLine,
                           WriteChar, WriteString, WriteLn;
      FROM WholeIO IMPORT ReadCard, WriteCard;
      FROM RealIO IMPORT ReadReal, WriteFixed;
      VAR in,out: ChanId; res: OpenResults; char: CHAR;
            card: CARDINAL; real: REAL;
           mistake: BOOLEAN;
      PROCEDURE set_mistake;
      BEGIN mistake:=TRUE;
         WriteString(out,
           "###################################");
         WriteLn(out); SkipLine(in)
      END set_mistake;
```

```
PROCEDURE test_mistake;
BEGIN IF ReadResult(in) # allRight
          THEN set_mistake
          END
END test_mistake;
PROCEDURE skip;
BEGIN WHILE (char = " ") &
                  (ReadResult(in) = allRight)
           DO ReadChar(in,char)
           END
END skip;
PROCEDURE spaces;
BEGIN skip; test_mistake
END spaces;

PROCEDURE name(nl: CARDINAL);
    VAR i: CARDINAL;
BEGIN IF ~ mistake
          THEN i:=0;
                    WHILE (char # " ") &
                             (ReadResult(in) = allRight)
                    DO i:=i+1; WriteChar(out,char);
                               ReadChar(in,char)
                    END;
                    test_mistake;



                    IF ~ mistake & (i > nl)
```

```
                    THEN set_mistake
                    END;
                    FOR i:=i+1 TO nl+1
                    DO WriteChar(out," ")
                    END
        END
END name;
PROCEDURE whole(wl: CARDINAL);
BEGIN IF ~ mistake
          THEN ReadCard(in,card); test_mistake;
                    IF ~ mistake
                    THEN WriteCard(out,card,wl)
                    END
         END
END whole;

BEGIN Open(in,read,res);
          OpenClean(out,"list",text,res);
          LOOP mistake:=FALSE; ReadChar(in,char);
          spaces; name(15); spaces; name(10); spaces;
          IF ~ mistake
          THEN IF (char = "M") OR (char = "F")
                    THEN WriteChar(out,char);
                              ReadChar(in,char)


                ELSE set_mistake
                END
```

```
                    END;
                    IF ~ mistake
                    THEN IF char # " "
                            THEN set_mistake
                            END
                    END;
                    whole(5); whole(4); whole(4);
                    IF ~ mistake
                    THEN ReadReal(in,real); test_mistake;
                             IF ~ mistake
                             THEN WriteFixed(out,real,1,6)
                             END
                    END;
                    IF ~ mistake
                    THEN skip;
                            IF ReadResult(in) # endOfLine
                            THEN set_mistake
                            ELSE SkipLine(in); WriteLn(out)
                            END
                    END;
                END
END entry.
```

Если в ходе работы этой программы с терминального устройства будут посланы сигналы

```
    Sanders Michel      M 1935     301   172       81
Parker  Andrew   M     1971   114      184   73.590
```

Jenkins Mary F 1966 227 157 58.77 ,

то после этого содержимым файла line станет табличка

Sanders	Michel	M	1935	301	172	81.0
Parker	Andrew	M	1971	114	184	73.6
Jenkins	Mary	F	1966	227	157	58.8

Пример 2:

На библиотечном файле source находится некоторый текст, подразделённый пробелами. Фрагменты текста, заключённые в скобки [и], рассматриваются как "примечания". (Каждая скобка находится внутри пары пробелов. Вложенности скобок нет.) Требуется примечания перенумеровать и вынести в конец текста, заменив их вхождение в сам текст соответствующим номером. Кроме того, в ходе преобразования каждая группа последовательных пробелов заменяется на единичный пробел. Преобразованный текст посылается в библиотечный файл target. Предлагаемая программа не проверяет правильности исходного текста.

```
MODULE COMMENTS;
    FROM IOChan IMPORT ChanId;
    FROM IOResult IMPORT ReadResults, ReadResult;
   FROM ChanConsts IMPORT text, OpenResults;
   FROM RndFile IMPORT FilePos, OpenOld,
                OpenClean,IsRndFile, StartPos, CurrentPos,
                EndPos, SetPos, Close;
```

```
FROM TextIO IMPORT ReadChar, ReadString,
          ReadToken, SkipLine,.WriteChar, WriteString,
          WriteLn;
FROM WholeIO IMPORT WriteInt;
VAR  source,target: ChanId; res: OpenResults;
        word: ARRAY [1..100] OF CHAR;
        starts: ARRAY [1..1000] OF FilePos;
        n,i: CARDINAL; on: BOOLEAN;
PROCEDURE skip_lines(write: BOOLEAN);
BEGIN WHILE ReadResult(source) = endOfLine
          DO  SkipLine(source);
                IF write
                THEN WriteLn(target)
                END
          END
END skip_lines;
PROCEDURE send(cond: BOOLEAN);
BEGIN  skip_lines(TRUE); ReadToken(source,word);
           IF cond & (word[1]="[")
           THEN n:=n+1; starts[n]:=CurrentPos(source);
                    WriteString(target,"[");
                   WriteInt(target,n,1);
                     WriteString(target,"] ");
                WHILE word[1] # "]"
                DO skip_lines(FALSE);
                    ReadToken(source,word)
                END
           ELSIF ~cond & (word[1]="]")
```

```
            THEN on:=FALSE
            ELSE WriteString(target,word);
                 WriteChar(target," ")
            END
      END send;

BEGIN OpenOld(source,"from.txt",text,res);
        OpenClean(target,"to.txt",text,res);
        n:=0;
       WHILE ReadResult(source) # endOfInput
        DO send(TRUE)
        END;
        FOR i:=1 TO n
        DO on:=TRUE; WriteLn(target);
           WriteString(target,
"-------------------------------------------------------
----"
                              );
           WriteLn(target); WriteString(target,"[");
           WriteInt(target,i,1);
           WriteString(target,"] ");
           SetPos(source,starts[i]);
           WHILE on
           DO send(FALSE)
            END
       END;
       Close(source); Close(target)
END COMMENTS.
```

Если к началу работы этой программы файл from.txt имеет следующее содержимое

```
 I  have a little shadow [ Not to confound with the shadow econ-
omy! ]
   That goes   in and out      with  me.      [     Like a bodyguard. ]
   And what can be the use    [    Calculable profit.   ]  of    him,
   Is more    than    I    can see.
                                                    (R.L.Stevenson) ,
```

то в результате этой работы содержимым файла to.txt станет

```
   I have a little shadow [1]
   That goes in and out with me. [2]
   And what can be the use [3] of him,
   Is more than I can see.
   (R.L.Stevenson)
-------------------------------------------------------------
[1] Not to confound with the shadow economy!
-------------------------------------------------------------
[2] Like a bodyguard.
-------------------------------------------------------------
[3] Calculable profit.
```

Пример 3:

В раздельных модулях inout описаны и открыты файл infile, связанный с устройством прямого доступа, и файл outfile, связанный с произвольным устройством. (Модуль реализации inout здесь не приводится.) Программный модуль token читает с файла infile последовательности литер, заключённые между пробелами, и записывает их в файл outfile с

новой строчки каждое. Предполагается, что каждая такая последовательность содержит не более 20 литер.

```
MODULE token;
    FROM inout IMPORT infile, outfile;
    FROM TextIO IMPORT ReadToken, SkipLine,
                                WriteString, WriteLn;
    FROM IOResult IMPORT ReadResult, allRight,
                                endOfLine, endOfInput;
    VAR s: ARRAY [1..20] OF CHAR;
BEGIN WHILE ReadResult(infile) # endOfInput
        DO ReadToken(infile,s);
            IF ReadResult(infile) = allRight
            THEN WriteString(outfile,s);
                WriteLn(outfile)
            ELSIF ReadResult(infile) = endOfLine
            THEN SkipLine(infile)
            END
        END
END token.
```

```
DEFINITION MODULE inout;
FROM IOChan IMPORT ChanId;
VAR infile, outfile: ChanId;
END inout.
```

Пример 4:

В следующей программе процедура sum суммирует произвольное количество вещественных чисел, вводимых через подразумеваемый канал ввода и

выводит полученную сумму через подразумеваемый канал вывода. Вводимые числа могут перемежаться концами строчек. Признаком конца ввода служит литера "#". В случае ошибочного ввода (неверная форма, переполнение) также осуществляется суммирование ранее введённых чисел и выдаётся сообщение об ошибке. Эта процедура вызывается программой трижды. При первом вызове подразумеваемые каналы совпадают со стандартными; при втором вызове подразумеваемые каналы совмещены соответственно с библиотечными файлами in.txt и out.txt; при третьем вызове подразумеваемые каналы снова совпадают со стандартными.

```
MODULE sums;
    FROM SWholeIO IMPORT ReadCard, WriteCard;
    FROM SRealIO IMPORT ReadReal, WriteFixed;
    FROM STextIO IMPORT WriteString, ReadChar,
                        SkipLine, WriteLn;
    FROM SIOResult IMPORT ReadResult, allRight,
                          wrongFormat, endOfLine;
    FROM RndFile IMPORT OpenOld, OpenClean,
                        Close;



    FROM StdChans IMPORT StdInChan, StdOutChan,
                         SetInChan, SetOutChan;
```

```
FROM IOChan IMPORT ChanId;
FROM ChanConsts IMPORT old, text, OpenResults;
VAR file1, file2: ChanId; res: OpenResults;
PROCEDURE sum;
    VAR k: CARDINAL; c,s: REAL; ch: CHAR;
          complete,terminate: BOOLEAN;
BEGIN k:=0; s:=0.0; ch:=" ";
        complete:=FALSE; terminate:=FALSE;
        WHILE ~complete & ~terminate
        DO ReadReal(c);
            IF ReadResult() = allRight
            THEN k:=k+1; s:=s+c;
            ELSIF ReadResult() = endOfLine
            THEN SkipLine
            ELSIF ReadResult() = wrongFormat
            THEN ReadChar(ch);
                    IF ch = "#"
                    THEN complete:=TRUE
                    ELSE terminate:=TRUE
                    END
            ELSE terminate:=TRUE
            END
         END;
         WriteLn; WriteString("The sum of  ");
        WriteCard(k,3);
         WriteString("  numbers is  ");
         WriteFixed(s,3,10);
          IF terminate
```

```
                THEN
                  WriteLn;
                   WriteString
                        ("Input interrupted: an error in a data")
                END
    END sum;

BEGIN sum;
    OpenOld(file1,"in.txt",text,res);
    OpenClean(file2,"out.txt",old+text,res);
    SetInChan(file1); SetOutChan(file2);
    sum;
    Close(file1); Close(file2);
    SetInChan(StdInChan()); SetOutChan(StdOutChan());
    sum
END sums.
```

18. ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ И ИХ ОБРАБОТКА

В предыдущих разделах были приведены некоторые возможные случаи, когда во время исполнения программы возбуждаются *исключительные ситуации* (такие исключительные ситуации будем называть *языковыми* — подробнее о них в 18.2). Кроме того, исключительные ситуации могут возбуждаться по усмотрению программиста (подробнее в 18.1). Исключительные ситуации программы могут *обрабатываться*. Исключительная ситуация считается обработанной, если её обработка завершилась.

С каждой сопрограммой (см. 20) в каждый данный момент может быть связан её текущий *обработчик исключительных ситуаций*. А именно, если в этот момент имеется такое полное тело блока **B** с исключительной частью **E**, что

- **B** есть последнее по времени начала исполнения полное тело блока, исполнение которого не завершилось,
- в ходе этого незавершившегося исполнения тела блока **B** не началось исполнение последовательности операторов **E**,

то **E** является в данный момент обработчиком исключительных ситуаций.

Каждая сопрограмма в каждый данный момент находится либо в *нормальном состоянии*, либо в *исключительном состоянии*.

При своём запуске сопрограмма находится в нормальном состоянии. При начале исполнения полного тела блока сопрограмма получает (либо сохраняет) нормальное состояние. После завершения исполнения полного тела блока сопрограмма получает то состояние, в каком она находилась перед началом исполнения этого тела блока.

Исполнение оператора возврата, содержащегося в исключительной части полного тела блока, сопровождается прекращением происходящей в данный момент обработки исключительной ситуации.

В исключительной части полного тела блока и только в ней может содержаться оператор повтора. Исполнение его состоит в следующем:

- состояние исполнения сопрограммы становится нормальным состоянием, и происходящая в данный момент обработка исключительной ситуации прекращается;
- прекращается исполнение этой исключительной части, и начинает исполняться нормальная часть того же тела блока.

Возбуждение исключительной ситуации при исполнении любой части некоторого тела блока приводит к обработке этой исключительной ситуации. При этом возможно имевшая место

обработка любой другой исключительной ситуации прекращается.

Обработка исключительной ситуации состоит в следующем:

- сопрограмма получает (или сохраняет) исключительное состояние;
- если имеется текущий обработчик исключительных ситуаций, являющийся исключительной частью тела **B** некоторого блока, то завершается исполнение нормальной части тела **B** и исполняется данный обработчик исключительных ситуаций;
- иначе, если в данный момент исполняется этап инициализации программного модуля, то исполнение этого этапа завершается;
- иначе, завершается происходящая в данный момент финализация блока программного модуля.

Если исполнение исключительной части тела блока некоторой процедуры или некоторого модуля завершается без исполнения какого-либо непосредственно вложенного в эту исключительную часть оператора возврата или оператора повтора, то это приводит к следующим действиям:

- если данное тело блока есть тело блока процедуры или внутреннего модуля процедуры, то соответствующая процедура финализируется;

- если данная процедура или, соответственно, данный модуль защищены, то осуществляется опускание уровня защиты соответствующей сопрограммы (см. 19);
- вновь начинает происходить обработка данной исключительной ситуации.

Если при исполнении программы исключительная ситуация оказалась необработанной, то в конкретной реализации об этом может быть выдано специальное сообщение.

18.1. Системный модуль EXCEPTIONS

Системный модуль EXCEPTIONS задаёт средства, позволяющие возбуждать исключительные ситуации, а также выяснять, находится ли текущая сопрограмма в исключительном состоянии и какая исключительная ситуация обрабатывается в данный момент.

В модуле определений EXCEPTIONS имеются следующие описания и определения типов и процедур:

```
TYPE ExceptionSource;
  ExceptionNumber = CARDINAL;
```

(Значениями типа ExceptionSource являются *источники исключительных ситуаций*. Для пар операндов этого типа определены операции

"совпадает с" и "не совпадает с" с их обычным представлением и обычным смыслом.

Значения типа ExceptionNumber служат "номерами исключительных ситуаций".)

```
PROCEDURE AllocateSource
                (VAR NewSource: ExceptionSource);
  (Процедура создаёт некоторый новый источник
   исключительных ситуаций и присваивает его
   переменной NewSource.)
PROCEDURE RAISE (source: ExceptionSource;
                 number: ExceptionNumber;
                message: ARRAY OF CHAR);
  (Процедура возбуждает исключительную ситуа-
   цию и связывает с ней тройку значений:
   source — источник исключительных ситуаций,
   которому "принадлежит" данная исключитель-
   ная ситуация, number — номер ситуации в ис-
   точнике и message — "сообщение о ситуации".)
PROCEDURE CurrentNumber
          (source: ExceptionSource): ExceptionNumber;
  (Если текущая сопрограмма находится в исклю-
   чительном состоянии и с обрабатываемой в дан-
   ный момент исключительной ситуацией связан
   источник, являющийся значением параметра
   source, то значением вызова этой процедуры-
   функции является номер данной ситуации. В
```

противном случае возбуждается соответствую щая языковая исключительная ситуация.)

