Введение

В мире программирования Pascal уже утвердился как язык программирования. Созданный специально с педагогическими целями, он распространился настолько, что поставил свой запрос на профессиональный язык, каким он является для мини- и микрокомпьютеров. Большая часть программного обеспечения составляется на языке Pascal, а в научной литературе он стал эталоном. С ним сопоставляются новые языки программирования, записываются алгоритмы на псевдо-Pascal-е и т.д. Достаточно упомянуть только, что такие языки как Modula-2 и Ada являются его наследниками и продолжителями. Но он обладает и недостатками. Первым его недостатком является отсутствие возможности осуществления раздельно-модульной компиляции. Вторым его недостатком является отсутствие средств для осуществления связи с файловой системой данного компьютера. Другим его недостатком является позднее появление международного стандарта, что привело к тому, что каждая его реализация в действительности решает проблемы по своему способу, чаще всего отличающемуся от других. Появление стандарта не изменило существенно ситуацию.

Созданием проекта языка UniPascal были преследованы следующие цели:

- близость к широко распространенной для персональных микрокомпьютеров IBM PC версии Turbo Pascal-я;
- согласование с ограниченными возможностями микрокомпьютеров фамилии Пылдин.

В результате получился язык, для которого можно утверждать, что хотя он и не обеспечивает полной переносимости программного обеспечения с UniPascal-я на Turbo Pascal, то создает все предпосылки достижения переносимости с небольшими затратами, поддерживая тот же самый метод условной компиляции и стандартные типы INTEGER, LONGINT, STRING и другие.

Расширенные формы Бэкуса-Наура

Повсюду в этом документе, где в описании языка UniPascal используется имя Pascal, речь идет о возможностях, обладаемых языками Pascal и UniPascal. Там, где используется имя UniPascal, речь идет о возможностях, которыми обладает только UniPascal или которые дополнены по сравнению с теми же в языке Pascal.

С формальной точки зрения язык представляет собой множество предложений. Каждое предложение состоит из слов. Слова образуют словарь языка. Словарь каждого языка представляет собой конечное множество. Но предложения, которые можно составить на этом языке, бесконечно много. Следовательно, слова можно описать простым перечислением, но для предложений этого нельзя применить. Для их описания используются правила, при помощи которых можно создать все предложения языка.

Для описания синтаксиса языка UniPascal будет использован модифицированный вариант форм Бэкуса-Наура (БНФ), называемый Расширенными Формами Бэкуса-Наура (РБНФ). Ниже следует краткое описание РБНФ.

Описание синтаксиса языка программирования с помощью РБНФ состоит из набора правил, еще называемых "продукциями". Каждое правило состоит из нетерминального символа и РБНФ-выражения, разделенных знаком равенства. Правило завершается точкой. Нетерминальный символ представляет собой "метаимя", которое определяется РБНФ-выражением.

РБНФ-выражение может быть пустым или содержать множество терминальных символов, нетерминальные символы и следующие метасимволы:

Метасимвол	Значение
=	Равно по определению
|	Или (альтернатива)
.	Конец правила
{A}	0 или более вхождений A в правило
[A]	0 или 1 вхождение A в правило
(A|B)	Группирование: или A, или B
"**ABC**" или '**ABC**'	Терминальный символ ABC
метаимя	Нетерминальный символ с именем "метаимя".

Метасимвол = служит для разделения правой от левой части синтаксического правила. Альтернативы разделяются метасимволом **|**.

Нетерминальные символы представляют собой набор из букв, цифр и символа подчеркивания '_'. Терминальные символы в РБНФ заключаются в " или '. В конце каждого правила ставится точка (.). Через **{ A }** обозначается, что **A** может иметь ноль или более вхождений в правило. Через **[A]** обозначается, что **A** входит ноль или один раз. Использование круглых скобок - традиционное для математических выражений - показано на примере: выражение S = A X B | A Y B можно записать : S = A (X | Y) B.

