Modula-2. Краткий курс

В сжатой форме излагаются все основные особенности языка Modula-2. Курс ориентирован на тех, кто уже имеет опыт работы с одним из языков программирования (C, C++, Pascal). Наиболее сложные аспекты языка иллюстрируются на простейших примерах. Автор проводит четкую грань между каноническим изложением языка и его практической реализацией (TopSpeed Modula-2). Данный курс может послужить хорошим справочным пособием и для тех, кто уже знаком с языком Modula-2.

0. Предисловие

Одна из серьезных причин слабой известности Modula-2 у нас в стране связана прежде всего с отсутствием хороших практических руководств по этому языку. Мое первое знакомство с языком началось с книги "Programming in Modula-2", которая была написана основоположником языка - швейцарским ученым Никлаусом Виртом (Niklaus Wirth) — одним из самых авторитетных в мире специалистов в области программирования. Спустя несколько лет эта книга была издана и у нас в стране. Появлялись за рубежом также и книги других авторов, таких как Richard Gleaves, Ed Knepley, Robert Platt, K.Christian, Richard Wiener, G.Ford, Richard Sincovec, J.Ogilvie, Pad Terry, Gustav Pomberger и многих, многих других. К сожалению, почти все они мало известны нашим программистам, хотя книги Нэпли и Кристиана были также переведены на русский язык. По простоте и доступности изложения в этом ряду особо выделяется книга Richard Gleaves

"Modula-2 for Pascal Programmers". Что же касается других книг, то их вряд ли можно порекомендовать для начального экспресс-знакомства с возможностями языка. Либо это книги для начинающих, где разжевываются самые элементарные вопросы, а на нюансы у авторов просто не остается времени, либо это крупные работы по технологическим приемам программирования с ориентацией на язык Modula-2. Вот почему многие из тех, кто хотел бы потратить минимум времени на знакомство с языком и начать сразу же практически использовать его в своей работе, оказываются на голодном пайке.

Другая не менее важная причина "экзотичности" языка кроется в отсутствии общепринятых библиотек. А ведь здесь, как, пожалуй, ни в одном другом языке, многие операции вынесены за пределы языка. Это касается ввода-вывода, файловой системы, работы со строками, математических функций. Хорошо известно по этому поводу мнение самого Вирта, который категорически возражал против введения какого бы то ни было стандарта на основные библиотеки. Тем не менее, в настоящее время уже находятся в стадии завершения работы специальной группы SC22/WG13 Британского Института Стандартов, которая в течение нескольких лет вела кропотливую работу по выработке стандарта языка Modula-2.

Данный курс предназначен не для новичков, а для тех, кто уже имеет опыт работы с другими языками и хотел бы быстро познакомиться с Modula-2, чтобы решить, стоит ли уделять этому языку более серьезное внимание. Материал излагается в конспективной форме. Подробнее раскрываются лишь те моменты, которые либо необычны, либо вообще не имеют аналогов в других языках. В связи с этим, дабы не загромождать материал, мы вообще не будем касаться околоязыковых вопросов, таких как его история, перспективы развития целого направления языков Modula2-Oberon-Oberon2, приемы и методы программирования, технология создания сложных программных комплексов. Все это — тема других книг и наших будущих работ.

Следует сразу оговориться, что автор будет максимально следовать канонам языка, но для придания практического характера курса, ориентироваться на TopSpeed Modula-2, одну из самых известных его реализаций для компьютеров типа PC и операционной системы MS-DOS. Другая, менее известная система программирования, Logitech Modula-2 фирмы MultiScope Inc., славящаяся прежде всего своим строгим и надежным компилятором, будет упоминаться лишь изредка. Мы не будем также рассматривать объектно-ориентированные средства языка, поскольку они в каноническом изложении Вирта отсутствовали, и, кроме того, потому что имеются другие языки этого семейства (Oberon, Oberon-2), где этот вопрос решен гораздо элегантнее. Еще один важный аспект — излагается не весь язык, а лишь его подмножество, который автор выработал для себя в ходе десяти лет его практического использования. В частности, слабо освещаются, либо не затрагиваются вообще такие средства, как вариантные записи, локальные модули и механизм приоритетов модулей.

Автор надеется, что данный курс поможет читателю найти ориентиры в бушующем море языков программирования и, быть может, покажет новую, неизвестную грань уже знакомого Вам языка.

1. Введение в язык программирования Modula-2

Если говорить о месте Modula-2 среди других языков, то это универсальный (general purpose) язык программирования высокого уровня, стоящий в одном ряду с такими языками, как Pascal и C. Обладая сопоставимыми с языком Ada выразительными возможностями, он в то же время неизмеримо компактнее и проще. Спроектирован он был на основе опыта работы с языком Pascal прежде всего для создания больших систем и для организации совместной работы больших коллективов программистов, принимающих участие в общем проекте. Этот язык, как и любой другой, накладывает определенный отпечаток на стиль мышления программиста. В отличие от большинства других языков он стимулирует программирование с жесткой дисциплиной и позволяет очень четко и точно выражать свои мысли. Он хорош не только для создания больших и сложных программных

комплексов, но и для крошечных программ. В то же время по эффективности кодогенерации в большинстве промышленных компиляторов он ни в чем не уступает языку С. Язык обладает всем необходимым и для низкоуровневого программирования. Достаточно сказать, что на языке Modula-2 была написана не одна операционная система. Это, например, Medos для компьютера Lilith и для транспьютеров разных моделей, Excelsior — для отечественного 32-разрядного Kronos.

Характерные черты языка Modula-2:

- строгая типизация;
- процедурные переменные;
- концепция модуля и связанное с ним понятие раздельной компиляции;
- технология абстрактных типов данных;
- переменные с абсолютными машинными адресами;
- квазипараллельное программирование.

2. Идентификаторы

Любой объект в любом языке должен быть как-то поименован. Для этого и используются идентификаторы. Идентификаторы в языке Modula-2 состоят только из букв (латинских) и цифр, причем на первом месте обязательно должна быть буква, В отличие от некоторых других языков в идентификаторах делается различие между большими и маленькими буквами.

Есть одна довольно удобная стратегия формирования имен. Всем переменным давайте имена, начинающиеся с маленькой буквы. Процедурам и типам — имена, начинающиеся с заглавной буквы, причем каждое последующее слово в составных именах должно также начинаться с заглавной буквы. Константам, за исключением, быть может, используемых в перечислении, давайте имена, полностью состоящие из больших букв. Если Вы будете следовать этому правилу, то скоро убедитесь, как легко Вам станет ориентироваться в собственной программе.

```
CONST
  N = 24;
TYPE
  WorkArray = ARRAY [1..N] OF CARDINAL;
VAR
  a      : WorkArray;
  isChar : BOOLEAN;
```

При формировании идентификаторов важно знать, какие идентификаторы являются уже зарезервированными. Вот их список.

DEFINITION	интерфейсный модуль
IMPLEMENTATION	исполнительный модуль
MODULE	модуль
FROM	используется в списке импорта
EXPORT	список экспорта (используется в локальных модулях)
QUALIFIED	уже отмерший механизм квалифицированного экспорта
IMPORT	список импорта
BEGIN	начало модуля/процедуры
END	конец модуля/процедуры/оператора
CONST	раздел описания констант
TYPE	раздел описания типов
VAR	раздел описания переменных
PROCEDURE	раздел описания процедур

IF	оператор **IF**
CASE	**оператор CASE, вариантная запись**
LOOP	**оператор LOOP**
WHILE	**оператор WHILE**
REPEAT	**оператор REPEAT**
FOR	**оператор FOR**
WITH	**оператор WITH**
THEN	**используется в IF**
ELSE	**используется в IF, CASE, в вариантных записях**
ELSIF	**используется в IF**
UNTIL	**используется в REPEAT**
OF	**используется в ARRAY, CASE, в вариантных записях**
TO	**используется в FOR, POINTER**
BY	**. используется в FOR**
DO	**используется в WHILE, WITH, FOR**
EXIT	выход из LOOP
RETURN	выход из процедуры
HALT	завершение программы
CHAR	**тип CHAR**
BOOLEAN	**тип BOOLEAN**
CARDINAL	**тип CARDINAL**
INTEGER	**тип INTEGER**
REAL	тип **REAL**
LONGCARD	**тип LONGCARD**
LONGINT	**тип LONGINT**
LONGREAL	**тип LONGREAL**
ARRAY	**массив**
RECORD	**запись**
POINTER	**указатель**
SET	**множество**
FALSE	логическая ЛОЖЬ
TRUE	**логическая ИСТИНА**
NIL	несуществующий адрес (указатель)
PROC	процедура без параметров
AND	логическое И
OR	логическое ИЛИ
IN	принадлежит ли элемент множеству
DIV	целочисленное деление
MOD	остаток от целочисленного деления
INC	инкрементирование
DEC	декрементирование
INCL	добавить элемент во множество
EXCL	исключить элемент из множества
ODD	является ли нечетным число
HIGH	**старший индекс открытого массива**
ABS	взятие модуля числа
CAP	преобразование в заглавные буквы
NEW	запросить элемент динамической памяти
DISPOSE	вернуть элемент динамической памяти
ORD	преобразование типов CHAR и перечисления в CARDINAL
CHR	преобразование CARDINAL в CHAR
FLOAT	преобразование CARDINAL/INTEGER в REAL
TRUNC	преобразование REAL в INTEGER
VAL	универсальное структурное преобразование типов

Что же касается TopSpeed, то Вы имеете возможность использовать кроме букв и цифр подчеркивание _, знак доллара $ и знак @. Последние два символа используются компилятором для формирования имен процедур (модуль$процедура) и переменных (модуль@переменная). Подчер-

кивание полезно для интерфейса с внешними именами из других языков, а также для формирования собственных составных идентификаторов.