```
PROCEDURE GetMessage
                    (VAR text: ARRAY OF CHAR);
```

(Если текущая сопрограмма находится в исключительном состоянии, то процедура присваивает переменной text определяемый типом этой переменной начальный отрезок строки, являющейся сообщением о ситуации, связанной с обрабатываемой в данный момент исключительной ситуацией и полученной от процедуры RAISE. В противном случае указанной переменной присваивается пустая строка.)

```
PROCEDURE IsCurrentSource
                    (source: ExceptionSource): BOOLEAN;
```

(Если текущая сопрограмма находится в исключительном состоянии и с обрабатываемой в данный момент исключительной ситуацией связан источник, являющийся значением параметра source, то значением вызова этой процедуры-функции является ***истина***, а в противном случае — ***ложь***.)

```
PROCEDURE IsExceptionalExecution (): BOOLEAN;
```

(Если текущая сопрограмма находится в исключительном состоянии, то значением вызова этой процедуры-функции является ***истина***, а в противном случае — ***ложь***.)

18.2. Языковые исключительные ситуации. Системный модуль M2EXCEPTION

Считается, что эффект возбуждения языковой исключительной ситуации равносилен эффекту вызова процедуры EXCEPTIONS.RAISE. При этом смысл параметров определяется следующим образом:

- источник исключительных ситуаций есть некоторый "языковый источник", который представлен либо типом M2Exceptions (см. ниже), либо типами StorageExceptions (см. 17.1.1 и ниже в настоящем пункте) и ProcessesExceptions (см. 22.1);
- считается, что языковые ситуации представляются константами из описания соответствующего типа, так что номерами этих ситуаций в источнике служат порядковые номера значений соответствующих констант;
- сообщение о ситуации определяется языком и реализацией.

Системный модуль M2EXCEPTION задаёт средства, позволяющие определять, обрабатывается ли в данный момент языковая исключительная ситуация, принадлежащая источнику M2Exceptions, и какая именно.

В модуле определений M2EXCEPTION имеются следующие описания и определения типов и процедур:

```
TYPE M2Exceptions =
   (indexException,        rangeException,
    caseSelectException,   invalidLocation,
    functionException,     wholeValueException,
    wholeDivException,     realValueException,
    realDivException,      complexValueException,
    complexDivException, protException,
    sysException,          coException,
    exException   );
```

```
PROCEDURE M2Exception (): M2Exceptions;
```

(Если в данный момент обрабатывается некоторая языковая исключительная ситуация, принадлежащая источнику M2Exceptions, то значением вызова этой функции является значение константы, представляющей эту языковую ситуацию. В противном случае возбуждается языковая исключительная ситуация exException.)

```
PROCEDURE IsM2Exception (): BOOLEAN;
```

(Эффект вызова этой функции равносилен эффекту вызова функции IsCurrentSource из модуля EXCEPTIONS с фактическим параметром — языковым источником M2Exceptions.)

Приведём сейчас ранее пропущенные описания (определения) из стандартного модуля Storage.

```
TYPE StorageExceptions = (nilDeallocation,
                          pointerToUnallocatedStorage,
                          wrongStorageToUnallocate);
```

(Три исключительные ситуации, принадлежащие источнику,представляемому данным типом, возбуждаются при исполнении вызова процедуры DEALLOCATE. Ситуация nilDeallocation возбуждается в том случае, если значение первого параметра указанного вызова есть ***псевдоимя***; ситуация pointerToUnallocatedStorage возбуждается в том случае, если подлежащая освобождению память не была выделена при исполнении процедуры ALLOCATE; ситуация wrongStorageToUnallocate возбуждается в том случае, если размер памяти, подлежащей освобождению, не совпадает с размером памяти, отведённой ранее процедурой ALLOCATE.)

```
PROCEDURE StorageException (): StorageExceptions;
```

(Если в данный момент обрабатывается некоторая языковая исключительная ситуация, принадлежащая источнику StorageExceptions, то значением вызова этой функции является эта исключительная ситуация. В противном случае возбуждается языковая исключительная ситуация exException.)

```
PROCEDURE IsStorageException (): BOOLEAN;
```
(Эффект вызова этой функции равносилен эффекту вызова функции IsCurrentSource из модуля EXCEPTIONS с фактическим параметром — языковым источником StorageExceptions.)

18.3. Примеры на исключительные ситуации

Пример 1:

В программном модуле exc1 исполняется вызов процедуры count, импортированной из модуля countmod. Если в ходе исполнения этой процедуры была возбуждена некоторая языковая исключительная ситуация, то исполнение программы прекращается с выдачей на печать определяемого реализацией сообщения о данной языковой ситуации (например, "zero divisor".)

```
MODULE exc1;
  FROM STextIO IMPORT WriteString;
  FROM EXCEPTIONS IMPORT GetMessage;
  FROM countmod IMPORT count;
  VAR text: ARRAY [1 .. 30] OF CHAR;
BEGIN count
EXCEPT GetMessage(text); WriteString(text); RE-
TURN
END exc1.
```

```
DEFINITION MODULE countmod;
   PROCEDURE count;
END countmod.
```

Пример 2 (схематический):

В программный модуль exc2 импортируются процедуры p1, ..., p9 из модуля определений P. В зависимости от конкретного содержимого модуля реализации P в той или иной из этих процедур может быть возбуждена исключительная ситуация, сообщение о которой содержит идентификатор соответствующей процедуры. (Исключительные ситуации возбуждаются с помощью процедуры rs, импортируемой в модуль реализации P из модуля etmod.) Эффект исполнения данной программы будет состоять в выдаче "протокола трассировки", то есть сообщения о том, какие участки программы последовательно пройдены и при каком состоянии программы.

```
MODULE exc2;
   FROM P IMPORT p2,p3,p4,p5,p6,p7,p8,p9;
   FROM etmod IMPORT t,B;
   MODULE m1; IMPORT t,p2,p3,p4,p5,p6,p7;
      MODULE m2; IMPORT t,p2,p3,p4,p5,p6;
         PROCEDURE proc1;
            MODULE mp; IMPORT t,p2;
            BEGIN t(5); p2; t(6)
```

```
            FINALLY t(15) EXCEPT t(16)
             END mp;
             PROCEDURE proc2;
             BEGIN t(8); p3; t(9)
            EXCEPT t(10); p4; t(11); RETURN
             END proc2;
          BEGIN t(7); proc2; t(12); p5; t(13)
         EXCEPT t(14)
          END proc1;
       BEGIN t(4); proc1; t(17); p6; t(18)
      EXCEPT t(19); RETURN
       FINALLY t(32) EXCEPT t(33)
       END m2;
   BEGIN t(20); p7; t(21) EXCEPT t(22); RETURN
   FINALLY t(30) EXCEPT t(31)
   END m1;
BEGIN t(23); p8; t(24)
EXCEPT IF B THEN B:=FALSE; t(25); RETRY END;
          t(26)
FINALLY t(27); p9; t(28) EXCEPT t(29);RETURN
END exc2.

DEFINITION MODULE etmod;
   PROCEDURE t(i: INTEGER);
   PROCEDURE rs(x: ARRAY OF CHAR);
   VAR B: BOOLEAN;
END etmod.
```

```
IMPLEMENTATION MODULE etmod;
   FROM P IMPORT p1;
   FROM EXCEPTIONS
      IMPORT ExceptionSource,GetMessage,RAISE,
             AllocateSource,IsExceptionalExecution;
   FROM STextIO IMPORT WriteString;
   FROM SWholeIO IMPORT WriteInt;
   VAR s: ExceptionSource;
       x: ARRAY [1..10] OF CHAR;
   PROCEDURE t(i: INTEGER);
   BEGIN GetMessage(x); WriteString(x); WriteInt(i,0);
            IF IsExceptionalExecution()
            THEN WriteString(":Ex")
           ELSE WriteString(":Nr")
            END;
           WriteString("/")
   END t;
   PROCEDURE rs(x: ARRAY OF CHAR);
   BEGIN RAISE(s,0,x)
   END rs;
BEGIN AllocateSource(s); B:=TRUE; t(1); p1; t(2)
        EXCEPT t(3); RETURN
FINALLY t(34)
END etmod.
```

```
DEFINITION MODULE P;
   PROCEDURE p1; PROCEDURE p2;
   PROCEDURE p3; PROCEDURE p4;
```

```
    PROCEDURE p5; PROCEDURE p6;
    PROCEDURE p7; PROCEDURE p8;
    PROCEDURE p9;
END P.
```

Ниже будут приведены протоколы трассировок для различных случаев возбуждения исключительных ситуаций и краткие пояснения к ним.

0) Исключительные ситуации не возбуждены.

1:Nr/ 2:Nr/ 4:Nr/ 5:Nr/ 6:Nr/ 7:Nr/ 8:Nr/ 9:Nr/ 12:Nr/ 13:Nr/ 15:Nr/ 17:Nr/ 18:Nr/ 20:Nr/ 21:Nr/ 23:Nr/ 24:Nr/ 27:Nr/ 28:Nr/ 30:Nr/ 32:Nr/ 34:Nr/

Порядок прохождения участков программы соответствует порядку исполнения модулей.

1) Исключительная ситуация возбуждена в процедуре p1.

1:Nr/ p1 3:Ex/ 4:Nr/ 5:Nr/ 6:Nr/ 7:Nr/ 8:Nr/ 9:Nr/ 12:Nr/ 13:Nr/ 15:Nr/ 17:Nr/ 18:Nr/ 20:Nr/ 21:Nr/ 23:Nr/ 24:Nr/ 27:Nr/ 28:Nr/ 30:Nr/ 32:Nr/ 34:Nr/

Возбуждённая исключительная ситуация обработана в исключительной части модуля реализации etmod.

2) Исключительная ситуация возбуждена в процедуре p2.

1:Nr/ 2:Nr/ 4:Nr/ 5:Nr/ p2 19:Ex/ 20:Nr/ 21:Nr/ 23:Nr/ 24:Nr/ 27:Nr/ 28:Nr/ 30:Nr/ 32:Nr/ 34:Nr/

В момент возбуждения указанной ситуации исполняются инициализирующее тело модуля mp и

объемлющее его инициализирующее тело модуля m2. Первое из них является неполным, а второе — полным. Его исключительная часть и служит текущим обработчиком исключительных ситуаций.

3) Исключительная ситуация возбуждена в процедуре p3.

1:Nr/ 2:Nr/ 4:Nr/ 5:Nr/ 6:Nr/ 7:Nr/ 8:Nr/ p3 10:Ex/ p3 11:Ex/ 12:Nr/ 13:Nr/ 15:Nr/ 17:Nr/ 18:Nr/ 20:Nr/ 21:Nr/ 23:Nr/ 24:Nr/ 27:Nr/ 28:Nr/ 30:Nr/ 32:Nr/ 34:Nr/

Обработка исключительной ситуации завершена в самой процедуре proc2.

4) Исключительные ситуации возбуждены в процедурах p3 и p4.

```
1:Nr/ 2:Nr/ 4:Nr/ 5:Nr/ 6:Nr/ 7:Nr/ 8:Nr/ p3 10:Ex/ p4
14:Ex/ 15:Nr/ p4 19:Ex/ 20:Nr/ 21:Nr/ 23:Nr/ 24:Nr/
27:Nr/ 28:Nr/ 30:Nr/ 32:Nr/ 34:Nr/
```

При возбуждении второй исключительной ситуации обработка первой ситуации прекращается. Заметим, что процедура proc2 исполняется в ходе исполнения процедуры proc1, блок которой содержит исключительную часть. Последняя и начинает обрабатывать данную ситуацию, но за отсутствием операторов возврата или повтора завершить обработку не может. Поэтому происходит финализация содержащегося в процедуре proc1 модуля mp и поиск нового обработчика исключительных ситуаций, каковым оказывается исключительная часть инициирующего тела модуля m2.

5) Исключительная ситуация возбуждена в процедуре p5.

```
1:Nr/ 2:Nr/ 4:Nr/ 5:Nr/ 6:Nr/ 7:Nr/ 8:Nr/ 9:Nr/ 12:Nr/ p5
14:Ex/ 15:Nr/ p5 19:Ex/ 20:Nr/ 21:Nr/ 23:Nr/ 24:Nr/
27:Nr/ 28:Nr/ 30:Nr/ 32:Nr/ 34:Nr/
```

Обработка исключительной ситуации завершается непосредственно в процедуре proc1, после чего происходит то же, что и в предыдущем случае.

6) Исключительная ситуация возбуждена в процедуре p6.

1:Nr/ 2:Nr/ 4:Nr/ 5:Nr/ 6:Nr/ 7:Nr/ 8:Nr/ 9:Nr/ 12:Nr/
13:Nr/ 15:Nr/ 17:Nr/ p6 19:Ex/ 20:Nr/ 21:Nr/ 23:Nr/
24:Nr/ 27:Nr/ 28:Nr/ 30:Nr/ 32:Nr/ 34:Nr/

После безмятежного завершения вызова процедуры proc1 модуль m2 завершает обработку исключительной ситуации в исключительной части инициирующего тела своего собственного блока.

7) Исключительная ситуация возбуждена в процедуре p7.

1:Nr/ 2:Nr/ 4:Nr/ 5:Nr/ 6:Nr/ 7:Nr/ 8:Nr/ 9:Nr/ 12:Nr/
13:Nr/ 15:Nr/ 17:Nr/ 18:Nr/ 20:Nr/ p7 22:Ex/ 23:Nr/
24:Nr/ 27:Nr/ 28:Nr/ 30:Nr/ 32:Nr/ 34:Nr/

Обработка ситуации завершена внутри модуля m1.

8) Исключительная ситуация возбуждена в процедуре p8.

1:Nr/ 2:Nr/ 4:Nr/ 5:Nr/ 6:Nr/ 7:Nr/ 8:Nr/ 9:Nr/ 12:Nr/
13:Nr/ 15:Nr/ 17:Nr/ 18:Nr/ 20:Nr/ 21:Nr/ 23:Nr/ p8
25:Ex/ 23:Nr/ p8 26:Ex/
#RTS: unhandled exception #0: p8
27:Nr/ 28:Nr/ 30:Nr/ 32:Nr/ 34:Nr/

Ситуация, возбуждённая в нормальной части инициирующего тела блока программного модуля, обрабатывается в исключительной части этого тела. Так как текущее значение переменной B есть ***истина***, исполняется оператор повтора. Это означает, что нормальная часть тела вновь начинает исполняться в нормальном состоянии. Снова

возбуждается та же исключительная ситуация, и снова она обрабатывается в той же исключительной части. Но значение B на этот раз есть ***ложь***, и оператор повтора не исполняется. Так как не исполняется также и оператор возврата, и так как не может быть найден в данный момент обработчик исключительных ситуаций, то исключительная ситуация остаётся необработанной, инициирование программного модуля завершается и начинается его финализация. Конкретная реализация может выдать специальную информацию о происшедшем событии (например, ту, которая приведена выше и отмечена полужирным шрифтом).

9) Исключительная ситуация возбуждена в процедуре p9.

1:Nr/ 2:Nr/ 4:Nr/ 5:Nr/ 6:Nr/ 7:Nr/ 8:Nr/ 9:Nr/ 12:Nr/ 13:Nr/ 15:Nr/ 17:Nr/ 18:Nr/ 20:Nr/ 21:Nr/ 23:Nr/ 24:Nr/ 27:Nr/ p9 29:Ex/ 30:Nr/ 32:Nr/ 34:Nr/

Обработка ситуации завершается внутри финализирующего тела блока программного модуля, после чего финализируются остальные модули.

Пример 3:

В программном модуле exc3 исполняется 100 раз подряд импортируемая из модуля prcstext процедура process. Имеется в виду, что эта процедура моделирует некоторый процесс, реально происходящий в разных условиях, в том числе и

приводящих к возбуждению определённых исключительных ситуаций. Все эти исключительные ситуации связаны с одним и тем же источником s и имеют своими номерами числа от 1 до 10. Если в ходе очередного исполнения процесса возбуждается исключительная ситуация, то это исполнение завершается. В конце своей работы программа печатает таблицу, которая для каждого номера исключительной ситуации указывает, сколько раз ситуация с данным номером была возбуждена.