В качестве примера использования РБНФ опишем ими их собственный синтаксис:

```
Production=     NonTerminal '=' Expression '.'.

Expression=     Term { '|' Term }.

Term=           Factor { Factor }.

Factor=         NonTerminal     |
                Terminal        |
```

```
                    '(' Expression ')' |
                    '[' Expression ']' |
                    '{' Expression '}' .

Terminal=           "'" ASCII_no_SP { ASCII_no_SP } "'" |
                    '"' ASCII_no_SP { ASCII_no_SP } '"' .

NonTerminal=        Letter { Letter | Digit }.

Letter=             'A' | ≈ 'z' | '_'.

Digit=              '0' | '1' | ≈ '9'.

ASCII_no_SP=        Letter | Digit |
                    '!'| '"'| '#'| '$'| '%'| '&'| "'"| '('|
                    ')'| '*'| '+'| ','| '-'| '.'| '/'| ':'|
                    ';'| '<'| '='| '>'| '?'| '@'| '['| '\'|
                    ']'| '^'| '`'| '{'| '|'| '}'| '~'.
```

Многоточием обозначен пропуск символов в тексте описания. Полное формальное описание цифры (Digit) и буквы (Letter) дается позже.

1. Основные понятия языка UniPascal

1.1. Основные символы языка UniPascal

Основные символы языка подразделяются на: буквы (Letters), цифры (Digits) и специальные символы.

Используется следующее подмножество символьного набора ASCII.

- прописные и строчные буквы латинского алфавита **A-Z, a-z**
- цифрры с 0 до 9 **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
- специальные символы: **пробел " # $ & ' ( ) * + , - . / : ; < = > [ ] ^ _ { | }**

Разрешается использование и других (кроме описанных) символов, но только в символьных строках или в комментариях.

1.2. Лексемы и разделители

Программа на Pascal-е состоит из лексем и символов - разделителей. Лексемы строятся из перечисленных выше символов и разделяются на следующие группы:

- специальные символы (+ - := <= ..);
- идентификаторы и зарезервированные слова (begin end counter myproc);
- числовые константы (1 $ff 256 3.1415926e+00 2.56e2);
- строки-константы ('UniPascal' 'UniDOS').

Разделителями между лексемами являются: пробел, комментарий, специальные символы и конец строки.

Правила РБНФ для букв и цифр:

Digit= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.

Letter= '_'| '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'.

Как можно увидеть, символ подчеркивания '_' считается буквой.

1.3. Идентификаторы

Идентификаторы служат для обозначения констант, меток, типов, переменных, полей записи, процедур, функций, модулей и программ.

Ident= Letter { Letter | Digit }.

IdentList= Ident { ',' Ident }.

Идентификатор - это комбинация произвольного количества букв и цифр, начинающаяся с буквы. Допустимы только буквы латынского алфавита. Существуют следующие исключения приведенного выше правила:

- запрещается использовать в качестве идентификаторов зарезервированные слова;
- идентификаторы, отличающиеся только использованием прописных или строчных букв, считаются эквивалентными;
- на общую длину идентификаторов не накладываются ограничения, но считается, что значащими являются только первые 8 символов. Идентификаторы, не отличающиеся первыми восмью символами, считаются одинаковыми;
- использованный в идентификаторе символ подчеркивания '_' считается буквой.

Пример правильных идентификаторов:

UniPascal uni_pascal counter x2 next

Пример неправильных идентификаторов:

max-value	содержит знак минуса
2_pi	начинается цифрой
счетчик	нельзя использовать кириллицы

В некоторых случаях (при использовании модулей) необходимо уточнить один идентификатор при помощи другого идентификатора (имени модуля). Такие идентификаторы называются квалифицированными (qualified). Уточнение (квалификация) происходит записыванием сначала квалифицирующего идентификатора (наименования модуля), за ним точки, а потом - квалифицируемого идентификатора. Например, чтобы использовать идентификатор MYPROC, описанный в модуле MYUNIT, можно записать следующее MYUNIT.MYPROC.

Синтаксис:

Quallident= [Ident '.'] Ident.

1.4. Зарезервированные слова и специальные символы

Следующие идентификаторы в языке UniPascal являются зарезервированными и используются с одним или несколькими значениями. С жирным шрифтом даются дополнительно зарезервированные по отношению языка Pascal (только в UniPascal-е) слова:

and	end	label	record	**uses**
array	file	mod	repeat	var
begin	for	nil	set	while
case	function	not	**segment**	with
const	goto	of	then	**xor**
div	if	or	to	
do	in	packed	type	
downto	**interface**	procedure	until	
else	**implementation**	program	**unit**	

Следующие специальные символы и пары специальных символов тоже используются с одним или несколькими значениями:

+ - * / = <> [ ] . , ( ) : ; ^ { } $ # <= >= <> := (* *)

1.5. Числовые константы

При записи целых и вещественных констант в UniPascal-е используется обычная запись в десятичной системе счисления, при том для вещественных чисел используется десятичная точка.