3. Типизация

Типизация — это краеугольный камень любого языка. В Modula-2 используется следующий принцип. Имеется фиксированный набор базовых типов. Каждый из этих типов является неделимым. Далее язык предоставляет несколько конструкторов, которые позволяют строить новые типы на основе базовых. Затем Вы можете создавать новые типы просто путем объявления нового имени для уже существующего типа (неважно, базового или составного). Наконец, за счет концепции модуля есть возможность создавать так называемые абстрактные типы данных, для каждого из которых кроме структуры можно определять еще и набор допустимых операций.

3.1. Базовые типы

С каждым типом данных связаны следующие атрибуты:

- размер элемента (выражается в байтах);
- диапазон допустимых значений;
- определенный набор операций.

В результате выполнения операции, как правило, получается значение того же самого типа. Исключение составляют лишь операции сравнения. В этом случае происходит переход в тип BOOLEAN (логический тип).

Тип	Размер	Диапазон значений	Операции
BYTE	1	OH..0FFH	=, #
WORD	2*	OH..0FFFFH	=, #
BITSET	2*	{}..{0,...,15}	=, #, INCL, EXCL, IN, +, -, *, /
ADDRESS	4*	..	=, #
CHAR	1	OC..377C	=, #, >, <, >= <=
BOOLEAN	1	FALSE..TRUE	=, #, AND, OR, NOT
CARDINAL	2	0..65535	=, # >, <, >=, <=, +, -, *, DIV, MOD
INTEGER	2	-32768..32767	=, #, >, <, >=, <=, +, -, * DIV, MOD
REAL	4	-3.4E+38..3.4E+38	=, # >, <, >=, <=, +, -, *, /
LONGCARD	4	0..4294967295	=, #, >, <, >= <=, +, -, * DIV, MOD
LONGINT	4	-2147483648..2147483647	=, #, >, <, >= <=, +, - *, DIV, MOD
LONGREAL	8	-1.7E+308..+1.7E+308	=, #, >, <, >=, <=, +, -, *, /

* означает зависимость от конкретного процессора (8/16/32 бита) и/или модели памяти.
- встроенной операции возведения в степень в языке Modula-2 нет.

Для записи операций можно использовать следующие эквиваленты:

AND		&
NOT	-	~
#	-	<>

Из всех базовых типов, мне кажется, незнакомыми Вам могут показаться лишь BITSET и ADDRESS. Тип BITSET очень удобен для работы с битами. В этом случае машинное слово (тип WORD) рассматривается как множество, состоящее максимум из 16 элементов (битов). Если бит выставлен, то множество содержит данный элемент, если же нет — то не содержит. Итак, BITSET рассматривается как тип:

```
TYPE  BITSET = SET OF WORD;
```

Правда, такое определение, данное Виртом, вносит серьезную путаницу. Если быть более точным, то тип BITSET лучше рассматривать как SET OF [0. .15], тем более, что запись SET OF WORD язык не допускает. К тому же конструктор SET опирается на диапазон значений опорного типа, а не на машинное представление его элемента. Пусть у нас есть следующее описание:

```
VAR a,b,c: BITSET;
```

Для работы с BITSET, как и с любым типом множества, помимо традиционных операций сравнения на равенство = и неравенство #, используются операции:

INCL	выставить указанный бит	INCL(a,5);
EXCL	сбросить указанный бит	EXCL(a,1);
IN	выставлен ли указанный бит	IF (3 IN a) THEN
+	побитовое ИЛИ (OR)	c := a + b;
-	побитовое И с дополнением b	c := a - b;
*	побитовое И (AND)	c := a * b;
/	исключающее ИЛИ (XOR)	c := a / b;

Что касается типа ADDRESS, то его значения машиннозависимы. Имеется специальная константа NIL, которая означает несуществующий адрес. Точное ее значение языком не регламентируется. Тип ADDRESS рассматривается в языке как указатель на тип WORD, из чего следует допустимость операции разыменования. Для адресной арифметики можно использовать только специальные процедуры, которые либо включаются в модуль SYSTEM, либо выносятся в другой внешний модуль. Первые четыре типа, а именно, BYTE, WORD, BITSET и ADDRESS, находятся в специальном модуле SYSTEM и напрямую недоступны (для TopSpeed это не так -- вы можете их использовать безо всякого импорта из SYSTEM).

Помимо указанных типов в TopSpeed Modula-2 имеются еще такие базовые типы, как SHORTINT (INTEGER в байте) и SHORTCARD (CARDINAL в байте), а также FarADDRESS (дальний указатель, аналог NIL — FarNIL) и NearADDRESS (ближний указатель, аналог NIL — NearNIL). Тип ADDRESS компилятор рассматривает как FarADDRESS, либо как NearADDRESS, в зависимости от модели памяти. Кроме того, имеется аналог типа WORD для 32-битного представления — тип LONGWORD. TopSpeed Modula-2 предоставляет также дополнительные операции по работе с битовым представлением чисел для типов SHORTCARD, CARDINAL и LONGCARD — это операции арифметического сдвига

i << j арифметический сдвиг *влево* на j битов (умножение на 2 в степени j)

i >> j арифметический сдвиг *вправо* на j битов (деление на 2 в степени j)

3.2. Конструкторы типов

Конструкторы служат для построения составных типов из базовых, либо из других составных типов. Обычно это понятие часто путают с понятием типа, хотя интуитивно ясно, что конструктор независимо от опорного типа использоваться не может. Наличие конструкторов позволяет программисту строить требуемую иерархию типов данных. При этом нужно понимать, что далеко не все конструкторы допускают произвольную глубину описания.

* Диапазон (subrange). Новый тип создается как ограничение значений уже существующего. Очень интенсивно используется при задании типа индекса в массиве. Диапазон можно задавать только для так называемых скалярных типов, т.е. типов с точными значениями. Типы REAL и LONGREAL скалярными не являются.

```
TYPE S = [1..4];
```

*** Перечисление (enumeration).** Строит новый тип из констант, которые имеют только имена, но "не имеют" конкретного значения.

```
TYPE E = (Red,Green,Blue);
```

На самом деле на уровне машинного представления этого типа каждой константе присваивается вполне определенное значение (Red=0, Green=1, Blue=2). Существует стандартная процедура ORD, которая позволяет осуществить структурное преобразование перечисления в тип CARDINAL. Вообще говоря, имена, используемые при задании перечисления, нужно рассматривать как константы типа CARDINAL.

Любой тип перечисления занимает в TopSpeed по умолчанию 2 байта. Изменить это можно с помощью прагмы компилятора data (var_enum__size => off) .

*** Массив (ARRAY).** Обычный массив. В качестве индексов могут быть элементы любого скалярного типа, например, типа, полученного с использованием перечисления. Кроме того, можно описывать многомерные массивы. Главное требование — компилятор должен точно знать количество элементов массива, поскольку память под него отводится статически.

```
TYPE
  E  = (Red,Green,Blue);
  S  = [Red..Blue];
  A1 = ARRAY [0..7] OF BYTE;
  A2 = ARRAY S OF BYTE;
  A3 = ARRAY [1..4] OF ARRAY [0..15] OF CARDINAL;
  A4 = ARRAY [1..4], [0..15] OF CARDINAL;
```

Последние два описания равнозначны.

*** Запись (RECORD).** Аналог записей (RECORD) в языке Pascal и структур (struct) в языке C. Запись состоит из фиксированного числа полей, каждое из которых имеет имя и обозначает элемент конкретного типа. Доступ к элементу поля обозначается через точку:

```
TYPE
  HeaderRec = RECORD
                magic : CARDINAL;
                name  : ARRAY [0..15] OF CHAR;
                id    : CARDINAL
              END;
VAR
  h : HeaderRec;
  . . .
h.magic := 123;
h.id    := 15;
```

В одной и той же записи не могут быть поля с одинаковыми именами. Для разных же записей это допустимо.

Для работы с полями в TopSpeed предусмотрена специальная процедура FieldOfs, которая позволяет получить смещение поля от начала записи.

*** Указатель (POINTER).** Указатели, как правило, нужны для работы с динамическими типами данных, которые строятся самим программистом для конкретной задачи. Указатель задает ссылку на другой тип (элемент другого типа). В принципе он представляет собой машинный адрес этого элемента. Все указатели имеют одну общую константу NIL, которая означает пустой указатель (несуществующий адрес). Конкретное значение этой константы в языке не регламентируется! Как правило, указатели используются вместе с соответствующими записями, причем очень часто возникает рекурсия описания. В этом случае задание указателя должно быть первичным по отношению к соответствующей записи.

```
TYPE
  List    = POINTER TO ItemRec;
  ItemRec = RECORD
              next : List;
              prev : List;
              body : CARDINAL
            END;
```

Для доступа к элементу, на который ссылается указатель, используется символ ^ (эта операция носит название операции *разыменования).*

```
VAR
  ptr : List;
. . .
ptr^.next := ptr^.next^.next;
ptr^.next^.prev := ptr;
ptr^.body := 13;
```

*** Множество (SET).** Этот конструктор обычно нужен довольно редко. Исключение составляют работа с битами (BITSET) и битмапами.

```
TYPE
  Register = BITSET;
  Line     = ARRAY [1..640 DIV 16] OF BITSET;
  BitMap   = ARRAY [1..480] OF Line;
  CharSet  = SET OF CHAR;
VAR
  reg  : Register;
  i    : CARDINAL;
  map  : BitMap;
  oper : CharSet;
. . .
IF (0 IN reg) THEN INCL(map[100,4],1) END;
oper  :=  CharSet{'+','-','*','/'};
```

В качестве базового типа для конструктора SET в соответствии с Виртом может выступать любой скалярный базовый тип, а также диапазон и перечисление. При использовании конструкции вида SET OF T считается, что тип T определяет тип возможных значений отдельного элемента множества, а не память, отводимую под представление множества. Обратите внимание, что элементы множества заключаются в фигурные скобки, а перед ними ставится имя типа.