```
MODULE exc3;
   FROM SWholeIO IMPORT WriteCard;
   FROM EXCEPTIONS IMPORT CurrentNumber;
   FROM prcstext IMPORT s,process;
   VAR i: CARDINAL;
         table: ARRAY [1 .. 10] OF CARDINAL;
   PROCEDURE processing;
      VAR k: CARDINAL;
   BEGIN process
   EXCEPT k:=CurrentNumber(s); table[k]:=table[k]+1;
            RETURN
   END processing;
BEGIN FOR i:=1 TO 10 DO table[i]:=0 END;
         FOR i:=1 TO 100 DO processing END;
         FOR i:=1 TO 10 DO WriteCard(table[i],0) END
END exc3.
```

```
DEFINITION MODULE prcstext;
   FROM EXCEPTIONS IMPORT ExceptionSource;
   PROCEDURE process;
   VAR s: ExceptionSource;
END prcstext.
```

19. ПРЕРЫВАНИЯ И ЗАЩИТА ОТ ПРЕРЫВАНИЙ

В отличие от исключительных ситуаций (языковых и задаваемых программистом) такие происходящие во время исполнения программы события, как *прерывания*, порождаются аппаратурой или операционной системой. Каждое конкретное прерывание принадлежит какому-то, определяемому реализацией, классу возможных прерываний — *источнику прерываний*. Источники прерываний являются значениями типа

CORОUTINES.INTERRUPTSOURCE

(см. 20).

При возникновении некоторого конкретного прерывания **I** программа получает *запрос на прерывание* **I**. Запрос этот подлежит, вообще говоря, *обработке* (см. 20, пояснения к процедуре IOTRANSFER). Запрос "находится" в программе с момента его получения до момента начала его обработки.

Имеющийся запрос может (в разные моменты времени) быть доступным или недоступным для обработки. Для управления указанной доступностью/недоступностью (по-другому — для управления "*защитой от прерываний*") как раз и используются значения типа защиты. (Заметим при этом, что только конкретная реализация определяет эффект получения запроса на прерывание, принадлежащее некоторому источнику прерываний, при наличии недоступного для обработки запроса на прерывание, принадлежащее тому же источнику. В частности, новый запрос может продолжить своё существование наряду с прежним или же прежний запрос может быть "аннулирован".)

Конкретная реализация сопоставляет каждому значению типа защиты некоторое множество источников прерываний, *маскируемых* этим значением. Рассматриваемое сопоставление должно удовлетворять следующим условиям:

- значение INTERRUPTIBLE не маскирует никакой источник прерываний;
- значение UNINTERRUPTIBLE маскирует все источники прерываний;
- если **P1** и **P2** суть значения типа защиты, **P1** $\leq$ **P2,** и **P1** маскирует некоторый источник прерываний, то и **P2** маскирует тот же источник.

Запрос на прерывание, принадлежащее источнику прерываний, замаскированному некоторым значением типа защиты, будем также считать запросом, замаскированным тем же значением.

Если программный модуль, модуль реализации или описание локального модуля содержит задание уровня защиты, то соответствующий модуль, а также все процедуры, описанные во вступительной части блока этого модуля, считаются *защищёнными*, а значение задания уровня защиты является *уровнем защиты* как данного модуля, так и всех данных процедур. Если в некоторый момент исполнения программы исполнение это происходит в такой точке программы, которая находится в некотором защищённом модуле или в теле некоторой защищённой процедуры, и при этом не происходит исполнение никакого содержащегося в этих конструкциях модуля или вызова процедуры, то данный момент исполнения программы также считается защищённым.

Будем рассматривать далее для каждой сопрограммы (см. раздел 20) её текущий *уровень защиты*, являющийся значением типа защиты. В случае главной сопрограммы начальный уровень защиты задаётся конкретной реализацией, а в остальных случаях он определяется по правилам, изложенным в указанном разделе. В дальнейшем текущий уровень защиты переопределяется при

подъёме и опускании уровня защиты, а также при вызове процедуры COROUTINES.LISTEN (см. тот же раздел).

Подъём уровня защиты осуществляется следующим образом:

- если каждый источник прерываний, маскируемый текущим уровнем защиты, маскируется также уровнем защиты того защищённого модуля или той защищённой процедуры, во время исполнения которых осуществляется подъём, то текущим значением уровня защиты становится уровень защиты данного модуля или данной процедуры;
- в противном случае возбуждается исключительная ситуация.

Опускание уровня защиты восстанавливает текущий уровень защиты, который имел место перед началом исполнения соответствующего тела блока того защищённого модуля или той защищённой процедуры, во время исполнения которых осуществляется опускание.

Переходим, наконец, к основному в настоящем разделе:

- если в некоторый незащищённый момент исполнения программы имеются какие-либо необработанные запросы на прерывания или если в некоторый защищённый момент исполнения программы имеются запросы на

прерывания, не замаскированные текущим уровнем защиты текущей сопрограммы, то в этот момент времени указанные запросы доступны для обработки;

- если в некоторый защищённый момент исполнения программы имеются запросы на прерывания, замаскированные текущим уровнем защиты текущей сопрограммы, то в этот момент времени указанные запросы недоступны для обработки.

20. СОПРОГРАММЫ. СИСТЕМНЫЙ МОДУЛЬ COROUTINES

В программе с помощью средств, предоставляемых системным модулем COROUTINES, могут создаваться особые объекты, называемые *сопрограммами*. При создании сопрограммы с ней связываются:

- *идентификация* сопрограммы;
- некоторая процедура типа PROC, описанная в каком-либо статическом модуле программы;
- некоторый участок памяти — *рабочая область* сопрограммы.

Одна и та же процедура может быть связана с несколькими сопрограммами, а рабочие области разных сопрограмм могут пересекаться. Каждая

программа имеет также обязательную *главную сопрограмму*, создаваемую в момент начала исполнения программы. С главной сопрограммой не связана никакая процедура, а рабочая область её определяется реализацией.

С сопрограммой в некоторые моменты исполнения программы может быть связан какой-либо источник прерываний. Один и тот же источник прерываний не может быть связан с разными сопрограммами.

Сопрограммы подлежат, как правило, исполнению. Начало исполнения сопрограммы считается её *запуском*. Исполнение главной сопрограммы состоит в исполнении самой программы. Исполнение любой другой сопрограммы состоит в исполнении вызова связанной с ней процедуры. Исполняемая сопрограмма может быть *приостановлена*, а затем *возобновлена*. Приостановка сопрограммы означает приостановку её исполнения, а возобновление сопрограммы — продолжение этого исполнения с той точки, на которой оно было приостановлено. (Сопрограмму, исполняемую в некоторый данный момент, будем также называть "текущей сопрограммой".) Рабочая область сопрограммы используется для размещения данных, возникающих при исполнении сопрограммы. Если при исполнении сопрограммы завершается исполнение тела блока процедуры, связанной с

сопрограммой, то эта процедура финализируется, после чего возбуждается исключительная ситуация.

В модуле определений COROUTINES имеются следующие описания и определения типов и процедур:

```
TYPE COROUTINE;
```
(Значениями этого типа являются идентификации сопрограмм.)

```
TYPE INTERRUPTSOURCE;
```
(Значениями этого, определяемого реализацией, типа являются источники прерываний.)

```
PROCEDURE NEWCOROUTINE
  (procBody: PROC; workspace: SYSTEM.ADDRESS;
   size: CARDINAL; VAR cr: COROUTINE;
   initProtection: PROTECTION);
```
(Вызов этой процедуры создает новую сопрограмму, идентификация которой присваивается переменной cr. С новой сопрограммой связывается процедура, являющаяся значением параметра procBody, и рабочая область, у которой логический адрес первой ячейки есть значение параметра workspace, а число ячеек — значение параметра size. Если это число ячеек оказывается меньшим, чем задаваемый реализацией минимальный размер рабочей области, то возбуждается исключительная ситуация. Началь-

ным уровнем защиты новой сопрограммы становится значение параметра initProtection.

Вызов данной процедуры может иметь и другой формат — без последнего фактического параметра. При применении этого варианта вызова начальным уровнем защиты новой сопрограммы становится текущий уровень защиты текущей сопрограммы.)

```
PROCEDURE TRANSFER (VAR from: COROUTINE;
                    to: COROUTINE);
```

(В результате вызова этой процедуры идентификация текущей сопрограммы присваивается переменной from, текущая сопрограмма приостанавливается, а сопрограмма, идентификация которой есть значение параметра to, возобновляется или запускается в зависимости от того, исполнялась она ранее или нет.)

```
PROCEDURE ATTACH
                (source: INTERRUPTSOURCE);
```

(Вызов этой процедуры связывает с текущей сопрограммой источник прерываний, являющийся значением параметра source. Если данный источник прерываний был связан с некоторой другой сопрограммой, то эта последняя связь устраняется.)

```
PROCEDURE DETACH
                (source: INTERRUPTSOURCE);
```

```
    (Если источник прерываний, являющийся значе-
      нием параметра source, связан с текущей сопро-
      граммой, то в результате вызова этой процеду-
      ры данная связь устраняется. В противном
      случае вызов процедуры не производит никако-
      го эффекта.)
PROCEDURE IsATTACHED
          (source: INTERRUPTSOURCE): BOOLEAN;
    (Значение вызова этой функции есть истина в
      том и только том случае, если источник преры-
      ваний, являющийся значением переменной
      source, связан в данный момент с некоторой
      сопрограммой.)
PROCEDURE HANDLER
        (source: INTERRUPTSOURCE): COROUTINE;
    (Если источник прерываний, являющийся значе-
      нием переменной source, связан с некоторой
      сопрограммой, то значение вызова этой функ-
      ции есть идентификация этой сопрограммы, а в
      противном случае указанный результат неопре-
      делён и может быть возбуждена исключительная
      ситуация.)
PROCEDURE IOTRANSFER
          (VAR from: COROUTINE; to: COROUTINE);
    (Пусть текущая сопрограмма есть C. Если с ней
      не связан никакой источник прерываний, то при
```

вызове данной процедуры возбуждается исключительная ситуация.

В противном случае в результате вызова данной процедуры идентификация сопрограммы **C** присваивается переменной from, сопрограмма **C** приостанавливается, а сопрограмма, идентификация которой есть значение параметра to, возобновляется или запускается в зависимости от того, исполнялась она ранее или нет.

Если в какой-либо ближайший момент времени, но до возобновления сопрограммы **C**, будут доступны для обработки запросы на прерывания, принадлежащие источникам прерываний, связанным с сопрограммой **C**, то идентификация текущей сопрограммы присваивается переменной from, текущая сопрограмма приостанавливается, а сопрограмма **C** возобновляется. Тем самым начинается обработка одного из этих запросов. Указать конец обработки в строго формальном смысле вряд ли возможно, но разумно считать им ближайший вызов процедуры IOTRANSFER или TRANSFER.

Для достижения целей, преследуемых рассматриваемым механизмом, желательно, чтобы по крайней мере тот фрагмент тела процедуры, связанной с сопрограммой **C**, который охватывает вызов процедуры IOTRANSFER и операторы,

осуществляющие указанную обработку, был защищён от прерываний.)

PROCEDURE LISTEN (p: PROTECTION);

(Исполнение этой процедуры состоит из следующих двух шагов:

- текущим уровнем защиты текущей сопрограммы становится значение параметра **p**;
- восстанавливается тот текущий уровень защиты текущей сопрограммы, который имел место перед началом исполнения процеду ры.)

PROCEDURE CURRENT (): COROUTINE;

(Значением вызова этой процедуры является идентификация текущей сопрограммы.)

PROCEDURE PROT (): PROTECTION;

(Значением вызова этой процедуры является текущий уровень защиты текущей сопрограммы.)

Примеры:

a)
```
MODULE crt [INTERRUPTIBLE];
  FROM SYSTEM IMPORT ADDRESS;
  FROM Storage IMPORT ALLOCATE;
  FROM COROUTINES
    IMPORT COROUTINE, NEWCOROUTINE,
           TRANSFER,IOTRANSFER,
           INTERRUPTSOURCE, ATTACH;
  FROM forwork IMPORT work,elaborate,intval;
  VAR hnd1,hnd2,t,s: COROUTINE;
```

```
      A,B: ADDRESS; is: INTERRUPTSOURCE;
      PROCEDURE h;
      BEGIN ATTACH(is); s:=t;
            LOOP IOTRANSFER(t,s); elaborate
            END
      END h;
  BEGIN  ALLOCATE (A,10000);
     ALLOCATE(B,10000);
      NEWCOROUTINE
           (h,A,10000,hnd1,UNINTERRUPTIBLE);
      NEWCOROUTINE
          (h,B,10000,hnd2,UNINTERRUPTIBLE);
      is:=intval(1);TRANSFER(t,hnd1);
      is:=intval(2);TRANSFER(t,hnd2);
      work
   END crt.


 DEFINITION MODULE forwork;
    FROM COROUTINES
                  IMPORT INTERRUPTSOURCE;
    PROCEDURE work;
    PROCEDURE elaborate;
    PROCEDURE intval(x: CARDINAL):
                              INTERRUPTSOURCE;
END forwork.
```

Исполнение модуля crt начинается с того, что создаются две сопрограммы hnd1 и hnd2, связанные

с одной и той же процедурой h и имеющие непересекающиеся рабочие области. Уровень защиты главной сопрограммы — минимальный, а остальных сопрограмм — максимальный. Затем запускается сопрограмма hnd1, которая связывает с собой некоторый источник прерываний, задаваемый импортируемой функцией intval, и после этого возобновляет главную сопрограмму. Главная сопрограмма запускает вторую из созданных сопрограмм, которая связывает с собой другой источник прерываний, после чего возобновлённая в очередной раз главная сопрограмма берётся за свою главную работу — исполнение импортированной процедуры work. Если в ходе последнего исполнения не возникнет прерываний, то дальнейших комментариев к программе не потребуется. Но допустим, что возникло прерывание, связанное с сопрограммой hnd1 или hnd2. Так как уровень защиты главной сопрограммы не маскирует никакие запросы на прерывания, то возобновляется соответствующая сопрограмма и импортируемая процедура elaborate осуществляет обработку соответствующего запроса. По окончании этой обработки процедура IOTRANSFER возобновляет исполнение главной сопрограммы, то есть процедура work продолжается с того места, на котором она остановилась. Поскольку сопрограммы "ловят"

прерывания с помощью бесконечного цикла, такие прерывания (и с любым порядком чередования) могут встретиться и быть обработанными в любом количестве. Прерывания же, отличные от двух рассматриваемых, не будут никогда обработаны, что, вообще говоря, может привести к неопределённости в работе программы.

б)
```
MODULE lst [INTERRUPTIBLE];
    FROM SYSTEM IMPORT ADDRESS;
    FROM Storage IMPORT ALLOCATE;
    FROM STextIO IMPORT WriteString;
    FROM SWholeIO IMPORT WriteCard;
    FROM COROUTINES
      IMPORT COROUTINE,NEWCOROUTINE,
                TRANSFER,IOTRANSFER,
              LISTEN;
    FROM forel IMPORT elaborate,count;
    FROM forins IMPORT ins,attach_all;
    VAR hnd,t,s: COROUTINE; A: ADDRESS;
        i: CARDINAL;
    PROCEDURE h;
    BEGIN  attach_all; s:=t;
           LOOP IOTRANSFER(t,s);elaborate
           END
    END h;
  BEGIN  ALLOCATE(A,10000);
    NEWCOROUTINE
```

```
            (h,A,10000,hnd,UNINTERRUPTIBLE);
    TRANSFER(t,hnd); ins;
        FOR i:=1 TO 10
        DO LISTEN(INTERRUPTIBLE)
        END;
        WriteString("There occurred ");
        IF count=10
       THEN WriteString("no less than")
       END;
        WriteCard(count,0);
       WriteString(" interruptions.")
END lst.