В отличие от стандартного Pascal-я, целые константы могут быть представлены в шестнадцатеричном виде, при том символ '**$**' должен предшествовать числу.

Букву "E" в вещественных константах следует интерпретировать как "умножить на 10 в степени".

Целые константы должны быть в интервале [-2147483648, 2147483647] или, если они шестнадцатеричные, в интервале [$0, $FFFFFFFF].

В отличие от стандартного Pascal-я, допускается группировать цифры одного числа знаком подчеркивания '_'. Это предусмотрено для удобного чтения длинных чисел. Например, для компилятора числа 1000000000 и 1_000_000_000 эквивалентны, но человек гораздо легче воспринимает вторую запись.

Синтаксис:

HexDigit= Digit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'.

Decimal= Digit { Digit | '_' }.

IntConst= Decimal | ('$' HexDigit { HexDigit | '_' }).

Sign= ['+' | '-'].

ScaleFactor= ('E' | 'e') Sign Decimal.

RealConstant= Decimal (('.' Decimal [ScaleFactor]) | (['.' Decimal] ScaleFactor)).

SignedRealConst= Sign RealConstant.

SignedIntConst= Sign IntConst.

Примеры:
Целые константы: 0, 1, 259, $0, $f, $7fff, $EF5
Вещественные константы: 0.0, 1.0, 259.0, 0.0005, 1.0E-23.

1.6. Символьные константы

Символьная константа - это последовательность нескольких (ноль или более) символов (букв) расширенного ASCII набора символов, записанная в одной последовательности, заключенная в кавычки (") или апострофы (') (одиночные кавычки).

Символьная константа без символов - это пустая строка. При обозначении начала и конца символьной константы должны быть использованы одинаковые символы.

Примеры: 'пример 1', "пример 2", 'ошибка".

Если необходимо записать кавычки внутри символьной константы, то нужно повторить их дважды.

Пример:
'Эта строка содержит апостроф: ''.'

Другая запись той-же символьной константы:

"Эта строка содержит апостроф: '."

Во втором случае не нужно писать апостроф дважды, потому что для обозначения начала и конца строки используются кавычки.

Есть особый вид символьной константы - односимвольная. Ее можно записать и так: **#n**, где **n** - ASCII код символа.

Последовательность строк, разделенных пробелом или символом конца строки, является одной строкой. Это дает возможность записать символьную константу в нескольких строках текста. Символьная константа не должна превышать 255 символов.

В символьной константе могут быть записаны все символы набора ASCII и все буквы кириллицы.

Следует общий вид символьных констант:

CharConst= '"' ASCII_8 '"'| "'" ASCII_8 "'"| '#'IntConst.

StringConst= { '"' { ASCII_8 } '"' |
"'" { ASCII_8 } "'" |
CharConst } |
'""' | "''".

ASCII_8= Digit | Letter |
'!'| '"'| '#'| '$'| '%'| '&'| "'"| '('| ')'| '*'| '+'| ','| '-'| '.'|
'/'| ':'| ';'| '<'| '='| '>'| '?'| '@'| '['| '\'| ']'| '^'| '_'| '`'|
'{'| '|'| '}'| '~'...

Здесь ASCII_8 означает символ 8-битового ASCII набора, включающий и буквы кириллицы, а IntConst - это ASCII код символа (целое число без знака в интервале [0..255]).

Примеры:

1) 'Это строка'
2) 'Эта строка содержит '''
3) "Эта строка содержит ' "
4) 'Эта строка содержит " '
5) "Это"
'длинная '
"строка"

1.7. Комментарий

Комментарий - это последовательность символов, заключенная в **{ }** или **(* *)**. Он может появляться везде в программе, где допускаются разделители. Комментарий предназначен для пояснений в тексте программы и при трансляции игнорируется. Он не влияет на выполнение программы. В программе на языке Pascal легко можно использовать комментарии. Рекомендуется возможно чаще использовать эту возможность.