*** Представление строк.** Для строк в языке нет выделенного типа. Для представления используется массив символов ARRAY [O..N-1] OF CHAR. Строки в Modula-2 фактически аналогичны строкам в языке C: конец строки обозначается символом OC, если, конечно для этого символа хватает места в массиве. Встроенных операций работы со строками в Modula-2 нет, все эти функции возлагаются на внешние библиотечные модули. Конечно же, имеется такая операция, как присваивание переменной строковой константы.

```
TYPE
  FileName = ARRAY [0..11] OF CHAR;
VAR
  file : FileName;
. . .
file := "README.TXT";
```

Для нормальной работы со строками нужны соответствующие библиотечные процедуры.

*** Процедурный тип.** Строго говоря, это не конструктор типов, в то же время к базовым типам его также нельзя отнести. Процедурный тип позволяет задать новый тип на основе описания интерфейса процедуры. В этом интерфейсе указывается тип формальных параметров и способ их передачи.

```
TYPE
  MyProcType = PROCEDURE ( CARDINAL );
VAR
  a,b: MyProcType;
. . .
a := CardToScreen;
b := CardToFile;
a(1);b(5);
```

Существует также зарезервированный тип PROC, который задает процедуру без параметров и соответствует описанию:

```
TYPE  PROC = PROCEDURE;
```

Процедурные типы очень полезны на практике. С их помощью можно, например, довольно изящно решать проблемы перенаправления ввода-вывода.

3.3. Определение новых типов

Определение типов дается в специальном разделе, начинающемся с TYPE. Такие разделы могут идти друг за другом и перемежаться с другими описательными разделами. Описание типов может быть не только внутри модулей, снаружи (в их интерфейсах), но и внутри процедур. Однако, вряд ли последнее дает какое-то полезное преимущество.

Наверное, не стоит объяснять, зачем нужно определять новые типы данных. Главным критерием при создании нового типа должна быть его практическая необходимость. Не надо злоупотреблять этим механизмом и начинать описывать все подряд, вплоть до выделения специальных типов под индексы массивов.

3.4. Константы и переменные

Описание констант и переменных осуществляется в соответствующих разделах описания, начинающихся со слов CONST и VAR. Для них справедливо все то, что говорится про определение новых типов. Однако, на некоторые нюансы стоит обратить внимание. Константы могут быть простые и структурные. Для простых достаточно просто написать их значение (точное, или вычисляемое с помощью константного выражения). Со структурными дело обстоит несколько сложнее. Во-первых, в разных компиляторах это делается по-разному, но в любом случае Вы должны некоторым образом указать тип, к которому относится константа, и перечислить значения составляющих ее простых компонент. В TopSpeed это выглядит так.

```
TYPE
  Color = ARRAY [1..3] OF CARDINAL;
CONST
  Black = Color(0,0,0);
  Red   = Color(255,0,0);
  White = Color(255,255,255);
```

Другой важный аспект, касающийся констант, - это специальная запись восьмеричных, шестнадцатеричных и символьных констант. Рассмотрим пример.

```
CONST
  MAGIC = OFFFFH;
  DEL   = 177B;
  LF    = 12C;
```

MAGIC - это шестнадцатеричная константа со значением FFFF (десятичное значение 65535). Лидирующий ноль обязателен, иначе константа будет считаться уже идентификатором. Буква"H"обозначает, что предшествующие ей буквы и цифры соответствовали шестнадцатеричному представлению числа.

DEL - восьмеричная константа (ей соответствует десятичное значение 127). Буква"B", как пояснил Вирт, была выбрана потому, что она напоминает цифру 8.

LF - это символьная константа (об этом говорит буква "C"). Она обозначает символ, который в кодовой таблице имеет *восьмеричный* номер 12.

Для обозначения строковых констант можно использовать либо одинарные, либо двойные кавычки. Есть хорошее правило. Если Вы используете кавычки для обозначения символьных констант (одиночные элементы типа CHAR), то применяйте одинарные. Для обозначения строчных используйте двойные. Это удобно еще и потому, что в Modula-2 строка и символ — это принципиально разные и несовместимые вещи.

3.5. Совместимость и эквивалентность

Это — один из самых сложных и весьма запутанных вопросов не только в языке Modula-2, но и в самой теории языков программирования. Сразу оговоримся, что в данном разделе будет излагаться наша точка зрения на этот вопрос, которая, хоть и помогает нам на практике в большинстве ситуаций, все же носит дискуссионный характер.

Итак, все типы данных — вне зависимости от того, простые они, составные или произвольные (абстрактные) — помимо уникального имени и множества операций характеризуются еще и набором *допустимых значений.* Причем вполне возможны такие ситуации, когда некоторые значения одного типа имеются и у другого. В языке Modula-2 примером этого могут служить типы CARDINAL и INTEGER. Типы, имеющие хотя бы одно совпадающее значение, мы будем считать *совместимыми.* При этом для нас абсолютно неважно, какой размер (в битах, в байтах, в словах) занимает это значение в том или ином типе.

С точки зрения положения конкретного типа среди остальных не менее важным является понятие эквивалентности. Мы выделяем три разновидности эквивалентности: *именную, структурную* и *размерную.* Такая классификация делит все типы на четыре группы. В первую входят типы, носящие одно и то же уникальное (а не относительное внутри модуля) имя. Следующая группа - типы с одинаковой структурой. Обычно это составные типы, построенные с помощью одинаковых конструкторов (массивы, записи, диапазоны). Ясно, что типы, эквивалентные по именам, также эквивалентны и по структуре. За ними идет группа типов с одинаковыми размерами своих элементов. Яркий пример — типы BITSET и CARDINAL, занимающие в точности 2 байта. Также вряд ли вызовет сомнение то утверждение, что эквивалентные по структуре типы эквивалентны и по размеру. Наконец, в последней группе оказываются неэквивалентные типы, элементы которых обладают разными размерами, не говоря уже об остальном.

На практике тип данных редко когда может быть обособлен и существовать сам по себе. В большинстве случаев требуется осуществлять перевод элемента одного типа данных в другой. Достигается это с помощью операций (функций, операторов) приведения и преобразования. *Приведение* типов (type coercion) мы рассматриваем как переход к именной эквивалентности. *Преобразование* же (type conversion) — как переход к структурной или размерной эквивалентности с другим типом. Рассмотрим пример. Пусть у нас описаны два типа с именами T1 и T2.

```
TYPE
  T1 = CARDINAL;
  T2 = CARDINAL;
```

Эти два типа имеют два разных имени, а потому они уже РАЗНЫЕ. Однако, и тот, и другой с точки зрения структуры одинаковы, поскольку определяются через общий базовый тип CARDINAL. Теперь

Вам должно быть понятно, в чем разница между именной и структурной эквивалентностью.

В языке Modula-2 поддерживается строгая типизация. Это означает, что нельзя смешивать между собой объекты разных типов. Имеются три основных **ситуации,** где у программиста могут возникать проблемы с совместным использованием переменных и констант различных типов. Это смешанные выражения, операторы присваивания и передача параметров процедурам. Modula-2 не допускает использование в выражениях объектов разных типов. Значение всего выражения должно быть одного вполне определенного типа, а все объекты, которые ему не соответствуют, должны приводиться к этому типу. Для того чтобы выполнить приведение структурно эквивалентных типов, достаточно просто указать, к какому типу приводится данный объект. Если типы не эквивалентны по структуре, требуется уже воспользоваться своей, либо одной из стандартных процедур преобразования. Поясним это на примере.

```
VAR
  a: T1;
  b: T2;
  c: INTEGER;
. . .
a := 1;
b := 4;
c := INTEGER(a + T1(b));
```

Здесь используются две переменные разных типов — T1 и T2. Как видно, в выражении a + T1 (b) используется приведение. Другими словами, программист этой записью дает понять компилятору, что он видит, что Ь другого типа, но считает, что все выражение имеет тип T1.

На практике крайне неудобно все время заниматься приведением. Поэтому Вирт ввел понятие *"совместимостъ по присваиванию"* (assignment compatibility). Типы могут быть разными, но если они совместимы по присваиванию, то требование именной эквивалентности ослабляется. Возникает справедливый вопрос, а какие же типы совместимы по присваиванию? Если говорить о базовых, то это INTEGER и CARDINAL, а также LONGINT и LONGCARD. Что касается составных, то все они несовместимы по присваиванию, за исключением случая, когда один описан как диапазон другого, либо когда оба описаны как диапазоны третьего.

Другая ситуация — это передача параметров процедурам. Как быть, если формальный параметр, описанный в процедуре, имеет один тип, а фактический, передаваемый Вами, — другой? Этот вопрос будет затронут в разделе о процедурах. Однако, основу здесь составляет все та же именная эквивалентность. Для преобразования типов нужно пользоваться стандартными процедурами (VAL, ORD, CHR, FLOAT, TRUNC). Помимо этого Вы можете применять свои методы и приемы. Рассмотрим один из них.

```
TYPE
  Union = RECORD
            CASE : CARDINAL OF
              0 : byte: ARRAY [0..3] OF BYTE
            | 1 : char: ARRAY [0..3] OF CHAR
            | 2 : bool: ARRAY [0..3] OF BOOLEAN
            | 3 : word: ARRAY [0..1] OF WORD
            | 4 : bit : ARRAY [0..1] OF BITSET
            | 5 : card: ARRAY [0..1] OF CARDINAL
            | 6 : int : ARRAY [0..1] OF INTEGER
            | 7 : adr : ADDRESS
            END
          END;
VAR
  unit : Union;
. . .
INCL(unit.bit[0],0);
IF (unit.int[0] < 0) THEN . . .
```

Здесь одни и те же ячейки памяти рассматриваются как разные типы. Для "наложения" типов используется вариантная запись с пустым полем признака (:CARDINAL). Размер всей записи равен размеру максимального по длине поля (4 байта). При доступе к элементам нужно точно знать, как они расположены в памяти компьютера. Для PC-компьютеров правило простое — сначала младшее слово, затем старшее слово; в каждом слове — сначала младший байт, затем старший байт.