DEFINITION MODULE forel;
    FROM COROUTINES
                IMPORT INTERRUPTSOURCE;
    VAR count: CARDINAL;
     PROCEDURE elaborate;
END forel.


IMPLEMENTATION MODULE forel;
    FROM COROUTINES
                IMPORT INTERRUPTSOURCE;
    PROCEDURE elaborate;
    BEGIN count:=count+1
    END elaborate;
```

```
BEGIN count:=0
END forel.
DEFINITION MODULE forins;
    FROM COROUTINES
                IMPORT INTERRUPTSOURCE;
    PROCEDURE ins;
    PROCEDURE attach_all;
END forins.
```

Будем считать, что импортируемая из модуля forins процедура attach_all, используя нужное число раз процедуру ATTACH, связывает с текущей сопрограммой (в данном случае с сопрограммой hnd) все имеющиеся в системе источники прерываний. Будем также считать, что модуль реализации forins снабжён заданием уровня защиты UNINTERRUPTIBLE. Поэтому и процедура ins, импортируемая из этого модуля, имеет максимальный уровень защиты, так что запросы на прерывания, которые могут быть получены в ходе исполнения указанной процедуры, не приводят, в отличие от того, что имело место в предыдущем примере, к возобновлению сопрограммы hnd и исполнению процедуры elaborate, а просто накапливаются в качестве недоступных для обработки. После исполнения ins произойдёт десятикратное

обращение к процедуре LISTEN. После первого шага каждого исполнения этой процедуры все имеющиеся запросы станут доступными для обработки и одна из них будет обработана возобновлённой сопрограммой hnd, то есть будет добавлена единица к счётчику count. После этого очередное исполнение процедуры LISTEN возобновится со второго шага (и оставшиеся запросы снова окажутся недоступными). Если при исполнении LISTEN запросы на прерывание отсутствуют, то исполнение это не произведёт никакого эффекта. В итоге программа выдаст число имевших место прерываний (от 0 до 9) или сообщит, что их было не меньше 10.

21. СИСТЕМНЫЙ МОДУЛЬ TERMINATION

Системный модуль TERMINATION задаёт средства, позволяющие определять, начался ли процесс завершения работы программы.

В модуле определений TERMINATION имеются следующие определения процедур.

PROCEDURE IsTerminating (): BOOLEAN;

(Значение вызова этой функции есть ***истина*** в том и только том случае, если началась финализация программного модуля.)

PROCEDURE HasHalted (): BOOLEAN;

(Значение вызова этой функции есть ***истина*** в том и только том случае, если к моменту этого вызова имело место исполнение вызова процедуры HALT.)

22. ДОПОЛНИТЕЛЬНЫЕ СТАНДАРТНЫЕ МОДУЛИ ЯЗЫКА

22.1. Процессы. Стандартный модуль Processes

Вместо механизма сопрограмм может использоваться более развёрнутый механизм *процессов*, предоставляемый стандартным модулем Processes. (Надо иметь в виду, что использование в одной и той же программе как средств, предоставляемых модулем COROUTINES, так и средств, предоставляемых модулем Processes, может в некоторых реализациях привести к непредсказуемости.)

Процессы суть объекты, каждый из которых единожды создаётся и уничтожается в ходе исполнения программы. При создании процесса с ним связываются:

- идентификация процесса;
- некоторая процедура типа PROC;
- целое число, называемое *срочностью* процесса;

- размер выделяемой реализацией *рабочей области* процесса;
- так называемый "параметр" процесса — логический адрес начальной ячейки некоторого участка памяти, используемого для хранения связанной с процессом информации.

Одна и та же процедура может быть связана с несколькими процессами. Каждая программа имеет также обязательный *главный процесс*, создаваемый в момент начала исполнения программы. С главным процессом не связана никакая процедура, его срочность есть 0, а размер рабочей области и "параметр" определяются реализацией.

С процессом в некоторые моменты исполнения программы могут быть связаны те или иные *источники событий*, то есть неотрицательные целые числа, сопоставленные тем или иным возможным *событиям*. Природа этих событий, их список, числа, сопоставляемые конкретным событиям, — всё это определяется конкретной реализацией. Существенно здесь для нас лишь то, что в тот или иной момент некоторое событие может произойти и вызвать некую реакцию. Один и тот же источник событий не может быть связан с разными процессами. Если в программе использован источник событий, которому реализацией не сопоставлено никакое событие, то дальнейшее исполнение программы может оказаться неопределённым.

Процессы подлежат, как правило, исполнению. Начало исполнения процесса считается его *запуском*. Исполнение главного процесса состоит в исполнении самой программы. Исполнение любого другого процесса состоит в исполнении вызова связанной с ним процедуры. Исполняемый процесс может быть *приостановлен*, а затем *возобновлён*. Приостановка процесса означает приостановку его исполнения, а возобновление процесса — продолжение этого исполнения с той точки, на которой оно было приостановлено. Будем говорить, что исполняемый процесс находится в *состоянии исполнения*. В любой момент в состоянии исполнения может находиться не более одного процесса. Рабочая область процесса используется для размещения данных, возникающих при его исполнении. По завершении своего исполнения процесс уничтожается. Эффект уничтожения главного процесса при существовании каких-либо других (готовых к исполнению — см. ниже) процессов программы определяется конкретной реализацией.

Если процесс в некоторый момент существует и не находится в состоянии исполнения, то он находится либо в *состоянии готовности*, либо в *состоянии ожидания*, либо в *пассивном состоянии*. Переход процесса из одного состояния в другое осуществляется либо программными средствами (о которых речь будет идти при рассмотрении процедур

модуля Processes), либо средствами непрограммными. К последним относятся действия диспетчера (часть операционной системы) и реакции на события.

Диспетчер осуществляет следующие обязательные действия:

- если ни один из процессов не находится в состоянии исполнения и имеются процессы, находящиеся в состоянии готовности, причём с последними связаны срочности, максимальная из которых есть **U**, то один из тех процессов, с которыми связана срочность **U**, переводится в состояние исполнения, то есть — в зависимости от того, исполнялся он ранее или нет — запускается или возобновляется;
- если имеется находящийся в состоянии исполнения процесс, с которым связана срочность **U**, и имеется находящийся в состоянии готовности процесс, с которым связана срочность **U1**, большая чем **U**, то второй процесс переводится в состояние исполнения, а первый процесс приостанавливается и переводится в состояние готовности.

(Стандарт не требует, чтобы диспетчер осуществлял эти свои обязательные действия сразу же, как к этому представляется возможность. Однако не каждая операционная система способна подчиниться столь жёсткому требованию Стандарта, и поэтому в

некоторых реализациях между переходом в состояние готовности процесса **P1** со срочностью, превышающей срочность исполняемого процесса **P0**, и переходом процесса **P1** в состояние исполнения может продолжать исполняться процесс **P0**. Такой эффект будем называть "задержкой диспетчера". Чтобы избежать последствий этого эффекта можно пользоваться средствами модуля Semaphores, о которых будет идти речь в 22.2.)

Кроме того, диспетчер может — в соответствии с особенностями конкретной реализации и операционной системы (наличие "диспетчеризации с вытеснением") — заменять взаимно состояние исполнения и состояние готовности у двух процессов, с которыми связана одна и та же срочность.

Реакция на происшедшее событие переводит в состояние готовности тот процесс (если таковой имеется и находится в состоянии ожидания), с которым связан соответствующий источник событий.

Действие диспетчера и реакция на событие не могут иметь место во время исполнения вызова какой-либо из процедур модуля Processes.

В модуле определений Processes имеются следующие описания и определения типов и процедур (в предположении, что идентификатор ADDRESS подвергнут неквалифицируемому импорту из модуля SYSTEM в данный модуль):

```
TYPE ProcessId;
```

```
    (Значениями этого типа являются идентификации
     процессов.)
TYPE
  Parameter = ADDRESS;
  Body = PROC;
  Urgency = INTEGER;
  Sources = CARDINAL;
  ProcessesExceptions =
                          (passiveProgram, processError);
    (Исключительная ситуация passiveProgram воз-
     буждается в том случае, если существует хотя
     бы один процесс и все существующие процессы
     находятся в пассивном состоянии. Исключи-
     тельная ситуация processError может быть воз-
     буждена при исполнении процедуры Switch.)
PROCEDURE Create
          (procBody: Body; extraSpace: CARDINAL;
           procUrg: Urgency; procParams: Parameter;
           VAR procId: ProcessId);
PROCEDURE Start
          (procBody: Body; extraSpace: CARDINAL;
            procUrg: Urgency; procParams: Parameter;
            VAR procId: ProcessId);
    (Вызов каждой из этих процедур создаёт новый
     процесс, идентификация которого присваивается
     переменной procId. С новым процессом связы-
     ваются процедура, являющаяся значением
```

параметра procBody, срочность процесса, являющаяся значением параметра procUrg, минимальный размер рабочей области, который задаётся значением параметра extraSpace, и "параметр" процесса, являющийся значением параметра procParams. Вызов процедуры Create придаёт созданному процессу пассивное состояние, а вызов процедуры Start — состояние готовности.)

(В пояснениях к дальнейшим процедурам будем обозначать исполняемый процесс как **E**, а процесс, идентификация которого есть значение параметра procId, — как **P**.)

PROCEDURE SuspendMe;

(При исполнении этой процедуры процесс **E** приостанавливается и переходит в пассивное состояние. Тем самым приостанавливается — перед самым завершением — и исполнение самой процедуры SuspendMe.)

PROCEDURE Activate (procId: ProcessId);

(Если процесс **P**, находится в пассивном состоянии или состоянии ожидания, то вызов данной процедуры переводит этот процесс в состояние готовности, а в противном случае вызов не производит никакого эффекта.)

PROCEDURE SuspendMeAndActivate
(procId: ProcessId);

(В предположении, что процесс **P** не находится в состоянии готовности, эффект исполнения данной процедуры к моменту завершения этого исполнения состоит в том, что

- если **E** не есть **P**, то процесс **E** перешёл в пассивное состояние;
- процесс **P** перешёл в состояние готовности.)

PROCEDURE Wait;

(Если с процессом **E** связан какой-либо источник событий, то при исполнении данной процедуры этот процесс приостанавливается и переходит в состояние ожидания. Тем самым приостанавливается и исполнение самой процедуры.)

PROCEDURE Switch (procId: ProcessId;
VAR info: Parameter);

(Исполнение данной процедуры определено только в предположении, что срочность, связанная с процессом **E**, не превышает срочности, связанной с процессом **P**, и состоит в следующем:

- процесс **P** переходит в состояние исполнения, и — одновременно — процесс **E** переходит в пассивное состояние;
- когда процесс **E** возобновляется, параметру-переменной info присваивается новое значение; если возобновление произошло в

результате вызова процедуры Switch, то этим значением является значение параметра info этого вызова; если же возобновление произошло в результате вызова процедуры Activate или SuspendMeAndActivate, то этим значением является ***псевдоимя***.

Если в момент вызова данной процедуры процесс **P** не находился ни в пассивном состоянии, ни в состоянии ожидания, то может быть возбуждена исключительная ситуация ProcessError.)

PROCEDURE StopMe;

(Если процесс **E** есть главный процесс, то вызов данной процедуры равносилен последовательно исполняемым вызовам процедур SuspendMe и HALT. В противном случае вызов процедуры завершает исполнение процесса **E**. С процессом **E** не должен быть связан никакой источник событий.)

PROCEDURE Attach (eventSource: Sources);

(Вызов этой процедуры связывает с процессом **E** источник событий, являющийся значением параметра eventSource. Если данный источник событий был связан с некоторым другим процессом, то эта последняя связь устраняется.)

PROCEDURE Detach (eventSource: Sources);

(Если источник событий, являющийся значением параметра eventSource, связан с процессом **E**, то

в результате вызова этой процедуры данная связь устраняется. В противном случае вызов процедуры не производит никакого эффекта.)

PROCEDURE IsAttached (eventSource: Sources): BOOLEAN;

(Значение вызова этой функции есть ***истина*** в том и только том случае, если источник событий, являющийся значением переменной eventSource, связан в данный момент с некоторым процессом.)

PROCEDURE Handler (eventSource: Sources): ProcessId;

(Если источник событий, являющийся значением переменной eventSource, связан с некоторым процессом, то значение вызова этой функции есть идентификация этого процесса.)

PROCEDURE Me(): ProcessId;

(Значением вызова этой функции является идентификация процесса **E**.)

PROCEDURE MyParam(): Parameter;

(Значением вызова этой процедуры является значение "параметра" процесса **E** на момент создания данного процесса.)

PROCEDURE UrgencyOf (procId: ProcessId): Urgency;

(Значением вызова этой процедуры является связанная с процессом 2E срочность.)

```
PROCEDURE ProcessesException ():
                                    ProcessesExceptions;
```

(Если в данный момент обрабатывается некоторая языковая исключительная ситуация, принадлежащая источнику ProcessesExceptions, то значением вызова этой функции является эта исключительная ситуация. В противном случае возбуждается языковая исключительная ситуация exException.)