При составлении комментария необходимо соблюдать следующие правила:

- комментарий должен быть заключен в комментарные скобки одного типа, т.е. нельзя начать комментарии с фигурной скобки, а завершить его с ***)**, или наоборот;

- если комментарий заключен в фигурные скобки **{ }**, то в нем нельзя использовать правую фигурную скобку **}**. Например: { Это } неправильный комментарий };
- если комментарий заключен в **(* *)**, то в нем нельзя использовать комбинацию ***)**. Например: (* И это *) неправильный комментарий *);
- не допускаются вложенные комментарии одного типа. Например: { Неправильный { комментарий }! };
- если комментарий начинается с символа **$**, то это специальный комментарий и он предназначен для управления работой компилятора. Например: (*$I+*);
- так как программисту на языке UniPascal разрешено записывать текст своей программы в нескольких файлах, комментарий должен быть записан полностью в одном файле.

Синтаксис:

```
Comment=        ('{' { ASCII_8 } '}') |
                ('(*' { ASCII_8 } '*)').
```

Примеры правильных комментариев:

```
{ Это правильный комментарий }
(* А это другой правильный комментарий *)
{ Символы *) можно записать и они входят в комментарий }
(* Здесь можно использовать фигурные скобки { и }, потому что комментарий
заключен в другие комментарные скобки *)
{   $ этот комментарий начинается с пробела, а не с $ }
```

Примеры неправильных комментариев:

```
{ не должно быть { вложенных } комментариев }
{ комментарные скобки должны быть одного типа *)
{$25.03 начинается с $ и считается директивой компилятора }
```

2. Общая структура программы

Каждая программа на языке UniPascal состоит из заголовка (Program Heading), списка используемых в программе внешних модулей (Uses Clause) и блока (Block).

```
Program=        ProgramHeading
                 UsesClause
                Block '.'.
```

Блок состоит из двух основных частей: раздел описания обрабатываемых данных (Declarations) и раздел определения действий над данными. В разделе описания данных даются все использованные в программе (процедуре или функции) идентификаторы. Они могут быть метками, константами, типами, переменными, процедурами и функциями. В отличие от стандартного языка Pascal в языке UniPascal порядок появления разделов описания не имеет значения. Только описание каждого идентификатора должно предшествовать его использованию в программе.

```
Block=          [ Declarations ]
                'begin'
                   Statement { ';'
                   Statement }
                'end'.

Declarations=   { { LabelDeclaration } |
                  { ConstDeclaration } |
                  { TypeDeclaration }  |
                  { VarDeclaration }   |
                  { PFDeclaration } }  .
```

2.1. Заголовок программы

В заголовке программы указывается имя программы и имена стандартных внешних файлов, с которыми она будет взаимодействовать. Стандартные внешние файлы следующие:

```
ProgramHeading=     'program' Ident [ '(' IdentList ')' ] ';'.
```

Более подробное описание списка идентификаторов, заключенного в скобках после имени программы (список внешних файлов) дается в п. 3.2.5.2.

2.2. Раздел описания меток

Любой оператор в программе можно маркировать, поставив перед ним через двоеточие метку. В отличие от некоторых других языков программирования (например, FORTRAN) каждая метка, прежде чем она будет использована в теле блока, должна быть описана в разделе описания меток. Этот раздел начинается с зарезервированного слова LABEL, за которым следует список используемых меток и точка с запятой. Метка в Pascal-е представляет собой целое число без знака, лежащее в диапазоне от 0 до 9999. В UniPascal-е в качестве метки может выступать идентификатор.

```
Label=              Ident | IntConst.

LabelDeclaration=   'label' Label { ',' Label } ';'.
```

Метка EXIT зарезервирована и ее нельзя описывать в разделе описания меток. Ее использование вызывает специальное действие в теле процедуры, функции, программы или модуля, где она встречается. Для более подробного описания эффекта ее использования смотри п. 12.

Пример описания меток: Label 11, Stop, Continue, 0;

2.3. Раздел описания констант

В разделе описания констант описываются идентификаторы, которые используются в качестве имен констант. Описание константы вводит имя как синоним некоторой константы. Использование имен (иднетификаторов) констант делает программу более 'читаемой' и способствует улучшению ее документируемости. Кроме того, это позволяет программисту сгруппировать в начале программы величины, зависящие от компьютера или являющиеся характерными для данного примера: здесь они более заметны и их легче изменить. Тем самым улучшается переносимость программ и их модульность.