Вообще говоря, хорошая идея разделения понятий структурной и именной эквивалентности при четком ее соблюдении компилятором здорово раздражает. Ведь цель ее введения состояла в том, чтобы уже на этапе компиляции, а не выполнения программы, когда цена ошибок крайне высока, предупредить программиста о возможной ошибке. И часто это помогает. Однако, большую часть проблем для программиста составляет постоянное приведение типов. Конечно, лучше было бы в самом языке предусмотреть это. Так, к примеру, в языке Modula-3, разработанному на базе Modula-2 в Systems Research Center фирмы DEC, эта проблема решена очень просто. Когда Вы описываете новый тип, то он считается эквивалентным любому эквивалентному по структуре типу. Если же Вы хотите его выделить, то с помощью специального атрибута описания Вы делаете его уникальным, заставляя уже работать механизм именной эквивалентности. На практике же компиляторы Modula-2 самостоятельно принимают решение об ослаблении жестких правил.

4. Основные программные конструкции

Не секрет, что любая программа состоит из данных и кода. С данными языка мы уже познакомились. Теперь рассмотрим, как же строится код. Как и в других языках, в Modula-2 есть выражения и операторы. Очень важно четко различать эти два понятия. Любое выражение всегда имеет значение вполне определенного типа. Оператор же представляет собой законченное действие и ни к какому типу не относится. Если провести параллель с естественным языком, то выражение можно рассматривать как фразу языка, а оператор — как целое предложение. Итак, выражение — это незаконченный оператор.

4.1. Выражения

Выражение состоит из операндов и операций. В роли операндов могут выступать константы, переменные, вызовы функций и, в свою очередь, другие выражения. Операции специфичны для каждого типа операндов. В языке Modula-2 выражения обычно используются в правой части оператора присваивания и в операторе IF.

Строго говоря, в Modula-2 при вычислении выражений используется приоритет операций, который совпадает с интуитивным, однако, лучше строго задавать приоритет с использованием обычных круглых скобок. Поверьте, это сэкономит Вам много нервов при отладке.

У выражений нужно обратить внимание еще на один интересный момент. Если Ваше выражение логическое, то оно при вычислении слева направо может не использовать всех входящих в него выражений, если результат и так очевиден. Если мы используем конъюнкцию (логическое И) и первый операнд дает значение FALSE, то остальные можно и не вычислять — результат все равно будет FALSE. В случае дизъюнкции (логическое ИЛИ) достаточно узнать, что первый операнд имеет значение TRUE, как тут же можно прекращать вычисления — результат также будет TRUE. Мы не зря остановились так подробно на этом вопросе, потому что часто возникают ситуации, когда незнание такой семантики вычислений в Modula-2 заставляет программиста "на всякий случай" использовать лишние операторы, затеняющие логику алгоритма. Рассмотрим простой пример.

```
VAR
  s : ARRAY [0..9] OF CHAR;
  i : CARDINAL;
```

```
i := 0;
WHILE  (i <= 9) & (s[i] # OC)  DO
  s[i]  :=  '*';
  INC(i)
END;
```

Здесь строка просто заполняется одним и тем же символом *. Давайте внимательнее посмотрим на предусловие цикла WHILE. Предусловие представляет собой логическое выражение. Все выглядит правильно, однако, если бы в Modula-2 была другая семантика вычислений логических выражений, то это могло бы привести к выходу индекса за границы диапазона при вычислении выражения s[i] # OC.

4.2. Операторы

Операторы — это основные строительные блоки кода любой программы. Все операторы языка Modula-2 можно разбить на следующие группы: присваивание, вызов процедуры, конструкторы управления, оператор WITH, операторы завершения.

*** Оператор присваивания.** Самый простой и не требует пояснений. Имеет единственную форму записи :=. В левой части может быть только переменная.

*** Вызов процедуры.** Тоже ничего нового. Разве что необычными могут показаться вызовы процедур через процедурные переменные.

*** Конструкторы управления.** Это целый набор операторов, позволяющий описать на языке три основные конструкции структурного программирования — последовательность, ветвление и цикл.

*** Оператор WITH.** Этот оператор стоит особняком среди остальных. Он дает удобные средства для работы с записями. Каждое поле записи имеет как относительное имя внутри записи, так и абсолютное имя, которое строится из префикса в виде имени переменной и точки. Если Вы не хотите постоянно повторять в программе, что обращаетесь к одной и той же переменной, а хотите иметь доступ к полям по относительным именам (читай — по их смещениям), то это очень удобное средство. К тому же оно позволяет компилятору генерировать эффективный код. Сравните два функционально аналогичных фрагмента программы.

```
TYPE
  List    = POINTER TO ItemRec;
  ItemRec = RECORD
              next : List;
              prev : List;
              body : CARDINAL
            END;
VAR
  list : List;

list^.next := NIL;
list^.prev := NIL;
list^.body := 13;

WITH  list^  DO
  next := NIL;
  prev := NIL;
  body := 13
END;
```

Не следует, однако, забывать о том, что пользоваться оператором WITH нужно аккуратно. Если у Вас имеются идентификаторы, имена которых совпадают с именами полей, то внутри оператора WITH компилятор будет в растерянности. Таких ситуаций лучше не допускать.

* Операторы завершения. Сюда относятся три оператора: EXIT — выход из безусловного цикла (оператора LOOP); RETURN — выход из любой точки процедуры и HALT — завершение программы. Первые два оператора используются весьма интенсивно. Если говорить о последнем, то он мне не потребовался ни разу. Имеются другие средства и приемы добиться аналогичного эффекта.

5. Конструкторы управления

Поскольку Modula-2 является классическим представителем идеи структурного программирования, то все его конструкторы управления можно четко разбить на три группы: последовательность, ветвление и цикл.

5.1. Последовательность

Последовательность операторов описывается на языке Modula-2 с помощью точки с запятой, которая отделяет друг от друга операторы одного уровня. Следующий оператор последовательности совсем не обязательно начинать с новой строчки: используя разделитель (точку с запятой), Вы можете располагать несколько операторов последовательности на одной и той же строке.

```
IF  (x > 0)  THEN
  IF  (y < 15)  THEN
    y  := 0;
    x  := 0;
    found  := TRUE
  END;
END;
```

Операторы x := 0 и found := TRUE — это операторы одного уровня. Между ними обязательно должна стоять точка с запятой. После оператора found := TRUE точка с запятой не стоит, поскольку END относится к оператору IF, объемлющему found := TRUE, а потому — оператору другого уровня. Это понятно. Вопрос: почему же тогда ставится точка с запятой между END одного IF и END другого IF? В языке Modula-2 эта запись в самом деле допустима. И вот почему. Существует понятие пустого оператора. Т.е. в принципе Вы можете писать:

```
x := 0;  ;;;    found := TRUE
```

Теперь Вам все должно быть понятно. Этот маленький нюанс можно использовать с выгодой для себя. Итак, если Вы не хотите, чтобы компилятор Вас часто поправлял, то, не задумываясь, ставьте точку с запятой после каждого оператора. Однако, очень часто опытные программисты допускают ошибки не в момент написания кода, а в момент его ручной сборки из уже написанных и работающих кусков. Здесь при компоновке чужеродных фрагментов потенциально и могут крыться ошибки, которые Вам придется искать очень долго. Если же Вы не будете использовать пустые операторы, а станете четко следовать общему правилу, то появится вероятность того, что из-за синтаксической ошибки Вы более пристально посмотрите на заимствованный фрагмент и выявите ошибку на самой ранней стадии.

5.2. Ветвление

В языке Modula-2 всего два оператора ветвления -- IF и CASE. Оператор IF почти ничем не отличается от традиционной трактовки (новое, разве что, ELSIF) и имеет следующие формы:

```
IF (...) THEN . . . END;
IF (...) THEN . . . ELSE . . . END;
IF (...) THEN . . . ELSIF (...) THEN . . . ELSIF (...) THEN . . . END;
IF (...) THEN . . . ELSIF (...) THEN . . . ELSIF (...) THEN . . . ELSE . . . END;
```

Оператор CASE во многом схож с оператором switch в языке C, однако, не забывайте о маленьком и существенном отличии. Если в C для прекращения дальнейшего просмотра вариантов нужно постоянно использовать break, то в Modula-2 это делается автоматически. С точки зрения языка если Вы опустите ELSE-часть оператора, а тестируемому значению переменной CASE не будет соответствовать ни один вариант, то это неизбежно приведет к ошибке на этапе выполнения. Компилятор такую ошибку не найдет. Пример:

```
TYPE
  Color = (Red,Green,Blue);
VAR
  color: Color;
. . .
CASE color OF
  Red   : Write("Красный")
| Green : Write("Зеленый")
| Blue  : Write("Синий")
ELSE
END;
```

В данном случае ELSE-часть не нужна, но лучше ее ставить всегда. Как видно из примера, варианты отделяются друг от друга символом | .