```
PROCEDURE IsProcessesException (): BOOLEAN;
```

(Эффект вызова этой функции равносилен эффекту вызова функции IsCurrentSource из модуля EXCEPTIONS с фактическим параметром — языковым источником ProcessesExceptions.)

Примеры:

1) (схематический; просьба вначале не обращать внимание на части текста, оформленные как примечания)

```
MODULE prcs;
   IMPORT Processes;
(* IMPORT Semaphors;  *)
   FROM SWholeIO IMPORT WriteCard;
   VAR pr1,pr2,pr3: Processes:ProcessId;
         b1,b2,b3: Processes.Parameter;
(*   VAR s: Semaphores.SEMAPHORE; *)
   PROCEDURE q (i: CARDINAL);
   BEGIN WriteCard(i,0)
```

```
    END q;
    PROCEDURE p1;
    BEGIN  q(4);
           Processes.SuspendMeAndActivate(pr1);
           q(5);
(*           Semaphores.Release(s) *)
    END p1;
    PROCEDURE p2;
    BEGIN  q(2); Processes.Activate(pr1); q(3);
           Processes.SuspendMe
    END p2;
    PROCEDURE p3;
    BEGIN q(1);
          Processes.SuspendMeAndActivate(pr2)
    END p3;
  BEGIN   (*Semaphores.Create(s,0); *)
          Processes.Create(p1,10000,1,b1,pr1);
          Processes.Create(p2,10000,2,b2,pr2);
          Processes.Start(pr3);
(*        Semaphores.Claim(s); *)
          q(6)
  END prcs.
```

Перед началом исполнения программы существует единственный, главный процесс со срочностью 0. В начале исполнения создаются три новые процесса: pr1 со связанной с ним процедурой p1 и срочностью 1, pr2

(соответственно, p2 и 2), pr3 (p3 и 3). Процессы pr1 и pr2 оказываются в пассивном состоянии, а процесс pr3 — в состоянии готовности. Дальше будут происходить следующие действия:

- так как срочность процесса pr3 больше срочности главного процесса, начинает исполняться процесс pr3, а главный процесс переходит в состояние готовности;
- процесс pr3 выдаёт число 1 и исполняет вызов процедуры SuspendMeAndActivate(pr2), который переводит процесс pr3 в пассивное состояние, а процесс pr2 — в состояние готовности; так как срочность процесса pr2 больше срочности главного процесса, начинает исполняться процесс pr2;
- процесс pr2 выдаёт число 2 и исполняет вызов процедуры Activate(pr1), который переводит процесс pr1 в состояние готовности; поскольку срочность процесса pr2 превышает срочность процесса pr1, будет исполняться по-прежнему процесс pr2;
- процесс pr2 выдаёт число 3 и исполняет вызов процедуры SuspendMe, который переводит процесс pr2 в пассивное состояние; после этого в состоянии готовности остаются главный процесс и процесс pr1; поскольку срочность последнего больше срочности главного процесса, он и начинает исполняться;

- процесс pr1 выдаёт число 4 и исполняет вызов процедуры SuspendMeAndActivate(pr1); поскольку параметр этого вызова есть сам процесс pr1, этот процесс переводится в состояние готовности, а затем снова в состояние исполнения; после этого он выдаёт число 5 и ввиду своего завершения перестаёт существовать;
- остались три процесса: pr1 и pr2 в пассивном состоянии и главный процесс — в состоянии готовности; последний возобновляется и выдаёт число 6.

Таким образом, эффект исполнения программы prcs свёлся к последовательной выдаче чисел 1, 2, 3, 4, 5 и 6.

(Если исполнение этого модуля происходит в операционной обстановке с "задержкой диспетчера", то переход процесса pr3 из состояния готовности в состояние исполнения может задержаться и число 6 будет выдано раньше, чем число 1.)

2) Пусть имеются два процесса. Первый из них воспринимает данные (в форме неинтерпретированного содержимого байтов) из неограниченного потока данных. Содержимое каждого очередного байта передаётся второму процессу, который проверяет, удовлетворяет ли

это содержимое некоторому условию, после чего передаёт первому процессу сигнал об исходе этого испытания. В зависимости от результатов этого испытания первый процесс пропускает или не пропускает рассматриваемое содержимое байта на выход. Указанная работа будет осуществляться программным модулем sw. Ввод очередного байта, проверка его и (возможная) выдача осуществляются соответственно процедурами in, test и out, импортируемыми из раздельных модулей Q.

```
MODULE sw;
   FROM Q IMPORT in,out,test;
   FROM Storage IMPORT ALLOCATE;
   FROM Processes IMPORT ProcessId, Parameter,
                         Create, Activate, Switch;
   VAR pr1,pr2: ProcessId; par1,par2,give: Parameter;

   PROCEDURE p1;
   BEGIN LOOP in(par1); give^:=par1^;
                Switch(pr2,par1); out(par1,give)
          END
   END p1;
   PROCEDURE p2;
   BEGIN LOOP Switch(pr1,par2); test(par2)
          END
   END p2;
BEGIN NEW(par1); NEW(par2); NEW(give);
```

```
        Create(p1,10000,1,par1,pr1);
        Create(p2,10000,1,par2,pr2);
        Activate(pr2);
END sw.


DEFINITION MODULE Q;
  FROM SYSTEM IMPORT LOC;
  FROM Processes IMPORT Parameter;
  PROCEDURE in (VAR x: Parameter);
  PROCEDURE test (VAR x: Parameter);
  PROCEDURE out
                (VAR x: Parameter; y: Parameter);
END Q.
```

Проследим за работой программы. После предварительных действий — создания переменных, на которые будут ссылаться значения переменных par1, par2, give, и создания процессов pr1 и pr2 — запускается процесс pr2, который немедленно передаёт управление процессу pr1. Этот "пустой шаг" был необходим, так как первое применение процедуры Switch происходит без передачи данных. Процесс pr1 присваивает с помощью процедуры in очередное вводимое данное (будем называть его **b**) переменной, на которую ссылается par1, а затем копирует его в переменную, на которую ссылается give. Значение

переменной-указателя par1, то есть ссылка на "первый экземпляр **b**", тут же передаётся процедурой Switch процессу pr2, работа которого возобновляется (а "второй экземпляр **b**" сохраняется для дальнейших действий). Возобновление работы процесса pr2 начинается с того, что завершается недоконченное исполнение вызова процедуры Switch, причём параметр-переменная par2 получает переданное, как сказано выше, значение переменной-указателя par1. Затем исполняется процедура test, которая проверяет, удовлетворяет ли **b** заложенным в процедуру условиям, и заменяет значение переменной, на которую ссылается par2, (то есть **b**) на значение **s**, сигнализирующее о выполнении или невыполнении этих условий. На следующем витке цикла управление передаётся процессу pr1, причём значение переменной par2, ссылающееся на **s**, присваивается переменной par1. Следующее за этим исполнение процедуры out в зависимости от значения **s** либо выдаёт "второй экземпляр **b**", либо не совершает никаких действий, после чего выполняется следующий виток цикла.

22.2. Взаимодействие процессов через семафоры. Стандартный модуль Semaphores

Взаимодействие между процессами может осуществляться также с помощью средств, предоставляемых стандартным модулем Semaphores, а именно, *семафоров*.

Семафоры суть объекты, создаваемые и уничтожаемые во время исполнения программы. С семафором в момент его создания связываются его *идентификация* и целочисленная переменная — *счётчик* семафора. С семафором в каждый данный момент связано также некоторое (первоначально пустое) множество процессов программы, *задержанных* данным семафором.

(Процессы, о которых сказано выше, суть процессы, создаваемые модулем Processes. Отсюда ясно, что использование модуля Semaphores требует импорта процедур модуля Processes, создающих и запускающих процессы. Возможность использования наряду с модулем Semaphores других средств из модуля Processes зависит от конкретной реализации.)

В модуле определений Semaphores имеются следующие описания и определения типов и процедур:

```
TYPE SEMAPHORE;
   (Значениями этого типа являются идентификации
    семафоров.)
PROCEDURE Create (VAR s: SEMAPHORE;
                  initialCount: CARDINAL);
```

(Вызов этой процедуры создаёт новый семафор, идентификация которого присваивается переменной s. Начальным значением счётчика семафора становится значение параметра initialCount.)

(В пояснениях к дальнейшим процедурам будем обозначать семафор, идентификация которого при его создании была присвоена переменной s, тоже как s.)

```
PROCEDURE Destroy (VAR s: SEMAPHORE);
```

(Если множество процессов, задержанных семафором s, пусто, вызов этой процедуры уничтожает семафор s. В противном случае исполнение процедуры зависит от реализации.)

```
PROCEDURE Claim (s: SEMAPHORE);
```

(Вызов этой процедуры осуществляет одно из следующих двух действий:

- если текущее значение счётчика семафора s больше нуля, то оно уменьшается на единицу;
- если значение счётчика равно нулю, то исполняемый процесс приостанавливается, переходя в состояние, отличное от состояния готовности; процесс этот включается во множество процессов, задержанных семафо-


```
PROCEDURE Release (s: SEMAPHORE);
```

(Вызов этой процедуры осуществляет одно из следующих двух действий:

- если множество процессов, задержанных семафором s, непусто, то один из процессов этого множества исключается из данного множества и переходит в состояние готовности;
- в противном случае текущее значение счётчика семафора s увеличивается на единицу.)

```
PROCEDURE CondClaim (s: SEMAPHORE):
                                    BOOLEAN;
```

(Если текущее значение счётчика семафора s больше нуля, то при ис полнении данной процедуры-функции значение этого счётчика уменьшается на единицу, а значением вызова функции становится ***истина***. Если же значение счётчика равно нулю, то значением вызова функции становится ***ложь***.)

Примеры:

1) (использован известный алгоритм; см. [4], стр. 200)

Будем моделировать работу двух устройств, одно из которых вводит, а другое выводит некоторые данные. Имеется в виду, что в ходе этой работы элементы данных накапливаются в каком-то "буфере", содержащем не более чем определённое количество N таких элементов. Будем считать, что в некотором раздельном модуле, из которого происходит неквалифицируемый импорт в основную программу, описаны следующие объекты:

- константа N;
- процедура rd с параметром i типа CARDINAL, вводящая очередной элемент данных и помещающая его в буфере на i-ом по счёту месте;
- процедура wr с параметром i типа CARDINAL, выводящая i-ый по счёту элемент буфера.

Будем считать далее, что в программном модуле созданы два семафора: used с начальным значением счётчика 0 и unused с начальным значением счётчика N, а также созданы и запущены два процесса pr1 и pr2, с которыми связаны соответственно процедуры p1 и p2, описания которых приводятся ниже. Используемые в процедурах переменные inp и

outp имеют тип CARDINAL и начальное значение 0.

```
PROCEDURE p1;
BEGIN LOOP Semaphores.Claim(unused);
                inp:=(inp MOD N)+1;
                 rd(inp); Semaphores.Release(used)
          END;
END p1;
PROCEDURE p2;
BEGIN LOOP Semaphores.Claim(used);
               outp:=(outp MOD N)+1;
                wr(outp); Semaphores.Release(unused)
          END;
END p2;
```

Счётчик семафора unused равен в каждый момент числу элементов данных, которые можно ещё поместить в буфер, и процесс pr1 приостанавливается, когда буфер оказывается заполненным. В свою очередь, счётчик семафора used равен в каждый момент числу элементов данных, занесённых в буфер (после последнего его очищения), и процесс pr2 приостанавливается, когда это число становится равным нулю и процесс не может поэтому осуществить очередной выдачи. Обеспечиваемое таким образом взаимодействие процессов позволяет наилучшим образом использовать вводящее и выводящее

устройства, которые, вообще говоря, могут работать с разными скоростями.

2) Вернёмся к примеру 1 из 22.1. При наличии в операционной обстановке "задержки диспетчера" обеспечить желаемый порядок исполнения процессов можно, как было сказано ранее, при помощи механизма семафоров. Убрав в программе из указанного примера скобки примечаний (* и *), мы получим программу, которая отличается от исходной лишь тем, что исполнение оператора, выдающего число 6, будет задержано до окончания исполнения всех созданных в программе процессов.

Заметим, что в примерах, связанных с семафорами, мы применяли простой импорт из модулей Processes и Semaphores, так как применение неквалифицируемого импорта создавало бы в программах коллизию между разными процедурами Create.

22.3. Стандартные модули для работы со строками

Здесь будут рассматриваться модули, дающие средства для работы со строками, абстрагированными из значений строкового типа.

22.3.1. Некоторые общие замечания к процедурам

Попытаемся дать некоторые единообразные пояснения к процедурам, определения которых будут приведены в дальнейшем.

Для некоторых из этих процедур один из параметров может быть объявлен "входным параметром" или же два параметра могут быть объявлены соответственно первым и вторым входными параметрами. Кроме того, для некоторых процедур один из параметров-переменных может быть объявлен "выходной переменной".

Если процедура имеет входной параметр или первый и второй входные параметры, то перед "основным этапом" исполнения этой процедуры из каждого строкового значения, являющегося значением входного параметра или же первого или второго входного параметра, абстрагируется строка, которая будет считаться "входной строкой" (соответственно, "первой, второй входной строкой") процедуры.

О действиях, совершаемых на основном этапе исполнения процедуры, будет идти речь в пояснениях к каждой отдельной процедуре. В частности, в ходе этих действий может быть создана некоторая "выходная строка".

Если процедура имеет выходную переменную, то после основного этапа исполнения процедуры выполняются следующие заключительные действия:

- если длина **L** выходной строки меньше, чем число **M** компонент в значении выходной

переменной, то первым **L** компонентам выходной переменной присваиваются соответствующие компоненты выходной строки, (**L**+1)-ой компоненте присваивается ***признак конца строки***, а значения всех последующих компонент становятся неопределёнными;

- в противном случае первые **M** компонент выходной строки присваиваются соответствующим компонентам выходной переменной.

22.3.2. Стандартный модуль Strings

Стандартный модуль Strings даёт средства высокого уровня, служащие в основном для преобразования одних строк в другие строки.

Дадим для процедур этого модуля некоторые предварительные общие пояснения, добавочные к пояснениям, данным в предыдущем пункте, и относящиеся к основному этапу исполнения процедуры. Будем, если это нужно, считать строки, с которыми работает этот основной этап, состоящими из последовательных сцеплённых "подстрок".

Пусть длина строки **S** есть **L**. Тогда подстрока **B** этой строки с "начальным индексом" **I** и "максимальной длиной" **M** определяется следующим образом:

- если **I** ≥ **L**, то подстрока **B** — пустая;

- иначе, если **I** + **M** > **L**, то подстрока **B** состоит из всех последовательных компонент строки **S**, начиная с компоненты, индекс которой есть **I**;
- иначе, подстрока **B** состоит из **M** последовательных компонент строки **S**, начиная с компоненты, индекс которой есть **I**.

Далее, если некоторая строка **S** состоит из подстрок **A**, **B** и **C**, то под "вычёркиванием" подстроки **B** из строки **S** будет пониматься создание строки, составленной из подстрок **A** и **C**. Если строка **S** состоит из подстроки **A** длиною **Q** и подстроки **C**, то под "вставкой" некоторой строки **B** в строку **S** с "позиции" **Q** будет пониматься создание строки, составленной из подстрок **A**, **B** и **C**.

Заметим ещё, что семи собственно процедурам модуля Strings с некоторыми идентификаторами **P** будут соответствовать семь процедур-функций с идентификаторами Can**P**All. Исполнение каждой такой функции даёт возможность заранее проверить, будет ли исполнение соответствующей процедуры **P** (с некоторыми определёнными параметрами) в каком-то смысле "удачным".

Кончив общие пояснения, переходим к самому модулю. В модуле определений Strings имеются следующие определения и описания типов и процедур.

```
TYPE String1 = ARRAY [0..0] OF CHAR;
   (Этот тип задан для того, чтобы из единичных
    литерных значений можно было строить значе-
   ния однолитерного строкового типа.)
TYPE CompareResults = (less, equal, greater);
PROCEDURE Length (stringVal: ARRAY OF CHAR)
                                    CARDINAL;
   (Значение вызова функции Length равно значению
    вызова функции LENGTH с тем же фактическим
    параметром.)
PROCEDURE Assign (source: ARRAY OF CHAR;
                  VAR destination: ARRAY OF CHAR);
   (Входной параметр есть параметр source. Выход-
   ная переменная есть переменная destination.
    Выходной строкой становится входная строка.)
PROCEDURE CanAssignAll
    (sourceLength: CARDINAL;
     VAR destination: ARRAY OF CHAR): BOO-
LEAN;
   (Значением вызова этой функции становится
    истина, если значение параметра sourceLength
не превышает числа компонент значения
    переменной destination, и ложь в противном слу-
чае.)
PROCEDURE Extract (source: ARRAY OF CHAR;
          startIndex, numberToExtract: CARDINAL;
              VAR destination: ARRAY OF CHAR);
```

(Входной параметр есть параметр source. Выходная переменная есть переменная destination. Выходной строкой становится подстрока входной строки с начальным индексом и максимальной длиной, равными соответственно значениям параметров startIndex и numberToExtract.)