Описание константы состоит из идентификатора (который принимает значение константы), знака равенства и значение константы.

ConstDeclaration= 'const' Ident '=' Constant ';' {
Ident '=' Constant ';' }.

Constant= SignedRealConst | SignedIntConst | CharConst | StringConst | ConstExpression.

ConstExpression= Expression.

В качестве констант используются не только обычные константы, но и выражения (Expression), значения которых можно вычислить во время компиляции. Примеры:

```
const N =      100;
      N^2 =    N * N;
      Pi =     3.141596;
      Name ='UniPascal';
```

Не допускаются рекурсивные описания констант - определяемый идентификатор не должен входить в константную часть (Constant) данного описания.

```
const x = x + x;        или        const x = y;
                                         y = x + 1;
```

2.4. Раздел описания типов

Одной из основных особенностей языка Pascal является сильная типизация. Это означает, что каждый объект связывается с одним и только с одним типом. Тип - это множество значений плюс множество операций, которые можно выполнить над этими значениями. Задавая объекту некоторый тип, программист определяет набор значений, которые можно присвоит этому объекту и набор операций, с помощью которых можно манипулировать с ним. Программист должен описать все объекты, указывая их типы и использовать объекты только в соответствии с их типами. Так можно еще во время компиляции обнаружить ошибки, связанные с некорректным использованием значений объекта или операций над этими значениями.

В языке Pascal не только предусмотрено несколько стандартных типов, но существует и механизм для определения типа, который вводит идентификатор для обозначения некоторого типа. Этот идентификатор можно использовать для определения новых, более сложных, типов данных или для описания переменных.

Синтаксис раздела определения типов:

```
TypeDeclaration=    'type' Ident '=' Type ';' {
                        Ident '=' Type ';' }.

Type=               TypeIdent | SimpleType | PointerType | StructuredType.

TypeIdent=          Ident.
```

При описании нового типа нужно использовать только идентификаторы уже описанных типов (не считая указатели).

2.5. Раздел описания переменных

Каждая переменная, встречающаяся в программе, должна быть описана в разделе описания переменных. Использование неописанной переменной считается ошибкой.

Синтаксис переменных:

```
VarDeclaration=     'var' IdentList ':' Type ';' {
                        IdentList ':' Type ';' }.
```

Где: Ident - идентификатор (имя описываемой переменной).

Описание переменной задает связь между идентификатором и некоторым типом данных, тем самым определяя и операции, которые можно применять над переменной.

2.7. Раздел описания процедур и функций

Каждое имя процедуры или функции должно быть описано до его использования в разделе операторов. Описание процедуры или функции состоит из заголовка и блока. Процедура представляет собою подпрограмму и активизируется через оператор процедуры. Функция - это тоже подпрограмма, но она возвращает по своему окончанию некоторое значение и используется как компонента выражения.

```
PFDeclaration=      { ProcDeclaration | FuncDeclaration }.
```

2.8. Правила доступности и область действия имен и меток

Идентификаторы и метки являются локальными в рамках блока, в начале которого они описаны. Это означает, что вне этого блока они недоступны. Идентификаторы доступны везде в этом блоке и в вложенных в нем блоках. Идентификаторы, описанные в блоке, представляющей собою программу, называются глобальными. Глобальные идентификаторы доступны везде в программе.

Следует учитывать несколько особенностей:

- метки доступны только в блоке, в котором они описаны;

- если некоторая переменная используется как управляющая переменная цикла 'FOR', то она должна быть локальной для блока, в теле которого находится сам цикл;
- если есть вложенные блоки, то каждый идентификатор внешнего блока можно переописать во внутреннем. Во внутреннем блоке имеет силу описание, сделанное в нем и программист не имеет никакой возможности использовать идентификатор в смысле описания, данном во внешнем блоке;
- любой идентификатор или метку можно дефинировать только один раз в рамках данного блока. Исключениями являются случаи переописания в блоке, вложенном в данном блоке (это описано выше) и случаи дефинирования идентификатора как поле структуры записи. Нельзя переописать идентификатор на том же уровне в той-же самой структуре записи;
- описание каждого идентификатора входит в силу с места появления его описания до конца блока, в котором это описание находится. Использование идентификатора до его описания неправильно и считается ошибкой. Исключение от этого правила допускается только при описании указателей. Разрешено описание указателя на еще неописанный тип, но его описание (неописанного типа) должно появиться до конца того же самого раздела описания типов, в котором находится описание указателя.