5.3. Цикл

Modula-2 предоставляет богатый набор операторов цикла. Все операторы цикла можно классифицировать следующим образом:

- безусловный цикл (LOOP);
- цикл с предусловием (WHILE);
- цикл с постусловием (REPEAT);
- цикл с фиксированным числом итераций (FOR).

Наиболее универсальным является оператор безусловного цикла (LOOP) , с помощью которого можно смоделировать любой другой цикл. Выход из цикла происходит с помощью оператора EXIT, который, как правило, используется в каком-то операторе ветвления внутри цикла LOOP. Оператор с предусловием (WHILE) удобен тогда, когда проверка условия выхода из цикла должна производиться в начале цикла. Оператор с постусловием (REPEAT) нужен при проверке условия выхода в конце цикла. Наконец, оператор FOR используется для циклов с фиксированным числом итераций. Этот цикл использует управляющую переменную цикла (скалярного типа) с начальным значением. Эту переменную нельзя менять иначе, как с помощью атрибута BY в описании цикла (по умолчанию используется BY 1).

Часто программисты ошибаются при использовании операторов WHILE и REPEAT. Это связано с тем, что условие, стоящее в операторе WHILE - это условие *продолжения* цикла, а условие в операторе REPEAT (UNTIL-часть) — это условие *выхода* из цикла.

Чтобы разобраться в особенностях операторов цикла, давайте рассмотрим пример, который описывает по сути одно и то же.

```
VAR
  i     : CARDINAL;
  s     : ARRAY [0..N-1] OF CARDINAL;
  found : BOOLEAN;

i := 0;  found := FALSE;
LOOP
  IF (i >= N) THEN EXIT END;
  IF (s[i] = 1) THEN
    found := TRUE;
    EXIT
  END;
  INC(i);
END;

i := 0;  found := FALSE;
WHILE (i < N) & ~found DO
  IF (s[i] = 1) THEN found := TRUE
                ELSE INC(i)
  END
END;

i := 0;  found := FALSE;
REPEAT
  IF (s[i] =1) THEN found := TRUE
               ELSE INC(i)
  END
UNTIL found OR (i >= N) ;

found := FALSE;
FOR i := 0 TO N-1 DO
  IF (s[i] = 1) THEN found := TRUE END
END;
```

В данном примере цикл FOR не совсем удачен: в случае быстрого обнаружения нужного элемента будут избыточные итерации.

Есть хорошее правило при использовании циклов. Если Вам нужен цикл и Вы сомневаетесь, какой использовать, всегда применяйте LOOP. Впоследствии, проанализировав его, Вы сможете заменить LOOP на другую форму.

С циклом LOOP связан еще один прием программирования. Итак, у Вас есть цикл, в середине которого по некоторому условию нужно вернуться на его начало. В языке Modula-2 принципиально отсутствует такое средство как goto. Как можно просто это смоделировать? Взгляните на скелет соответствующего программного фрагмента.

```
VAR
  exit: BOOLEAN;
. . .
exit := FALSE;
LOOP
  LOOP
    IF (...) THEN EXIT END;
    . . .
    IF (...) THEN
      exit := TRUE;
      EXIT
    END;
    . . .
  END;
```

```
  IF  exit  THEN  EXIT  END
END;
```

Этим же приемом можно воспользоваться в ситуации, когда имеется некоторая последовательность операторов, но при определенных условиях нужно игнорировать оставшиеся операторы этой последовательности. А строить изощренные конструкции с помощью операторов ветвления не хочется. Достигается это с помощью *фиктивного* цикла.

```
LOOP
  . . .;
  IF  (...)  THEN  EXIT  END;
  . . .;
  EXIT
END;
```

Обратите внимание, в конце фиктивного цикла нужно не забывать ставить EXIT. Безусловно, этим приемом нужно пользоваться довольно аккуратно.

При использовании TopSpeed не применяйте много вложенных FOR-циклов. Практика показала, что компилятор иногда некорректно генерирует код для таких структур.

6. Программные блоки

Программные блоки в языке Modula-2 состоят из процедур и модулей. Здесь очень важным являются два понятия — *область видимости* и *область существования.*

Итак, Вы описали некоторые объекты программы. Сколько времени они существуют? Ответ прост — все то время, пока активен тот программный блок, в котором они объявлены. В случае процедур это понятно. Как только осуществляется вызов процедуры, в стеке резервируется место под локальные переменные. Когда процедура завершается, выделенная память возвращается. Для модулей (забудем про локальные модули!) перед выполнением первого оператора основного модуля (то бишь программы) производится распределение памяти под используемые библиотечные модули и соответственно их инициализация. Таким образом, объекты, описанные в модулях, существуют все то время, пока не завершится Ваша программа.

Область видимости идентификаторов (scope) тоже не сложна для восприятия. Все процедуры можно рассматривать как прозрачные модули. Иными словами, все идентификаторы, описанные снаружи процедуры, автоматически доступны и внутри нее. Обратное, конечно, не верно. Если проводить аналогию с реальной жизнью, то процедуры — как зеркальные очки. Они не имеют никаких средств для управления областью видимости. Модули же за счет механизма экспорта-импорта могут четко специфицировать, что просачивается извне и что доступно снаружи. В связи с этим в языке исчезает понятие глобальных объектов.

6.1. Процедуры и функции

Понятие процедуры в языке Modula-2 мало чем отличается от традиционного. Процедуры могут быть вложены одна в другую, и Modula-2 разрешает использовать рекурсивные вызовы. Есть, правда, три важных момента — способы передачи параметров, открытые массивы и процедурные типы.

Modula-2 поддерживает два способа передачи параметров — по *значению* и по *ссылке.* При передаче параметров по значению фактические параметры копируются в стек и процедура работает с их копиями. При возврате из процедуры обратного копирования не происходит, и стек процедуры очищается. Таким образом, при передаче по значению Вы работаете с параметрами только на

чтение. В случае передачи по ссылке Вы работаете с оригиналами, а потому любые изменения параметров внутри процедуры сохранятся и после ее завершения. Здесь работа с параметрами производится и на чтение, и на запись.

```
PROCEDURE  Square ( x: REAL; VAR sq: REAL );
BEGIN
  sq := x * x
END  Square;
```

В процедуре Square вычисляется квадрат числа. Параметр x является входным и передается по значению, а параметр sq является выходным и передается по ссылке (ключевое слово VAR).

Помимо основной формы процедуры имеют еще и другую, которая называется *функцией.* Функция отличается от процедуры тем, что кроме выполнения указанных в ней действий она еще и возвращает через свое имя результат некоторого типа. Наш предыдущий пример можно переписать так.

```
PROCEDURE  Square ( x: REAL ): REAL;
BEGIN
  RETURN  (x * x)
END  Square;
```

Здесь важно знать, какие типы можно использовать в качестве результата функции. Это - все базовые, а также указатели. Записи и массивы результатом функции быть не могут. Для возврата результата нужно обязательно использовать оператор RETURN. Если функция не имеет параметров, то при использовании ее в выражении нужно обязательно ставить пустые скобки. При вызове процедур этого делать не нужно.

```
ok := Initialized();
```

Для того чтобы можно было писать библиотечные процедуры, которые не привязываются к точному числу элементов массива, передаваемому в виде параметра, в язык введена специальная конструкция, которая носит название *открытый массив* (open array). Она позволяет работать с унифицированной формой представления массива. Ведь проблема не только в том, что массивы могут иметь различную длину, но и в том, что диапазоны индексов у них могут быть различны (первый номер совершенно не обязательно должен быть нулем). В Modula-2 эта проблема решается так.

Вы передаете в процедуру в качестве фактического параметра некоторый одномерный (!) массив. В случае, если соответствующий формальный параметр — это открытый массив (скажем, a), то считается, что Ваш массив индексируется от 0 до HIGH (a) , где HIGH — встроенная в язык функция определения последнего индекса для открытого массива. Количество элементов массива, таким образом, равно HIGH(a)+1. Рассмотрим пример.

```
PROCEDURE  Sum (a: ARRAY OF REAL ): REAL;
  VAR  i: CARDINAL;
       s: REAL;
BEGIN
  s  :=  0.0;
  i := 0;
  LOOP
    s := s + a[i];
    INC(i);
    IF  (i > HIGH(a))  THEN  EXIT  END;
  END;
  RETURN  s
END  Sum;
```

Обратите внимание, что открытые массивы могут быть только одномерными. Это заметный недостаток языка. Еще, что важно знать о передаче параметров, так это то, что типы формальных и фактических параметров должны быть идентичны. Кроме того, в языке имеется возможность использовать конструкцию ARRAY OF WORD, которая совместима с *любым* типом фактического параметра. Так сказано в описании языка, однако на практике чаще используют запись ARRAY OF BYTE, которая более точно отражает суть механизма.

В языке Modula-2 введен специальный тип — *процедурный*. Для описания его достаточно специфицировать интерфейс процедуры без указания имен формальных параметров. Обратите внимание на синтаксис: типы параметров при описании процедурного типа идут не через точку с запятой, а через запятую.

```
TYPE
  IncDec = PROCEDURE ( VAR CARDINAL, CARDINAL );
VAR
  incCard, decCard : IncDec;
  i : CARDINAL;

PROCEDURE  Inc ( VAR x: CARDINAL; n: CARDINAL );
BEGIN
  x := x + n
END  Inc;

PROCEDURE  Dec ( VAR x: CARDINAL; n: CARDINAL );
BEGIN
  x := x - n
END  Dec;
. . .
incCard := Inc;  decCard := Dec;
incCard(i,1); decCard(i,5)
```

Как видно из примера, прежде чем обращаться к процедурной переменной, надо ее проинициализировать, т.е. связать с конкретной существующей процедурой, интерфейс которой в точности соответствует описанию процедурного типа.