```
PROCEDURE CanExtractAll
    (sourceLength, startIndex,numberToExtract:
                                        CARDINAL;
     VAR destination: ARRAY OF CHAR):
BOOLEAN;
```

(Значением вызова этой функции становится ***истина***, если

sourceLength ≥ startIndex + numberToExtract ,

а значение параметра numberToExtract не превышает числа компонент значения переменной destination, и ***ложь*** в противном случае.)

```
PROCEDURE Delete
            (VAR stringVar: ARRAY OF CHAR;
             startIndex, numberToDelete: CARDINAL);
```

(Входным параметром и одновременно выходной переменной является параметр stringVar. Выходной строкой становится строка, полученная из входной строки вычёркиванием её подстроки с начальным индексом и максимальной длиной, равными соответственно значениям параметров startIndex и numberToDelete.)

```
PROCEDURE CanDeleteAll
            (stringLength, startIndex,
             numberToDelete: CARDINAL): BOOLEAN;
```

(Значением вызова этой функции становится ***истина***, если

stringLength ≥ startIndex + numberToExtract ,

и ***ложь*** в противном случае.)

```
PROCEDURE Insert (source: ARRAY OF CHAR;
                  startIndex: CARDINAL;
                   VAR destination: ARRAY OF CHAR);
```

(Первый и второй входные параметры суть соответственно параметры destination и source. Выходная переменная есть переменная destination.

Выходной строкой становится строка, полученная вставкой второй входной строки в первую входную строку с позиции, равной значению параметра startIndex.)

```
PROCEDURE CanInsertAll
    (sourceLength, startIndex: CARDINAL;
     VAR destination: ARRAY OF CHAR): BOOLEAN;
```

(Значением вызова этой функции становится ***истина***, если

startIndex ≤ LENGTH(destination) ,

а сумма

sourceLength + LENGTH(destination)

не превышает числа компонент значения переменной destination, и ***ложь*** в противном случае.)

```
PROCEDURE Replace (source: ARRAY OF CHAR;
                   startIndex: CARDINAL;
                   VAR destination: ARRAY OF CHAR);
```

(Первый и второй входные параметры суть соответственно параметры destination и source. Выходная переменная есть переменная destination.

Выходной строкой становится строка, полученная последовательно вычёркиванием из первой входной строки её подстроки с начальным индексом и максимальной длиной, равными соответственно значению параметра startIndex и значению выражения LENGTH(source), и вставкой второй входной строки в первую входную строку с позиции, равной значению параметра startIndex.)

```
PROCEDURE CanReplaceAll
    (sourceLength, startIndex: CARDINAL;
     VAR destination: ARRAY OF CHAR): BOO-
LEAN;
```

(Значением вызова этой функции становится ***истина***, если
sourceLength + startIndex ≤ LENGTH(destination) ,
и ***ложь*** в противном случае.)

```
PROCEDURE Append (source: ARRAY OF CHAR;
                  VAR destination: ARRAY OF CHAR);
```

(Первый и второй входные параметры суть соответственно параметры destination и source. Выходная переменная есть переменная destination.
Выходной строкой становится строка, составленная из сцепленных подстрок, совпадающих соответственно с первой и второй входными строками.)

```
PROCEDURE CanAppendAll
    (sourceLength: CARDINAL;
     VAR destination: ARRAY OF CHAR): BOO-
LEAN;
```

(Значением вызова этой функции становится ***истина***, если сумма

sourceLength + LENGTH(destination)

не превышает числа компонент значения переменной destination, и ***ложь*** в противном случае.)

```
PROCEDURE Concat
                (source1, source2: ARRAY OF CHAR;
                 VAR destination: ARRAY OF CHAR);
```

(Первый и второй входные параметры суть соответственно параметры source1 и source2. Выходная переменная есть переменная destination.
Выходной строкой становится строка, составленная из сцепленных подстрок, совпадающих соответственно с первой и второй входными строками.)

PROCEDURE CanConcatAll
(source1Length, source2Length: CARDINAL;
VAR destination: (ARRAY OF CHAR): BOOLEAN;
(Значением вызова этой функции становится ***истина***, если сумма
source1Length + source2Length
не превышает числа компонент значения переменной destination, и ***ложь*** в противном случае.)

PROCEDURE Compare (stringVal1, stringVal2:
ARRAY OF CHAR): CompareResults;
(Первый и второй входные параметры суть соответственно параметры stringVal1 и stringVal2.
Определим для двух строк **A** и **B** отношение *лексикографического предшествования*. Строка **A** предшествует — в смысле этого отношения — строке **B**, если выполняется одно из следующих условий:

- строка **A** — пустая, а строка **B** — непустая;
- обе строки непустые, первая компонента строки **A** есть литерное значение **a**, первая компонента строки **B** есть литерное значение **b**, и для этих двух значений выполняется отношение **a** < **b**;
- строка **A** состоит из последовательных подстрок **A1** и **A2**, строка **B** состоит из последовательных подстрок **B1** и **B2**, строки

A1 и **B1** совпадают, а строка **B1** предшествует строке **B2**.

Легко видеть, что для двух произвольных строк **A** и **B** всегда вы полняется одно из трёх отношений: либо строка **A** предшествует строке **B**, либо строка **B** предшествует строке **A**, либо строки **A** и **B** совпадают.

Значением вызова данной функции становится значение константы less, если первая входная строка предшествует второй входной строке, значение константы greater, если вторая входная строка предшествует первой входной строке, и значение константы equal, если первая и вторая входные строки совпадают.)

```
PROCEDURE Equal (stringVal1, stringVal2:
                 ARRAY OF CHAR): BOOLEAN;
```

(Первый и второй входные параметры суть соответственно параметры stringVal1 и stringVal2.

Значением вызова этой функции становится ***истина***, если первая и вторая входные строки совпадают, и ***ложь*** в противном случае.)

```
PROCEDURE FindNext
          (pattern, stringToSearch: ARRAY OF CHAR;
           startIndex: CARDINAL;
           VAR patternFound: BOOLEAN;
           VAR posOfPattern: CARDINAL);
```

```
PROCEDURE FindPrev
 (pattern, stringToSearch: ARRAY OF CHAR;
  startIndex: CARDINAL;
  VAR patternFound: BOOLEAN;
  VAR posOfPattern: CARDINAL);
```

(Для обеих процедур первый и второй входные параметры суть соответственно параметры stringToSearch и pattern.

Если первая входная строка состоит из последовательных подстрок **A**, **B** и **C**, длина **P** строки **A** есть такое значение — в случае процедуры FindNext минимальное, но не меньшее чем значение параметра startIndex, а в случае процедуры FindPrev максимальное, но не большее чем значение того же параметра — что строка **B** совпадает со второй входной строкой, то переменной patternFound присваивается значение ***истина***, а переменной posOfPattern — значение **P**. В противном случае переменной patternFound присваивается значение ***ложь***.)

```
PROCEDURE FindDiff
            (stringVal1, stringVal2: ARRAY OF CHAR;
             VAR differenceFound: BOOLEAN;
             VAR posOfDifference: CARDINAL);
```

(Первый и второй входные параметры суть соответственно параметры stringVal1 и stringVal2.

Если первая и вторая входные строки совпадают, то переменной differenceFound присваивается

значение ***ложь***. В противном случае первую входную строку можно считать состоящей из таких последовательных подстрок **A1** длины **P** и **A2**, а вторую входную строку — состоящей из таких последовательных подстрок **B1** и **B2**, что строки **A1** и **B1** совпадают и имеет место одно из следующих трёх условий:

- строка **A2** — пустая, а строка **B2** — непустая,
- строка **A2** — непустая, а строка **B2** — пустая,
- первые компоненты строк **A2** и **B2** — различные литерные значения.

При этих условиях переменной differenceFound присваивается значение ***истина***, а переменной posOfPattern — значение **P**.)

```
PROCEDURE Capitalize
                    (VAR stringVar: ARRAY OF CHAR);
```

(Как входным параметром, так и выходной переменной является параметр stringVar.

Выходной строкой становится строка, каждая **i**-я компонента которой есть значение вызова функции CAP(**A**), где **A** есть **i**-я компонента входной строки.)

Примеры:

а) (чисто иллюстративный)

```
MODULE ex1;
   FROM STextIO IMPORT WriteString, WriteLn;
   FROM Strings IMPORT String1, Length,
                                CanAssignAll, Assign,
                                 CanExtractAll, Extract,
                                CanReplaceAll, Replace,
                                CanAppendAll, Append,
                                CanConcatAll, Concat;
  TYPE ciphers =

(zero,one,two,three,four,five,six,seven,eight,nine);
   VAR fitting: BOOLEAN; ch: String1;
         d: ARRAY [1..6] OF CHAR;
         d1,d2: ARRAY [one..six] OF CHAR;
BEGIN fitting:=FALSE; ch:="M";
   fitting:=fitting OR
         CanAssignAll(Length("OPENNESS"),d2);
   Assign("OPENNESS",d2);
   fitting:= fitting OR
         CanExtractAll(Length("SCIENCE"),0,40,d1);
   Extract("SCIENCE",0,40,d1);
   fitting:=fitting OR
         CanReplaceAll(Length(d1),1,d2);
   Replace(d1,1,d2); d2[four]:="";
   fitting:=fitting OR
         CanAppendAll(Length("OWNER"),d2);
   Append("OWNER",d2);
   fitting:=fitting OR
```

```
        CanConcatAll(Length(ch),Length(d2),d);
    Concat(ch,d2,d);
    IF ~fitting THEN WriteString(d) END
END ex1.
```

К концу исполнения данной программы значением переменной fitting будет ***ложь*** (поскольку каждый из вызовов функций вида Can**P**All выработает ***ложь***, предсказав тем самым "неудачность" предстоящего исполнения процедуры **P**), и поэтому условный оператор разрешит напечатать полученное крайне запутанным способом строковое значение переменной d, а именно строку "MOSCOW".

б) Программа может получить через стандартный канал некоторое количество (не более 49) строк, каждая из которых содержит не более 25 литер. Признаком конца ввода должна служить строка, содержащая единственную литеру #. Программа лексикографически упорядочивает эти строки и в полученном порядке выдаёт их.

```
MODULE ex2;
  FROM StextIO
        IMPORT ReadString, SkipLine, WriteString,
                WriteLn;

  FROM Strings IMPORT Equal, Compare,
```

```
                                    CompareResults;
  TYPE Word = ARRAY [1..25] OF CHAR;
  VAR happened: BOOLEAN;
        table: ARRAY [0..50] OF Word;
        i,j: CARDINAL;
  PROCEDURE permut(VAR s1,s2: Word);
    VAR s: Word;
  BEGIN IF Compare(s1,s2) # less
          THEN happened:=TRUE; s:=s1; s1:=s2;
                s2:=s
           END
  END permut;
BEGIN happened:=TRUE; table[0]:=""; j:=0;
        WHILE ~Equal(table[j],"#")
         DO j:=j+1; ReadString(table[j]); SkipLine
         END;
         WHILE happened
          DO happened:=FALSE; i:=1;
               WHILE i # j
                DO permut(table[i-1],table[i]); i:=i+1
                END;
               j:=j-1
          END;
           i:=1;

           WHILE ~Equal(table[i],"#")
           DO WriteString(table[i]); WriteLn; i:=i+1
```

```
END
END ex2.
```

В частности, если ввести данные об одиннадцати послевоенных президентах США в форме:

```
Truman H.S.       1945   —1953
Eisenhower D.D.   1953   —1961
Kennedy J.F.      1961   —1963
Johnson L.B.      1963   —1969
Nixon R.M.        1969   —1974
Ford G.R.         1974   —1977
Carter J.         1977   —1981
Reagan R.W.       1981   —1989
Bush G.H.W.       1989   —1993
Clinton B.        1993   —2001
Bush G.W.         2001   ---
#                                —
```

то выданы будут следующие строчки:

```
Bush G.H.W.       1989—1993
Bush G.W.         2001---
Carter J.         1977—1981
Clinton B.        1993—2001
Eisenhower D.D.   1953—1961
Ford G.R.         1974—1977
Johnson L.B.      1963—1969
Kennedy J.F.      1961—1963
Nixon R.M.        1969—1974
Reagan R.W.       1981—1989
```

Truman H.S. 1945—1953

в) Данная процедура ищет в строке, сопоставленной значению параметра text, все (не пересекающиеся друг с другом) вхождения строки, сопоставленной значению параметра old, и заменяют каждое такое вхождение на строку, сопоставленную значению параметра new. (Оператор pos:=pos+1 включён в процедуру для того, чтобы можно было осуществлять также и тождественную замену.)

```
PROCEDURE FindAndChange
              (VAR text: ARRAY OF CHAR;
               old,new: ARRAY OF CHAR);
VAR pos: CARDINAL; found: BOOLEAN;
BEGIN pos:=0;
        LOOP FindNext(old,text,pos,found,pos);
                IF found
                THEN Delete(text,pos,Length(old));
                        Insert(new,pos,text);
                       pos:=pos+1
                ELSE RETURN
                END
          END
END FindAndChange
```

22.3.3. Стандартные модули для строково-числовых преобразований

Рассматриваемые ниже (см. 22.3.3.2, 22.3.3.3) стандартные модули предоставляют средства высокого уровня для преобразования строк в представляемые этими строками числовые значения и для преобразования числовых значений в соответствующие им строки. В п. 22.3.3.1 мы познакомимся с одним из типов, используемых в этих преобразованиях.

22.3.3.1. Стандартный модуль ConvTypes

В модуле определений ConvTypes имеется, в частности, следующее описание типа.

```
TYPE ConvResults =
        (strAllRight, strOutOfRange, strWrongFormat,
         strEmpty);
```

Для удобства изложения мы будем считать, что данный идентификатор типа подвергается неквалифицируемому импорту во все рассматриваемые ниже стандартные модули строковых преобразований.

22.3.3.2. Стандартный модуль WholeStr

С помощью средств данного модуля осуществляются строково-числовые преобразования для целочисленных значений. При этом внешнее представление соответствующей строки (то есть последовательность литер из изображения строки, имеющего данную строку своим значением) может

иметь либо форму целого без знака, либо форму целого со знаком.

В модуле определений WholeStr имеются следующие определения процедур:

```
PROCEDURE StrToCard
    ( str: ARRAY OF CHAR; VAR card: CARDINAL;
      VAR res: ConvResults);
```

(Входной параметр есть параметр str.

Процедура преобразует входную строку в некоторое целочисленное значение. Процесс просмотра компонент входной строки начинается с того, что из неё удаляются все находящиеся в ней начальные компоненты, соответствующие пробелам. Если после этого строка оказывается пустой, то переменной res присваивается значение константы strEmpty, а значение переменной card становится неопределённым. Иначе, если внешнее представление остатка строки имеет форму целого без знака, а соответствующее ему целочисленное значение находится в пределах диапазона типа CARDINAL, то переменной res присваивается значение константы strAllRight, а переменной card присваивается указанное целочисленное значение. Иначе, если внешнее представление остатка строки имеет форму целого без знака, а соответствующее ему целочисленное значение

находится вне диапазона типа CARDINAL, то переменной res присваивается значение константы strOutOfRange, а переменной card присваивается значение выражения MAX(CARDINAL). В противном случае переменной res присваивается значение константы strWrongFormat, а значение переменной card становится неопределённым.)