Каждая программа на языке Pascal считается описанной как локальный блок некоторой обхватывающей его среды. В этой внешней среде (внешний блок) описаны все стандартные идентификаторы. В языке UniPascal ситуация отличается от выше описанной. Программа автоматически использует модуль STANDARD, так що все стандартные идентификаторы считаются описанными в нем. Любой из них может быть переопределен, но это считается признаком дурного стиля. Можно переописать и стандартные типы. Например:

```
type WorkType = integer;          { integer - стандартный тип integer}
     integer =  real;                    { real - стандартный тип real}
     real =     WorkType;          { real описывается как integer}
```

На UniPascal-е это можно сделать еще проще:

```
type integer = Standard.real;
     real = Standard.integer;
```

После этих дефиниций, если описать переменную r так:

```
var r: real;
```

то ей можно присваивать только целые значения. Как сами могли убедиться, Pascal достаточно гибкий язык, но всеми его возможностями надо пользоваться в меру.

3. Типы данных

Одной из основных особенностей языка Pascal (по сравнению с существующими до его появления) является концепция типов данных. По словам автора языка Н. Вирт организация данных в языке Pascal основывается на теории Хоара о структурной организации данных. Согласно ей тип обладает следующими свойствами:

- тип определяет класс значений (к которому могут принадлежать константы или которые могут принимать переменные и выражения) и множество операций над этим классом значений;
- каждое значение принадлежит одному и только одному типу;
- тип константы, переменной или выражения можно определить наличной информацией в представлении операнда или извлечь из контекста: не приходится ждать выполнения программы (это условие статических типов данных);
- каждой операции соответствует некоторый фиксированный тип ее операндов и некоторый фиксированный тип ее результата;
- для каждого типа свойства значений и элементарные операции над ними задаются аксиомами.

При работе с языком высокого уровня, знание сущности типа данных позволяет компилятору обнаруживать в программе бессмысленные конструкции и решать вопрос о методе представления данных и о способе выполнения компьютером преобразований над данными.

Сильная типизация (типы обладают указанными выше свойствами), введенная в языке Pascal, является первым шагом в направление абстрактных типов данных, идея и реализация которых появляются в позже разработанных языках.

И так, основные характеристики типа - это множество значений, принадлежащих этому типу данных, и операции, определенные над объектами этого типа. При описании типов они будут представлены с точки зрения этих двух аспектов.

3.1. Простые типы данных

К простым типам данных в UniPascal-е относятся стандартные и перечисляемые типы и типы диапазона. Вводитя и понятие ординального типа, включающего в себя все простые типы за исключением вещественого.

SimpleType= OrdinalType | RealType.

OrdinalType= Enumerated | SubRange | StandardType.

StandardType= 'integer' | 'shortint' | 'longint' |
'cardinal' | 'shortcard' | 'natural' |
'char' | 'boolean' |
'byte' | 'word' | 'longword'|.

RealType= 'real'.

Для всех простых типов данных дефинированы следующие операции (не будем специально останавливаться на них, но будем предполагать их наличие): присваивание значения (:=) и отношения равенства (=), неравенства ( ), меньше (<), не больше (<=), больше (>) и не менше (>=). При операциях отношений тип результата - логический (BOOLEAN).

3.1.1. Перечисляемый тип

Перечисляемый тип (Enumerated) определяется перечислением всех возможных его значений. Описание состоит из списка констант.

Enumerated= '(' IdentList ')'.

Пример (приведенные определения остаются в силе до конца главы, чтобы не писать их перед каждым примером):

```
type Color =      (Black, Red, Green, Blue, White);
       DayOfWeek = (Sunday, Monday, Tuesday, Wednesday,
                    Thursday, Friday, Saturday);
var CarColor: Color;
      yesterday, today, tomorrow: DayOfWeek;
```

Для любого перечисляемого типа T = (W_0, W_1, ..., W_n), где T - идентификатор типа, а W_0, W_1, ..., W_n - константные идентификаторы, в силе следующие отношения и правила:

- W_i <> W_j, если i <> j (различимость);
- W_i < W_j, если i < j (упорядоченность);
- значениями типа T могут быть только W_0, W_1, ..., W_n;
- существует взаимно-однозначное соответствие между значениями типа и положительными целыми числами в интервале [0,n]. При том W_0 сопоставляется 0, W_1 - 1 и т.д.