6.2. Концепция модуля

Модули — самая, пожалуй, интересная часть языка. Да и названием своим язык обязан именно этому понятию (Modula — ***modular language)***. Благодаря этому средству Вы становитесь не только строителем, но и архитектором сложного здания под названием "программный комплекс". Модуль — это единица компиляции, которая заключает в себе описание объектов программы — типов, констант, переменных и процедур.

Модули могут быть двух видов: *основные* и *библиотечные*. В любой программе основной модуль (MODULE) может быть только один. Библиотечные модули состоят в точности из двух частей — интерфейса (DEFINITION MODULE) и его реализации (IMPLEMENTATION MODULE). Это — как две ветви власти: законодательная и исполнительная. В роли судебной выступают компилятор и компоновщик.

Для работы основному модулю, да и библиотечным, могут потребоваться объекты из других библиотечных модулей. Все те объекты, которые библиотечный модуль делает доступным для своих клиентов, фигурируют в интерфейсной части модуля. Константы и переменные здесь описываются полностью, как обычно. Процедуры — только в виде интерфейса, а типы — как правило, полностью, но могут быть представлены одним лишь своим именем. Весь интерфейсный модуль рассматривается как список экспорта этого библиотечного модуля. Модулю-клиенту, чтобы получить доступ к этим объектам, можно в списке импорта либо указать одно лишь имя данного модуля (IMPORT), либо уточнить, какие конкретно объекты из этого модуля ему требуются (FROM . . . IMPORT . . .). Практика показала, что удобнее всегда пользоваться первым способом импорта; исключение здесь может составлять разве что специальный модуль SYSTEM. Этот модуль занимает особое положение: он в точности заранее известен компилятору и из-за принципиальной невозможности выразить некоторые его объекты средствами языка (например, процедуру TSIZE), в большинстве реализаций Modula-2 он вообще не имеет ни интерфейсной, ни исполнительной части. В TopSpeed за счет понятия алиаса он имеет интерфейсную часть, что во многих случаях очень удобно.

Взгляните на пример. Для простоты изложения здесь опущены детали низкоуровневого программирования, которые зависят от конкретной компьютерной системы. Ниже приводятся два модуля: основной (BufTst) и библиотечный (Buffer). Если говорить об отображении модулей на конкретную файловую систему, то BufTst хранится в файле BUFTST.MOD, интерфейсная часть модуля Buffer — в файле BUFFER.DEF, а его реализация — в файле BUFFER.MOD.

```
MODULE BufTst;          (* простейший распределитель *)
  FROM SYSTEM IMPORT BYTE;
  IMPORT Buffer;

  TYPE
    Event = ( Read, Write, Stop );

  VAR
    event: Event;

  PROCEDURE Get ( VAR b: BYTE );
  BEGIN
    . . .               (* эта процедура каким-то образом считывает байт из *)
                        (* некоторого порта ввода *)
  END Get;

  PROCEDURE Put ( b: BYTE );
  BEGIN
    . . .               (* эта процедура каким-то образом записывает байт в *)
                        (* некоторый порт вывода *)
  END Put;

BEGIN
  LOOP
    . . .  (* каким-то образом определяем внешнее событие; *)
           (* оно заносится в event *)
    CASE event OF
      Read : IF ~Buffer.Full() THEN
               Get(b);
               Buffer.Put(b)
             END
    | Write: IF ~Buffer.Empty() THEN
               Buffer.Get(b);
               Put(b)
             END
    | Stop : EXIT
    ELSE
    END;
  END;
END BufTst.


DEFINITION MODULE Buffer;  (* кольцевой буфер *)
  FROM SYSTEM IMPORT BYTE;

  PROCEDURE Empty (): BOOLEAN;      (* буфер пуст? *)
  PROCEDURE Full  (): BOOLEAN;      (* буфер заполнен? *)

  PROCEDURE Put ( b: BYTE );        (* записать элемент в буфер *)
  PROCEDURE Get ( VAR b: BYTE );    (* считать элемент из буфера *)

END Buffer.
```

```
IMPLEMENTATION MODULE  Buffer;
  FROM SYSTEM IMPORT BYTE;

  CONST
    N = 10;    (* макс, количество элементов буфера *)

  VAR
    buf   : ARRAY [O..N-1] OF BYTE;   (* сам буфер *)
    wr    : CARDINAL;     (* номер последней записанной ячейки в буфер *)
    rd    : CARDINAL;     (* номер последней считанной ячейки из буфера *)

  PROCEDURE  Empty (): BOOLEAN;
  BEGIN
    RETURN  (wr = rd)
  END  Empty;

  PROCEDURE  Full  (): BOOLEAN;
  BEGIN
    RETURN  ((wr + 1) MOD N) = rd)
  END  Full;

  PROCEDURE  Put  ( b: BYTE );
  BEGIN
    IF  ~Full()  THEN
      wr := (wr + 1) MOD N;
      buf[wr] := b
    END;
  END  Put;

  PROCEDURE  Get  ( VAR b: BYTE );
  BEGIN
    IF  ~Empty()  THEN
      rd := (rd + 1) MOD N;
      b  := buf[rd]
    END;
  END  Get;

BEGIN  (* раздел инициализации модуля *)
  wr  :=  N-1;   (* считаем, что сначала буфер пуст и что последней считанной *)
  rd  :=  N-1;   (* и записанной ячейкой является ячейка с номером N-1  *)
END Buffer.
```

В примере используется кольцевой буфер на 10 байтов. Для реализации самого буфера используется библиотечный модуль Buffer. За счет простейшей операции — взятия остатка от целочисленного деления (MOD) — достигается эффект кольца. Кольцевой буфер — весьма известный на практике прием, но, обратите внимание на то, насколько четко и лаконично он выражается на языке Modula-2.

Все то, что фигурирует в интерфейсной части модуля безо всякого дополнительного импорта доступно в его исполнительной части (IMPLEMENTATION). Исполнительная часть нужна для того, чтобы можно было доопределить те объекты, которые фигурируют в интерфейсной. Для этого Вам могут потребоваться новые константы, типы, переменные, процедуры — все они являются приватной частью модуля и снаружи недоступны. Библиотечный модуль в своей исполнительной части может иметь так называемый *раздел инициализации,* где обычно требуется устанавливать начальные значения внутренних переменных модуля. Этот раздел можно рассматривать как некую специальную процедуру без параметров, которая вызывается всего один раз, при запуске программы, да и то неявно.

Если теперь Вам понятна идея модуля, можно переходить к вопросам, которые из нее непосредственно вытекают. Это — раздельная компиляция, проблема инициализации модулей, интерфейс с другими языками, smart-компоновка.

Понятие *раздельной компиляции* (separate compilation), возможно, будет для Вас откровением. Гораздо более известной является *независимая компиляция* (independent compilation). Остановимся на этом более подробно. Итак, Вы написали на некотором языке программу, которая состоит из нескольких единиц компиляции. Каждая из этих единиц находится в отдельном файле, и в ней определяются объекты программы. Оттранслировав с помощью компилятора такой файл, Вы получаете объектный код. Все объектные файлы, необходимые для программы, затем собираются с помощью компоновщика в исполняемую программу. Важно здесь то, что когда компилятор обрабатывает очередную единицу компиляции, то он ничего не знает о ее связях с другими и на основании этого генерирует код, оставляя неопределенными ссылки на внешние объекты (на процедуры и переменные), которые доопределяются на этапе компоновки. При раздельной компиляции, поддерживаемой в языке Modula-2, компилятор имеет доступ к спецификации всех внешних объектов единицы компиляции; раздельно компилируются интерфейс модуля и его реализация. При этом осуществляется полный статический контроль типов и параметров процедур, а также проводится контроль областей видимости. Вообще говоря, при компиляции исполнительных модулей получаются объектные файлы, а для интерфейсных модулей — так называемые SYM-файлы. Это поддерживается в первоначальной реализации Вирта и его коллег — компиляторе Modula-2 в операционной системе RT-11 для компьютеров типа PDP-11, а также в Logitech Modula-2. В TopSpeed-реализации это не так. Разработчики среды упростили себе задачу, и компилятор каждый раз транслирует все необходимые интерфейсные модули.

Недостатки независимой компиляции в том, что ошибки в названиях внешних идентификаторов ив интерфейсах внешних процедур могут быть обнаружены лишь на этапе компоновки. Но это еще не самое страшное. Гораздо серьезнее ситуация, когда над программным проектом работает несколько человек и при использовании общих своих библиотек ввиду их постоянной модернизации происходит рассинхронизация. Безусловно, многие возникающие проблемы можно решить за счет создания изощренных внешних утилит, примерами которых могут служить так называемые *make*-системы. Однако, в ряде случаев и они бывают бессильны. Если для обнаружения рассогласования используется только дата модификации файла, т.е. внешний атрибут, который может быть легко изменен безо всякого умысла, а не специальный уникальный ключ, заносимый компилятором в объектный и SYM-файл, то проблема не исчезнет, а лишь будет ловко замаскирована. Компоновщик "подлога" может не заметить.

Еще одно интересное следствие концепции модуля — *проблема инициализации* библиотечных модулей. Как уже говорилось, до выполнения первого оператора основного модуля должна быть проведена инициализация всех прямо и косвенно импортируемых модулей. В какой же последовательности ее осуществлять? Это — чисто математическая задача, носящая название топологической сортировки. Для решения ее нужно проанализировать зависимость модулей друг от друга. При этом возможен взаимный импорт. Затем нужно построить цепочку инициализации. Это не простая проблема: в TopSpeed-реализации она решается довольно сложно. В Logitech это сделано намного элегантнее — в тело инициализации компилятор подставляет вызовы процедур инициализации всех импортируемых модулей, и, кроме того, гарантирует, что инициализация конкретного модуля производится ровно один раз. Но в любом случае все это делает сама система поддержки (RTS), а потому повода для беспокойства у Вас быть не должно.