```
PROCEDURE StrToInt
        (str: ARRAY OF CHAR; VAR int: INTEGER;
         VAR res: ConvResults);
```

(Входной параметр есть параметр str.
Процедура преобразует входную строку в некоторое целочисленное значение. Процесс просмотра компонент входной строки начинается с того, что из неё удаляются все находящиеся в ней начальные компоненты, соответствующие пробелам. Если после этого строка оказывается пустой, то переменной res присваивается значение константы strEmpty, а значение переменной int становится неопределённым. Иначе, если внешнее представление остатка строки имеет форму целого без знака или форму целого со знаком, а соответствующее ему целочисленное значение находится в пределах диапазона типа INTEGER, то переменной res присваивается значение константы strAllRight, а переменной int присваивается указанное целочисленное значение. Иначе, если внешнее представ-

ление остатка строки имеет форму целого без знака или форму целого со знаком, а соответствующее ему целочисленное значение находится вне диапазона типа INTEGER, то переменной res присваивается значение константы strOutOfRange, а переменной int присваивается значение выражения MAX(INTEGER) или выражения -MAX(INTEGER) в зависимости от того, является ли рассматриваемое целочисленное значение положительным или отрицательным. В противном случае переменной res присваивается значение константы strWrongFormat, а значение переменной int становится неопределённым.)

```
PROCEDURE CardToStr
      (card: CARDINAL;VAR str: ARRAY OF CHAR);
   (Выходная переменная есть переменная str.
    Выходной строкой становится такая строка, представляющая значение параметра card, внешнее представление которой имеет форму целого без знака.)
PROCEDURE IntToStr
         (int: INTEGER; VAR str: ARRAY OF CHAR);
   (Выходная переменная есть переменная str.
    Выходной строкой становится такая строка, представляющая значение параметра int, внешнее
```

представление которой имеет форму целого без знака, если значение параметра int неотрицательно, и форму целого со знаком в противном случае.)

22.3.3.3. Стандартные модули RealStr и LongStr

С помощью этих модулей осуществляются строково-числовые преобразования для вещественных числовых значений. При этом внешнее представление соответствующих строк может иметь либо форму вещественного с фиксированной точкой, либо форму вещественного с плавающей точкой.

В модулях определения RealStr и LongStr имеются следующие определения процедур, где **R** обозначает соответственно REAL или LONGREAL:

```
PROCEDURE StrToReal
                (str: ARRAY OF CHAR; VAR real: R;
                 VAR res: ConvResults);
```

(Входной параметр есть параметр str.

Процедура преобразует входную строку в некоторое вещественное числовое значение. Процесс просмотра компонент входной строки начинается с того, что из неё удаляются все находящиеся в ней начальные компоненты, соответствующие пробелам. Если после этого строка оказывается пустой, то переменной res присваивается значе-

ние константы strEmpty, а значение переменной real становится неопределённым. Иначе, если внешнее представление остатка строки имеет форму вещественного с фиксированной точкой или форму вещественного с плавающей точкой, а соответствующее ему вещественное числовое значение находится в пределах диапазона типа **R**, то переменной res присваивается значение константы strAllRight, а переменной real присваивается указанное вещественное числовое значение. Иначе, если внешнее представление остатка строки имеет форму вещественного с фиксированной точкой или форму вещественного с плавающей точкой, а соответствующее ему вещественное числовое значение находится вне диапазона типа **R**, то переменной res присваивается значение константы strOutOfRange, а переменной real присваивается значение выражения MAX(**R**) или выражения -MAX(**R**) в зависимости от того, является ли это вещественное числовое значение положительным или отрицательным. В противном случае переменной res присваивается значение константы strWrongFormat, а значение переменной real становится неопределённым.)

```
PROCEDURE RealToFixed
                    (real: R; place: INTEGER;
                     VAR str: ARRAY OF CHAR);
```

(Выходная переменная есть переменная str.
Выходной строкой становится такая строка, внешнее представление которой есть исходное строковое представление значения параметра real, настроенное на позицию, являющуюся значением параметра place.)

```
PROCEDURE RealToFloat
                    (real: R; sigFigs: CARDINAL;
                     VAR str: ARRAY OF CHAR);
PROCEDURE RealToEng
                    (real: R; sigFigs: CARDINAL;
                     VAR str: ARRAY OF CHAR);
```

(Выходная переменная в этих процедурах есть переменная str.
Процедура вырабатывает вначале последовательность литер, являющуюся соответственно исходным или техническим строковым представлением значения параметра real, настроенным на число значащих цифр, являющееся значением параметра sigFigs. Если в этой последовательности дробная часть не содержит цифры, отличной от 0, то дробная часть удаляется. Также, если в последовательности показатель представляет число 0, то он удаляется вместе с литерой "E". Выходной строкой становится строка, внешним представлением которой является полученная последовательность.)

```
PROCEDURE RealToStr(real: R;
                    VAR str: ARRAY OF CHAR);
```

(Выходная переменная есть переменная str.
Если при некотором, по возможности наибольшем, целочисленном значении **P** исходное строковое представление значения параметра real, настроенное на позицию **P**, состоит из стольких литер, каково число компонент переменной str, то вызов данной процедуры равносилен вызову процедуры RealToFixed(cid,real,**P**,width). В противном случае вызов данной процедуры равносилен вызову процедуры RealToFloat(cid,real,sigFigs,width) с таким ненулевым значением параметра sigFigs, чтобы количество литер в вырабатываемой этой процедурой последовательности по возможности не превосходило числа компонент переменной str. Особым случаем является вызов процедуры RealToString(cid,real,width), когда значение параметра width равно 0. Такой вызов равносилен вызову RealToFloat(cid,real,0,0).)

22.4. Стандартный модуль SysClock

Стандартный модуль SysClock предоставляет средства для контроля за протеканием системного времени в вычислительной системе. Системное время характеризуется годом, месяцем, сутками

(определяемыми по правилам григорианского календаря), часом, минутой, секундой и некоторой дробной частью секунды.

В модуле определений SysClock имеются следующие определения и описания констант, типов и процедур.

```
CONST maxSecondParts = M;
```

(**M** имеет тип CARDINAL. Значение его определяется реализацией.)