Весьма актуальным вопросом является *интерфейс с другими языками.* Благодаря возможности разделять модуль на интерфейс и реализацию можно просто написать интерфейс к инородному объектному файлу, где отразить те внешние объекты, которые Вам нужны. Это создает важную основу для интеграции. Но проблемы на этом не кончаются. Есть еще множество подводных камней в виде моделей памяти, способов передачи параметров, соглашений о внешних именах и т.п. К сожалению, все это выходит за рамки данного курса.

Концепция модуля породила возможность компоновать в исполняемый файл программы только те объекты, которые действительно в ней используются. Этот процесс "интеллектуальной компоновки" получил название *smart-компоновка.* Любой программист может по достоинству оценить эту возможность, если ему когда-нибудь приходилось сталкиваться с проблемой компоновки своей программы с большими библиотеками. Предпосылки для реализации smart-

компоновки предоставляет сам язык за счет четкого разграничения областей видимости. Правда, надо заметить, что практически все промышленные компиляторы Modula-2, за исключением TopSpeed, не реализуют эту возможность. А зря.

6.3. Абстрактные типы данных

С областью видимости тесно связана еще одна идея. Это так называемая *инкапсуляция данных* (encapsulation). Инкапсуляция (сокрытие) является неотъемлемой частью теории *абстрактных типов данных.* В соответствии с этой теорией каждый абстрактный тип рассматривается как некое множество величин, с которым связан конкретный набор операций. Ни структура величин, ни внутренность операций извне типа недоступны.

Трудно переоценить те возможности, которые открываются благодаря этой идее. Вы можете строить для себя необходимые абстрактные типы и, единожды отладив их, больше не возвращаться к уже решенной задаче, а стать пользователем собственного модуля, рассматривая его просто как черный ящик. Перед Вами — пример спецификации стека.

```
DEFINITION MODULE  Stack;

  TYPE T;

  PROCEDURE  New (): T;
  PROCEDURE  Del ( VAR stack: T );

  PROCEDURE  Empty ( stack: T ): BOOLEAN;
  PROCEDURE  Push  ( stack: T;     ch: CHAR );
  PROCEDURE  Pop   ( stack: T; VAR ch: CHAR );

END  Stack.


IMPLEMENTATION MODULE  Stack;
  FROM  SYSTEM  IMPORT
    (* PROC *)    TSIZE;
  IMPORT  Storage;

  CONST
    N = 1024;
  TYPE
    Rec = RECORD
            area : ARRAY [1..N] OF CHAR;   (* область под стек *)
            i    : CARDINAL;   (* индекс верхнего элемента стека *)
          END;
    T   = POINTER TO Rec;

  PROCEDURE  New (): T;
    VAR  ptr: T;
  BEGIN
    Storage.ALLOCATE(ptr, TSIZE(Rec)) ;
    IF  (ptr # NIL)  THEN
      ptr^.i := 0
    END;
    RETURN ptr
  END  New;

  PROCEDURE  Del ( VAR stack: T );
  BEGIN
    Storage.DEALLOCATE(stack)
  END Del;
```

```
  PROCEDURE  Empty ( stack: T ): BOOLEAN;
  BEGIN
    IF  (stack # NIL)  THEN  RETURN (stack^.i = 0)
                       ELSE  RETURN TRUE   (* не обрабатываем ошибку *)
    END
  END  Empty;

  PROCEDURE  Push ( stack: T; ch: CHAR );
  BEGIN
    IF  (stack # NIL) & (stack^.i < N) THEN
      WITH  stack^  DO
        INC(i);
        area[i]  := ch
      END
    END
  END  Push;

  PROCEDURE  Pop ( VAR ch: CHAR );
  BEGIN
    IF  (stack # NIL) & (stack^.i > 0) THEN
      WITH  stack^  DO
        ch  := area[i];
        DEC(i)
      END
    END
  END  Pop;
END  Stack.
```

Здесь используется библиотечный модуль Storage, который в принципе должен быть в любой реализации Modula-2. Он отвечает за управление динамической памятью. Обратите внимание, как в примере задается абстрактный тип T. В интерфейсной части он не описан, а доопределяется в исполнительной. Такое описание типа в языке носит название *скрытый тип* (opaque type). Важно знать, какими типами можно доопределять скрытый тип. В роли скрытых типов могут выступать только указатели. Вообще-то говоря, это не всегда удобно. Иногда требуется использовать некоторые скалярные типы, например, CARDINAL.

К сожалению, в языке нет возможности иной работы с абстрактным типом данных, как только через процедуры. Нет возможности превращать эти процедуры в настоящие операторы с инфиксной, префиксной, постфиксной записью, с использованием специальных символов, с заданием приоритета этих операций и с механизмом совмещения имен операций разных абстрактных типов (так называемый полиморфизм).

7. Низкоуровневые средства

Низкоуровневые средства языка позволяют оперировать:

- битами (тип BITSET);
- байтами и словами (типы BYTE и WORD);
- адресами (тип ADDRESS);
- машинными регистрами (псевдо-модуль SYSTEM);
- переменными с фиксированными адресами.

Обработка прерываний в языке возлагается на механизм квазипараллельных процессов, в частности, на процедуру IOTRANSFER из псевдомодуля SYSTEM. Однако, часто бывает удобно работать со специальными процедурами, написанными на ассемблере, которые позволяют осуществлять доступ к средствам используемой операционной системы через системные вызовы.

Для работы с внешними устройствами исходя из идеи отображаемого в памяти оборудования (memory-mapped I/O) язык дает возможность работать с фиксированными ячейками памяти, рассматривая их как обычные переменные. Зная, за что отвечает конкретный адрес в Вашем компьютере, Вы просто при описании переменной указываете абсолютный адрес. Рассмотрим пример, специфичный для PC-компьютеров и библиотеки TopSpeed (модуль ввода/вывода IO).

```
MODULE  MemTst;
  IMPORT  IO;
  VAR
    mem[OOOOH:0413H]: CARDINAL;
BEGIN
  IO.WrStr ("Total memory: "); IO.WrCard(mem,1); IO.WrStr ('K'); IO.WrLn
END MemTst.
```

Этот пример позволяет узнать размер базовой (conventional) памяти в килобайтах, который хранится в ячейке по адресу OOOOH:0413H.

В языке в явном виде не допускается использовать ассемблерные вставки, которые подчас нужны для повышения эффективности. Однако, TopSpeed вместо этого предлагает замену — так называемые *inline*-процедуры. Идея их проста — Вы пишете какую-то часть на ассемблере, а затем полученный код как данные подставляете к описанию интерфейса inline-процедуры в виде массива констант. Вот как выглядит в TopSpeed Modula-2 чтение байта из порта ввода.

```
TYPE  T2 = ARRAY [0..1] OF BYTE;
INLINE PROCEDURE InByte ( port: CARDINAL ): BYTE = T2 (OOECH,OOCBH);
```

Ясно, что таким средством нужно пользоваться крайне аккуратно. Другими низкоуровневыми вещами считаются функции структурного преобразования типов (мы их уже подробно рассматривали) и такие процедуры из модуля SYSTEM, как ADR, SIZE, TSIZE, NEWPROCESS, TRANSFER, IOTRANSFER.

8. Квазипараллельное программирование

В этом кратком курсе нет возможности подробно рассмотреть столь интересный раздел языка, которому в свое время Вирт уделял наиболее пристальное внимание. Но обойти его вниманием было бы не совсем корректно. В язык введен механизм *сопрограмм* (coroutine), или процессов. Сопрограмму можно рассматривать как процедуру без параметров, работающую со своим независимым стеком. Получив управление, сопрограмма может явно его передать другому процессу с помощью специальной процедуры. Таким образом, на однопроцессорной архитектуре происходит имитация параллельной работы за счет частого переключения процессов. Кроме того, внешние устройства работают параллельно с центральным процессором, и совокупность всех процессов вместе с драйверами внешних устройств можно рассматривать как систему взаимодействующих параллельных процессов.

В языке предусмотрены три специальные процедуры, формирующие базис для *квазипараллельного программирования* (quasi-concurrent programming).

```
NEWPROCESS   ( proc: PROC; stack: ADDRESS; size: CARDINAL; VAR ptr: ADDRESS );
TRANSFER     ( VAR from, to: ADDRESS );
IOTRANSFER   ( VAR from, to: ADDRESS; intr: CARDINAL );
```

NEWPROCESS используется для порождения нового процесса на базе процедуры proc. Вы должны указать процедуре адрес и размер требуемого стека, а она Вам вернет указатель на дескриптор порожденного процесса. Это просто загрузка процесса, управление ему не передается!

TRANSFER позволяет явно передать управление от текущего процесса другому процессу с дескриптором to. При этом его состояние на момент приостановки запоминается в дескрипторе по адресу from.

IOTRANSFER является аналогом TRANSFER для процессов-драйверов внешних устройств. Каждый драйвер обычно захватывает определенный вектор прерывания (intr). Захват вектора и описывается этой процедурой. Другими словами, Вы породили некоторый процесс, который хотите сделать драйвером. В теле его процедуры ставится IOTRANSFER, которая при своем выполнении связывает текущий процесс (from) с конкретным вектором прерывания. При этом осуществляется передача управления другому процессу. Теперь, когда произойдет прерывание по этому вектору, управление будет передано нашему процессу-обработчику, а после обработки оно вновь возвращается процессу to.