```
TYPE Month = [1 .. 12]; Day = [1 .. 31];
     Hour = [0 .. 23]; Min = [0 .. 59]; Sec = [0 .. 59];
     Fraction = [0 .. maxSecondParts];
     UTCDiff = [-780 .. 720];
     DateTime =
        RECORD year: CARDINAL; month: Month;
               day: Day; hour: Hour; minute: Min;
               second: Sec; fractions: Fraction;
               zone: UTCDiff;
                 summerTimeFlag: BOOLEAN;
        END;
```

(Если диапазон значений типа CARDINAL в данной конкретной реализации достаточно велик, то значениями поля year могут служить фактические календарные года. В противном случае соответствие между календарными годами и значениями этого поля определяется реализацией.

Поле zone содержит поправку часового пояса — разность в минутах между "международным согласованным", то есть гринвичским, временем и местным временем.

Поле summerTimeFlag может служить для информации о том, действует ли "летнее время".)

PROCEDURE CanGetClock (): BOOLEAN;

(Значение вызова этой функции есть ***истина*** в том и только том случае, если система позволяет узнавать текущее системное время с помощью процедуры GetClock.)

PROCEDURE CanSetClock (): BOOLEAN;

(Значение вызова этой функции есть ***истина*** в том и только том случае, если система позволяет "переводить системные часы", то есть совмещать с помощью процедуры SetClock данный физический момент с определённым моментом системного времени.)

PROCEDURE IsValidDateTime (userData: DateTime): BOOLEAN;

(Значение вызова этой функции есть ***истина*** в том и только том случае, если значения полей параметра userData задают некоторую календарно допустимую дату.)

PROCEDURE GetClock (VAR userData: DateTime);

(Процедура присваивает первым семи полям переменной userData соответствующие значения, определяемые текущим системным временем. Полям zone и summerTimeFlag присваиваются соответствующие значения, определяемые текущим состоянием вычислительной системы. Если какое-либо из требуемых числовыхзначений по какой-либо причине недоступно, то полю month или day присваивается число 1, а любому иному полю — число 0.)

PROCEDURE SetClock (userData: DateTime);

(Если значение вызова функции CanSetClock есть ***ложь***, то эффект исполнения данной процедуры равносилен исполнению пустого оператора. Иначе, если значения полей параметра userData задают календарно недопустимую дату, то эффект исполнения процедуры неопределён. Иначе, при исполнении процедуры системное время вычислительной системы устанавливается в соответствии со значениями соответствующих полей параметра userData. Равным образом, текущее состояние вычислительной системы переопределяется в соответствии со значениями полей zone и summerTimeFlag параметра userData.)

ПРИЛОЖЕНИЯ

А. ЧИСЛЕННЫЕ ОГРАНИЧЕНИЯ В КОНКРЕТНЫХ РЕАЛИЗАЦИЯХ

Конкретные реализации могут накладывать те или иные численные ограничения на воспринимаемые этими реализациями программы. Стандарт устанавливает, однако, что эти ограничения не должны быть более жёсткими, чем те, которые определяются приводимой ниже таблицей:

длина простого идентификатора	32
размерность многомерного массива	16
глубина вложенности явных представлений типов-записей	16
число описаний в последовательности описаний локальных модулей	16
глубина вложенности описаний локальных модулей	16
число описаний в последовательности описаний процедур	16
глубина вложенности описаний процедур	16
глубина вложенности THEN в условном операторе	16
глубина вложенности THEN и ELSE в условном операторе	4
глубина вложенности безусловных циклов	16
глубина вложенности циклов пока	16
глубина вложенности циклов до	16

глубина вложенности операторов присоединения	16
глубина вложенности круглых скобок в выражениях	16
число импортируемых модулей	16

Дальнейшая часть таблицы действительна только для тех реализаций, для которых значение SYSTEM.BITSPERLOC × SYSTEM.LOCSPERWORD (число битов в слове) больше или равно 16.

число литер в изображении строки	32
число компонент одномерного массива	4096
число идентификаторов в типе-перечислении	256
число сопрограмм	16
число параметров процедуры	16
глубина вложенности циклов с шагом	16
число альтернатив в операторах выбора	255
глубина рекурсии при обращении к процедуре	256

Б. ОСОБЕННОСТИ РЕАЛИЗАЦИИ XDS

Входной язык реализации XDS отличается от Стандарта Модулы-2 за счёт ряда внесённых в него расширений и изменений. Часть этих расширений и изменений (см. Б.1, Б.2) имеют силу во время трансляции и исполнения конкретной программы лишь при задействовании (включении) специальных

опций языка, а часть (см. Б.3, Б.4) — безотносительно к опциям.

В Б.5 будут рассмотрены некоторые детали языка, которые Стандарт оставляет на усмотрение конкретных реализаций и которые доопределены реализацией XDS. Численные ограничения, устанавливаемые данной реализацией, будут рассмотрены в Б.6. Те из этих деталей и этих ограничений, которые определяются не только собственно реализацией, но также конкретной операционной системой и конкретной машинной конфигурацией, в этих пунктах, как правило, не приводятся — с ними следует знакомиться при изучении соответствующей машинной обстановки. Наконец, в Б.7 будут приведены определённые реализацией XDS значения констант модулей LowReal и LowLong.

Б.1. Расширения и изменения языка при задействованной опции M2EXTENSIONS

Б.1.1. Лексические расширения и изменения

Примечания (см. 3)

Кроме примечаний, заключенных в скобки (* и *), допускаются примечания, ограниченные слева двумя черточками "--", а справа — концом строчки.

Числовые изображения литер (см. 3.3.2)

Числовым изображением литеры может быть не только восьмеричное число, сопровождаемое буквой "C", но и шестнадцатиричное число, сопровождаемое буквой "X".

Б.1.2. Динамические массивы (см. 5.6, 5.5, 9.1)

Введём понятие типа динамического массива, который характеризуется некоторой размерностью и некоторым типом компоненты. Явное представление типа динамического массива синтаксически совпадает с некоторым ("соответствующим ему") представлением открытого массива из спецификации типа формального параметра. Оно содержит столько вхождений пары ключевых слов "ARRAY" и "OF", какова данная размерность, и представление типа компоненты открытого массива, служащее представлением данного типа компоненты.

Применение типов динамических массивов допустимо лишь в следующих случаях:

1) Идентификатор типа динамического массива может находиться в позиции спецификации типа формального параметра, и тогда он просто замещает соответствующее ему представление открытого массива.

2) Тип динамического массива может быть использован как связанный тип типа-указателя (в частности адресного типа). Отведение памяти для значения этого типа (динамического массива), на

который будет указывать значение типа-указателя, и освобождение этой памяти осуществляются с помощью расширенных версий соответственно процедуры NEW и процедуры DISPOSE (см. Б.1.8).

3) Если для некоторого типа динамического массива размерность равна 1, а тип компоненты есть либо порядковый тип, либо вещественный или комплексный числовой тип, то представление такого типа динамического массива может быть помещено в конструкторе массива вместо идентификатора типа. Компонентами структуры в таком "конструкторе константного массива" могут быть лишь константные выражения типа, идентичного типу компоненты, так что значение самого конструктора константного массива есть константа. Этот конструктор (или содержащая его основа) может в дальнейшем использоваться лишь в качестве фактического параметра оператора процедуры или вызова функции.

Б.1.3. Защищённые параметры (см. 10)

В спецификации формальных параметров описания процедуры после формального параметра-значения может быть поставлен знак "-". Такой параметр называется *защищённым параметром*; блок данной процедуры не должен допускать изменения значения данного параметра. Вместе с защищённым параметром типа-массива или типа-записи защищены также все их компоненты и поля. Защищённые

параметры не используются в модулях определений.

Б.1.4. Процедура с переменным числом параметров (см. 10)

Последним по счёту формальным параметром процедуры может быть *формальный параметр – последовательность*. Этот параметр помечается в заголовке процедуры — а если необходимо, то и в явном представлении типа-процедуры — синтаксическим символом "SEQ" (в той же позиции, в которой параметр-переменная помечается символом "VAR"). Тип соответствующего формального параметра должен быть типом SYSTEM.LOC (или идентичным ему типом, например, типом SYSTEM.BYTE — см. Б.4). В вызове процедуры (или вызове функции) формальному параметру – последовательности может соответствовать некоторая (в том числе и пустая) последовательность фактических параметров разных типов. При этом значение фактического параметра комплексного числового типа рассматривается как запись с двумя полями соответствующего вещественного числового типа. При исполнении вызова процедуры каждое значение фактического параметра подвергается преобразованию в соответствии со следующими правилами:

- значения типов
  SHORTINT (см. Б.2),
  SYSTEM.INT8, SYSTEM.INT16,

SYSTEM.INT32 (см. Б.3)

и целочисленные отрицательные константы преобразуются в значения типа INTEGER;

- значения типов

  BOOLEAN, CHAR, SHORTCARD (см. Б.2), SYSTEM.CARD8, SYSTEM.CARD16, SYSTEM.CARD32 (см. Б.3),

  типов-перечислений и целочисленные неотрицательные константы преобразуются в значения типа CARDINAL;
- значение типа-отрезка преобразуется так же, как значение объемлющего типа этого типа-отрезка;
- значение любого вещественного числового типа преобразуется в значение типа LONGREAL;
- значение типа-указателя или адресного типа (а значит, и значение скрытого типа), также как и значение типа-процедуры, преобразуется в значение адресного типа;
- значение структурного типа (типа-массива или типа-записи) размещается в 12 ячейках, содержащих в частности логический адрес и уменьшенное на единицу число ячеек, отводимых для структуры.

Значения типа защиты не могут относиться к рассматриваемым фактическим параметрам.

Полученные в результате всех этих действий битовые представления располагаются в последовательных ячейках памяти в порядке

встречаемости соответствующих фактических параметров, и совокупность их рассматривается как массив значений типа SYSTEM.LOC, который и передается формальному параметру – последовательности как единственный соответствующий ему фактический параметр, имеющий тип

ARRAY [0..**n**] OF SYSTEM.BYTE ,

где **n** есть уменьшенное на единицу число ячеек в этой совокупности.

Б.1.5. Унарное вычитание для множеств (см. 12.5)

Префиксная операция "-", применённая к множеству **A** типа SET OF **b**, даёт множество всех элементов типа **b**, не принадлежащих к множеству **A**.

Пример (см. пример из 5.1.3):

При наличии описания

TYPE days = SET OF week;

результат операции

- days{Monday .. Friday}

есть множество, совпадающее со значением конструктора множества

days{Sunday,Saturday} .

Б.1.6. Защищённый экспорт (см. 14.4)

В модуле определений после идентификатора переменной, содержащегося в описании переменных, и после идентификатора поля, содержащегося в описании типа-записи, может быть помещен символ "-". В первом случае соответствующая переменная, будучи экспортирована из модуля определений, не должна изменять своего значения. Во втором случае не должно изменять своего значения соответствующее поле экспортированных переменных данного типа. Запрет на изменение значения распространяется как на явное присваивание, так и на возможность передачи данного объекта через параметр-переменную. Если защищённые таким образом объекты имеют тип-массив или тип-запись, то вместе с ними защищены их компоненты и поля.

Б.1.7. Переименование импортируемых модулей (см. 14.5)

Модуль, импортированный в некоторый другой модуль, может в этом последнем быть переименован. Собственное его имя в этом модуле становится недоступным. Чтобы осуществить переименование, надо в списке импорта перед идентификатором импортируемого модуля поместить новое имя модуля, сопровождаемое символом :=.

Пример:

IMPORT Str := Strings

Б.1.8. Новые и изменённые стандартные процедуры (см. 15.1)

Процедура ASSERT

Вызов процедуры ASSERT имеет либо один фактический параметр, либо два фактических параметра. Первый параметр есть выражение типа BOOLEAN, а возможный второй параметр — выражение типа CARDINAL.

Если при вызове процедуры значение первого параметра есть ***ложь***, то возбуждается исключительная ситуация. Если эта исклю чительная ситуация оказывается необработанной, то выдаваемая системой информация об этом событии будет содержать значение второго параметра, а если таковой отсутствует — некоторое другое значение.

Процедура COPY

Вызов процедуры COPY должен иметь два параметра: выражение строкового типа и переменную строкового типа.

Вызов процедуры COPY(**x**,**v**) осуществляет последовательное присваивание компонент выражения **x** компонентам переменной **v** до исчерпания одной из этих компонент. Если в результате этого присваивания значением одной из компонент переменной **v** окажется ***признак конца строки***, то следующие её компоненты могут стать неопределёнными.

Процедуры NEW и DISPOSE

Имеются варианты процедур NEW и DISPOSE с дополнительными фактическими параметрами, являющимися переменными типа CARDINAL. Эти варианты применяются в том случае, если включена дополнительная опция STORAGE.

Вызов NEW(**P**,$\mathbf{v}_1$, ... , $\mathbf{v}_\mathbf{n}$), где тип переменной **P** есть тип-указатель, имеющий своим связанным типом некоторый тип динамического массива **T** с размерностью **n**, создает **n**-мерный массив, у которого тип компоненты есть **T**, а тип каждого **k**-го индекса есть [0..$\mathbf{v}_\mathbf{k}$-1], и присваивает переменной **P** в качестве её значения указатель на этот массив.

Вызов DISPOSE(**P**,$\mathbf{v}_1$,... , $\mathbf{v}_\mathbf{n}$) уничтожает массив, на который указывает значение параметра **P** и размеры которого определяются размерами дополнительных параметров.

Б.1.9. Новые стандартные функции (см. 15.2)

Функция ENTIER

Вызов функции ENTIER имеет один фактический параметр, являющийся выражением некоторого вещественного числового типа. Тип вызова функции — INTEGER.

Значением вызова ENTIER(**x**) является наибольшее целое значение, не превышающее значения параметра **x**.

Функция ASH

Вызов функции ASH имеет два фактических параметра, являющихся выражениями целочисленных типов. Тип вызова функции совпадает с типом первого параметра.

Значение вызова ASH(**x**,**n**) при условии, что значение **N** выражения **n** неотрицательно, есть результат применения операции * к значению выражения **x** и **N**-й степени числа 2, а в противном случае — результат применения операции DIV к значению выражения **x** и (–**N**)-й степени числа 2.

Функция LEN

Вызов функции LEN может иметь один или два фактических параметра.

Первый параметр есть выражение типа

ARRAY $\mathbf{I}_0$, ..., $\mathbf{I}_\mathbf{n}$ OF **C** ($\mathbf{n} \geq 0$),

а второй параметр — выражение некоторого целочисленного типа. Значение второго параметра должно быть неотрицательным и не превышать **n**. Тип вызова функции — CARDINAL.

Значением вызова LEN(**v**,**k**) является число элементов в типе индекса $\mathbf{I}_\mathbf{k}$. Значение вызова LEN(**v**) совпадает со значением вызова LEN(**v**,0).

Б.1.10. Совместимость по присваиванию с типами SYSTEM.LOC и SYSTEM.BYTE (см. 5.10.2 и 16.1)

Тип LOC (а с учётом Б.4 и тип BYTE) из модуля

SYSTEM совместим по присваиванию со стандартными типами CHAR, BOOLEAN, а также типами INT8 и CARD8 из модуля SYSTEM (см. Б.3).

Эти типы также совместимы по присваиванию с типами SHORTINT и SHORTCARD (см. Б.2).

Б.2. Расширения и изменения языка при задействованной опции M2ADDTYPES

Дополнительные целочисленные типы (см. 5.1.1)

Вводятся дополнительные стандартные типы SHORTINT и SHORTCARD, рассматриваемые как типы-отрезки, объемлющие типы которых суть соответственно типы INTEGER и CARDINAL.

Вводятся также дополнительные стандартные типы LONGINT и LONGCARD, идентичные соответственно типам INTEGER и CARDINAL.

Дополнительные правила для совместимости по выражениям и совместимости по присваиванию (см. 5.10.1, 5.10.2 и 7.2)

При выполнении инфиксных арифметических операций или операций отношения типы операндов считаются совместимыми по выражениям также и в том случае, если выполняется одно из следующих условий:

- тип одного из этих операндов есть тип LONGREAL, а тип другого операнда (скажем, **E**)

есть либо тип REAL, либо некоторый целочисленный тип;

- тип одного из операндов есть тип REAL, а тип другого операнда (скажем, **E**) есть целочисленный тип.

Выполнение операции при указанных выше условиях сопровождается заменой значения выражения **E** на значение выражения VAL(LONGREAL,**E**) или, соответственно, выражения VAL(REAL,**E**).

Тип **T0** считается совместимым по присваиванию с типом **T1** выражения **P** также и в том случае, если выполняется одно из следующих условий:

- тип **T0** есть тип LONGREAL, а тип **T1** есть либо тип REAL, либо некоторый целочисленный тип;
- тип **T0** есть тип REAL, а тип **T1** есть некоторый целочисленный тип.

Присваивание при указанных выше условиях сопровождается заменой значения выражения **P** на значение выражения VAL(LONGREAL,**P**) или, соответственно, выражения VAL(REAL,**P**).

Б.3. Дополнительные типы системного модуля SYSTEM (см. 16)

Реализация XDS подразумевает, что в модуле SYSTEM заданы некоторые дополнительные типы.

Дополнительные целочисленные типы

Каждый из целочисленных типов CARD8, CARD16

и CARD32 определяет множество идущих подряд целых неотрицательных чисел, находящихся в диапазоне от 0 до соответственно MAX(CARD8), MAX(CARD16) или MAX(CARD32). Каждый из целочисленных типов INT8, INT16 и INT32 определяет множество идущих подряд целых чисел, находящихся соответственно в диапазоне от MIN(INT8) до MAX(INT8), от MIN(INT16) до MAX(INT16) или от MIN(INT32) до MAX(INT32). При этом должны иметь место неравенства:

MAX(CARD8)<MAX(CARD16)<MAX(CARD32)

и

MIN(INT32)<MIN(INT16)<MIN(INT8)<MAX(INT8)<
MAX(INT16)<MAX(INT32)

В совокупности типов CARDINAL, CARD8, CARD16, CARD32 (а также и SHORTCARD) и в совокупности типов INTEGER, INT8, INT16, INT32 (а также и SHORTINT) каждый тип совместим по выражениям и по присваиванию с каждым другим типом той же совокупности. При этом не исключено, что некоторые пары типов (например, CARDINAL и CARD32, INTEGER и INT32) идентичны.

Дополнительные типы, аналогичные типу BITSET

Каждый из типов SET8, SET16 и SET32 либо идентичен типу BITSET, либо отличается от него лишь верхней границей интервала, соответствующего

базовому типу. При этом значение указанной верхней границы должно расти при переходе от SET8 к SET16 и далее к SET32.

Дополнительный логический тип

Логический тип BOOL32 аналогичен по своему определению и по своему использованию стандартному типу BOOLEAN, но не идентичен ему. (Лексически совпадающие идентификаторы FALSE и TRUE из обоих типов считаются одинаковыми.)

Б.4. Одно отклонение от Стандарта (см. 16.1)

Тип BYTE идентичен типу LOC, и значение константы LOCSPERBYTE равно 1.

Б.5. Доопределение языка реализацией XDS

а) (См. 1.) Принятой моделью исполнения программы является модель со строгим предшествованием.

б) (См. 3.) Реализация приняла в качестве полного набора литер "расширенный" набор ASCII. В частности, дополнительные литеры включают все заглавные и строчные русские буквы, а также знаки $ % ?.

в) (См. 3.3.2.) В качестве ***признака конца строки*** выступает нулевое значение (значение числового изображения литеры 0C).

г) (См. 5.4.1.) Представления значений типов-множеств упакованы.

д) (См. 5.9.) Реализация XDS применяет тип защиты только тогда, когда она работает под операционной системой DOS. При этом тип защиты определён как тип-множество с базовым типом,
определяющим одноэлементное множество, так что идентификатор INTERRUPTIBLE обозначает пустое множество, а идентификатор UNINTERRUPTIBLE всё это одноэлементное множество.

е) (См. 17.4.) Как стандартным каналом ввода, так и стандартным каналом вывода является терминальное устройство компьютера. (Это не исключает того, что конкретная система может предоставить пользователю возможность выбора иных стандартных каналов.)

ж) (См. 17.4.4.3.) Если значение "позиции" ограничено числом **P0**, то при **P** > **P0** исходное строковое представление, настроенное на позицию **P**, имеет **P** десятичных знаков в своей дробной части, но представляет вещественное число с округлением до десяти в степени –**P0**.
Если значение "числа значащих цифр" ограничено числом **F0**, то при **F** > **F0** исходное строковое представление, настроенное на число значащих цифр **F**, совпадает с исходным строковым представлением, настроенным на число значащих цифр **F0**.

Б.6. Некоторые численные ограничения и численные параметры реализации XDS

Значения функций MIN, MAX и SIZE для параметров — стандартных типов и типов из модуля SYSTEM

MIN (CARDINAL) = 0
MAX(CARDINAL) = 4294967295
SIZE (CARDINAL) = 4
(То же для типов LONGCARD и SYSTEM.CARD.32)

MIN (SYSTEM.CARD16) = 0
MAX(SYSTEM.CARD16) = 65535

```
SIZE (SYSTEM.CARD16) =        2
(То же для типа SHORTCARD)

MIN (INTEGER) = -2147483648
MAX(INTEGER) =  2147483647
SIZE (INTEGER) =              4
(То же для типов LONGINT и SYSTEM.INT32)

MIN (SYSTEM.INT16) = -32768
MAX(SYSTEM.INT16) =  32767
SIZE (SYSTEM.INT16) =         2

MIN (SYSTEM.INT8)  = -128
MAX(SYSTEM.INT8) =   127
SIZE (SYSTEM.INT8) =       1
(То же для типа SHORTINT)

MIN (REAL) = -3.402823E+38
MAX(REAL) =  3.402823E+38
SIZE (REAL) =                 4

MIN (LONGREAL) = -1.7976931348623E+308
MAX(LONGREAL) =  1.7976931348623E+308
SIZE (LONGREAL) =                        8

SIZE (COMPLEX) = 8

SIZE (LONGCOMPLEX) = 16
```

MIN (CHAR) = 0
MAX(CHAR) = 255
SIZE (CHAR) = 1

MIN (BOOLEAN) = FALSE
MAX(BOOLEAN) = TRUE
SIZE (BOOLEAN) = 1

MIN (SYSTEM.BOOL32) = FALSE
MAX(SYSTEM.BOOL32) = TRUE
SIZE (SYSTEM.BOOL32) = 4

Если **T** есть тип защиты, заданный как тип-перечисление, то

MIN (**T**) = INTERRUPTIBLE
MAX(**T**) = UNINTERRUPTIBLE
SIZE (**T**) = 2

(Для остальных типов-перечислений значение функции SIZE равно 1, а значение функций MIN и MAX зависят от типа.)

MIN (BITSET) = 0
MAX(BITSET) = 31
SIZE (BITSET) = 4
(То же для типа SYSTEM.SET32)

MIN (SYSTEM.SET16) = 0

```
MAX(SYSTEM.SET16) =  15
SIZE (SYSTEM.SET16) =    2

MIN  (SYSTEM.SET8) =  0
MAX(SYSTEM.SET8)  = 7
SIZE (SYSTEM.SET8)  = 1
```

Значение функции SIZE для типов-указателей и типов-процедур равно 4, а для типов-массивов и типов-записей зависит от типа.

Верхняя граница числа элементов базового типа типа-множества — 32

Значения констант модуля SYSTEM

BITSPERLOC = 8
LOCSPERWORD = 4

Число элементов базового типа

типа BITSET — 32
типа SET32 — 32
типа SET16 — 16
типа SET8 — 8

Некоторые ограничения на программные средства

максимальная длина простого идентификатора — 127 литер
максимальная длина изображения строки — 256 литер

Б.7. Значения констант модулей LowReal и LowLong (см. 17.1.2)

	LowReal	LowLong
radix	2	2
places	32	64
expoMin	–126	–1022
expoMax	127	1023
nModes	6	6
large	3.402823E+38	

1.7976931348623E+308
small 1.175494E-38 2.2250738585072E-308

IEEE	***истина***	***истина***
ISO	***ложь***	***ложь***
rounds	***истина***	***истина***
gUnderflow	***ложь***	***ложь***
exception	***истина***	***истина***
extend	***истина***	***истина***

В. СЛОВАРИ

В.1. Термины языка

В.2. Синтаксические понятия

В.3. Стандартные идентификаторы

В.4. Внешние представления некоторых значений

псевдоимя 5.5

СПИСОК ЛИТЕРАТУРЫ

1. DRAFT INTERNATIONAL STANDARD. ISO/IEC DIS 10514. 1994.Information technology — Modula-2.
2. Information technology — Programming languages — Modula-2 2nd Committee Draft Standard: CD 10514. December 1992.Document ISI/IEC JTC1/SC22/WG13 D18
3. Вирт Н. Программирование на языке Модула-2: Пер. с англ. /Под ред. В.М.Курочкина. — М.: Мир, 1987. — 222 с.
4. Пейган Ф. Практическое руководство по Алголу 68: Пер. с англ. — М.: Мир,1979. — 240 с.