Безусловно, это — лишь базовые средства. На практике используют соответствующий диспетчер процессов, выполняемый в виде библиотечного модуля. Диспетчер обеспечивает жесткое централизованное управление и предоставляет для работы Вам более высокоуровневые средства взаимодействия процессов (сигналы, рандеву и т.п.).

В TopSpeed Modula-2 тоже имеется подобный диспетчер, который называется Process. Существенным недостатком его реализации помимо скорости переключения контекстов является жесткое ограничение на количество одновременно запущенных процессов (всего 32).

9. Встроенные процедуры

Все встроенные процедуры языка можно условно разбить на три группы: общие процедуры, процедуры преобразования типов и процедуры из модуля SYSTEM (низкоуровневые).

*** Общие процедуры**

INCL(s,i)	добавить элемент во множество	s := s + {i}
EXCL(s,i)	исключить элемент из множества	s := s - {i}
INC(x)	простое инкрементирование	x := x + 1
INC(x,n)	инкрементирование	x := x + n
DEC(x)	простое декрементирование	x := x - 1
DEC(x,n)	декрементирование	x := x - n
ODD(x)	является ли нечетным число	(x.MOD.2) # 0
HIGH (a)	старший индекс открытого массива	
ABS (x)	взятие модуля числа (INTEGER, LONGINT, REAL, LONGREAL)	
CAP (ch)	преобразование буквы в заглавную	

*** Процедуры преобразования типов**

ORD(x)	преобразование типов CHAR и перечисления в CARDINAL
CHR(x)	преобразование CARDINAL в CHAR
FLOAT (x)	преобразование CARDINAL/INTEGER в REAL
TRUNC(x)	преобразование REAL в INTEGER
VAL (type, x)	универсальное структурное преобразование типов

*** Процедуры из псевдо-модуля SYSTEM**

SIZE (x)	размер переменной (в байтах)
TSIZE(type)	размер типа (в байтах)
ADR(x)	адрес переменной (ADDRESS)
NEWPROCESS(proc, adr, size, ptr)	породить процесс
TRANSFER.(from, to)	передать управление другому процессу
IOTRANSFER (from, to, intr)	обработать прерывание

Помимо этих процедур TopSpeed предоставляет следующие:

Ofs (x)	смещение для адреса переменной x (CARDINAL)
Seg(x)	сегмент для адреса переменной x (CARDINAL)
FieldOfs (rec.field)	смещение поля в записи

MAX(type)	максимальное значение типа type
MIN(type)	минимальное значение типа type

В языке имеются также две процедуры, связанные с рапределением динамической памяти - это NEW (запросить память для переменной) и DISPOSE (освободить выделенную память). Они были унаследованы языком Modula-2 от своего предшественника — языка Pascal. Использовать их крайне неудобно из-за того, что в текущей области видимости должны быть доступны соответственно процедуры ALLOCATE и DEALLOCATE из модуля Storage (через которые и реализуются операторы NEW и DISPOSE). Похоже, такое замысловатое решение было выбрано Виртом умышленно, дабы постепенно превратить их в пережиток прошлого.

10. Комментарии и прагмы компилятора

Комментарии выделяются в Modula-2 парами (* и *) . Они могут быть и вложенными. В большинстве компиляторов Modula-2 комментарии используются также для управления процессом компиляции. Каждая реализация языка делает это по-своему. Управление компилятором выполняется с помощью специального языка прагм.

TopSpeed Modula-2 обладает одним из самых изощренных языков прагм. Прагмы начинаются с комбинации (*#, затем идет имя прагмы и значения ее параметров. Рассмотрим наиболее важные из прагм (подробнее с ними можно ознакомиться по технической документации).

Основные прагмы TopSpeed можно разделить на следующие категории (здесь приводится точный синтаксис и значения по умолчанию).

optimize *оптимизация кодогенерации*

```
(*# optimize (const => on)   *)   константы — в регистрах (on|off)
(*# optimize (copro => emu)  *)   сопроцессор  (emu|87|287|387)
(*# optimize (cpu   => 86)   *)   процессор    (86|286|386|486)
(*# optimize (speed => on)   *)   скорость/объем кода  (on|off)
```

check *генерация проверочного кода*

```
(*# check (index    => off) *)   нарушение индекса              (on|off)
(*# check (nil_ptr  => off) *)   разыменование NIL              (on|off)
(*# check (overflow => off) *)   арифметическое переполнение    (on|off)
(*# check (range    => off) *)   выход за границы диапазона    (on|off)
(*# check (stack    => off) *)   переполнение стека            (on|off)
```

module *настройка модуля*

```
(*# module (implementation => on)  *)  наличие реализации модуля   (on|off)
(*# module (init_code      => on)  *)  наличие части инициализации (on|off)
(*# module (smart_link     => on)  *)  smart-компоновка            (on|off)
```

name *формирование внешних имен*

```
(*# name (prefix => modula) *)  стратегия (none|c|modula|os2_lib|windows)
```

call *вызовы процедур*

```
(*# call (near_call => off) *)   ближний вызов (16 бит)          (on|off)
(*# call (c_conv    => off) *)   соглашение языка C             (on|off)
(*# call (inline    => off) *)   копировать функцию в код без call (on|off)
(*# call (interrupt => off) *)   IRET вместо RET                 (on|off)
(*# call (o_a_copy  => on)  *)   открытый массив копировать в стек (on|off)
(*# call (set_jmp   => off) *)   нелокальные JMP                 (on|off)
(*# call (var_arg   => off) *)   переменное число параметров     (on|off)
(*# call (windows   => off) *)   код для Windows                 (on|off)
(*# call (result_optional => off) *)  функция как процедура      (on|off)
(*# call (seg_name  => <name>) *)     задать имя сегмента кода <name>_TEXT
(*# call (reg_param => ax, bx, ex, dx, st0, st6, st5, st4 , st3 ) *)  список регистров
(*# call (reg_return => ax,dx,st0)*)  список регистров для возврата результата
(*# call (reg_saved  => si , di , ds, st1, st2 ) *)   список сохраняемых регистров
```

* **data** *области данных*

```
(*# data (near_ptr     => off)      *) ближний указатель (16 бит)          (on|off)
(*# data (volatile     => off)      *) данные не держать в регистрах       (on|off)
(*# data (far_ext      => off)      *) различ.сегменты под внеш.имена      (on|off)
(*# data (var_enum_size => on)     *) под перечисление 2 байта            (on|off)
(*# data (seg_name     => <name>)   *) задать имя сегмента данных <name>_DATA
(*# data (stack_size   => 06000H)  *) размер стека в байтах (16K)
(*# data (heap_size    => OFFFFH)   *) сколько под near-кучу
```

* **save, restore** *сохранение/восстановление прагм*

```
(*# save    *)   сохранить состояние прагм
(*# restore *)   восстановить состояние прагм
```

11. Рекомендуемая литература

Из тех книг, которые вышли у нас в стране, можно порекомендовать следующие.

1. Н.Вирт *Программирование на языке Модула-2.* М., Мир, 1987.
2. Н.Вирт *Алгоритмы и структуры данных.* М., Мир, 1989.
3. Э.Нэпли, Р.Платт *Программирование на языке Модула-2.* М., Радио и связь, 1989.

Помимо них определенный интерес могут представлять следующие книги. Выделены те из них, что уже переведены и изданы у нас. Большая часть этих книг доступна через фонды ГПНТБ.

1. **Wirth N. *Programming In Modula-2.* Springer, 1985.**
2. Cleaves R. *Modula-2 for Pascal Programmers.* Springer, 1984.
3. **Knepley Ed, Platt R. *Modula-2 Programming.* Reston Publishing Company, 1985.**
4. Greenfield S. *Invitation to Modula-2.* Petrocelli Books, 1985.
5. WiatrowskiC. *From C to Modula-2... and Back Bridging the Language Gap.* John Wiley &Sons, 1987.
6. **Christian K. *A Guide to Modula-2.* Springer, 1986.**
7. Ogilvie J. *Modula-2 Programming.* McGraw-Hill, 1985.
8. Thallmann D. *Modula-2: An Introduction.* Springer, 1985.
9. Terry P. *An Introduction to Programming with Modula-2.* Addison-Wesley, 1987.
10. **Riley D. *Using Modula-2.* Boyd & Fraser Publishing, 1987.**
11. Wiener R., Sincovec R. *Software Engineering with Modula-2 and Ada.* John Wiley & Sons, 1984.
12. Pomberger G. *Software Engineering and Modula-2.* Prentice-Hall, 1984.
13. Ford G., Wiener R. *Modula-2: A Software Development Approach.* John Wiley & Sons, 1985.
14. Sale A. *Modula-2: Discipline & Design.* Addison-Wesley, 1986.
15. Sincovec R., Wiener R. *Data Structures with Modula-2.* John Wiley & Sons, 1986.
16. Messer P. *-Modula-2: Constructive Program Development.* Blackwell Scientific Publ., 1986.
17. Ward T. *Advanced Programming Techniques in Modula-2.* London, 1987.
18. Walker R. *Modula-2 Library Modules. A Programmer's Reference.* TAB Books Inc., 1988.
19. Lins C. *The Modula-2 Software Component Library.* Springer, 1989.

В настоящее время на полках многих книжных магазинов можно увидеть книгу Д.Райли "Использование языка Modula-2". Конечно, тот пространный стиль, который избрал автор, у многих может вызвать раздражение. Однако, в ней есть, что почитать. Книга изобилует примерами и много внимания в ней уделяется не только различным приемам программирования, но также и всему процессу разработки программ. Остается добавить, что книга написана на базе университетского курса лекций, который был прочитан автором книги в университете штата Висконсин.

Р. Богатырев