Рассмотрим операции, применяемые для перечисляемого типа:

- присваивание значения. Переменной типа T можно присваивать новое значение. Например: CarColor:= Red; today:= Tuesday;
- отношение. Операции отношения (<, >, <=, >=, =,) определены для любых двух объектов одного и того же перечисляемого типа. Константы перечисляемого типа считаются упорядоченными в том порядке, в котором они появились в его описании;
- получение порядкового номера осуществляется стандартной функцией ORD, которая определена для любого перечисляемого типа и как результат дает неотрицательное число, представляющее собою порядковый номер константы в описании типа. Например: ORD(Black) = 0;
- получение значения перечисляемого типа по заданному порядковому номеру. Любой перечисляемый тип определяет функцию над целыми неотрицательными числами в перечисляемом типе с тем-же самым именем. Например: Color(1) = Red, DayOfWeek(6) = Saturday;
- получение минимального и максимального значения. Определены стандартные функции MIN и MAX над каждым перечисляемым типом, которые дают минимальное и максимальное значение этого типа, соответственно. Например: Min(Color) = Black, Max(DayOfWeek) = Saturday;
- получение следующего и предыдущего элемента. Определены стандартные функции Succ и Pred над любым перечисляемым типом и дающие результат того-же самого типа:

  Succ(W_i) = W_{i+1}, для i = 0, 1, ..., n-1
  Pred(W_i) = W_{i-1}, для i = 1, 2, ..., n

Например: Succ(Friday) = Saturday, Pred(White) = Blue.

В случаях, когда порядковый номер полученного результата выходит из интервала [0, n], операции над объектами являются неопределенными или приводят к ошибке (см. п. 9).

3.1.2. Логический тип

Логический тип формально описывается так:

type boolean = (false, true);

Поэтому все функции, определенные для перечисляемого типа, можно применять и для логического типа. Кроме того, над переменными логического типа определены и другие операции:

- NOT - Отрицание - логическое 'не';
- AND - Конъюнкция - логическое 'и';
- OR - Дизъюнкция - логическое 'или';
- XOR - Сумма по модулю 2 - исключающее 'или'.

Необходимо обратить внимание на то, что объект логического типа - это не только переменная или константа типа BOOLEAN, но и любое выражение типа BOOLEAN. Результат выполнения этих операций тоже логического типа.

Например: 55 < 77 = true

В Pascal-е принят следующий приоритет операций:

Самый высокий:	NOT и функции
	*, /, div, mod, and, &
	+, -, OR, |, XOR
Самый низкий:	=, <>, <, <=, >, >=, in

Операции одинакового приоритета выполняются слева направо. Если не использованы скобки, то операции выполняются в порядке убывания их приоритетов.

Например:

not P and Q	=	(not P) and Q
P <= Q and R	=	P <= (Q and R)
A = B and C <= D	=	(A = (B and C)) <= D

Последний пример, обычно, считается ошибкой так, как имеется в виду совсем другое: (A = B) and (C <= D).

В Pascal-е нельзя записывать двухсторонное неравенство. Например, вместо 0 < X < 22 следует записать (0 < X) and (X < 22).

В UniPascal-е вычисление логического выражения прекращается сразу после получения результата. Например, если нужно вычислить выражение P AND Q и значение P = FALSE, то Q не будет вычисляться, потому что значение выражения уже известно - FALSE.

3.1.3. Целые типы

В UniPascal-е есть несколько разных целых типов, представляющих целые числа в разных диапазонах. Все они совместимы в интервале их пересечения.

Тип	границы диапазона
Integer	-32767..32767
ShortInt	-128..127
Cardinal	0..65535
ShortCard	0..255
Natural	0..32767
LongInt	-2147483647..2147483647

Над объектами целого типа определены следующие операции: сложение (+), вычитание (-), умножение (*), деление (**div**), получение остатка деления (**mod**) и смена знака (одноместная операция минуса) (-). Если операнды двухместной операции принадлежат разным целым типам, то они приводятся к минимальному типу, включающему диапазоны обоих типов.

Следующие стандартные функции определены над целыми типами и дают целый результат:

- ABS(x) абсолютная величина аргумента;
- SQR(x) аргумент в степени 2 (x * x);
- все стандартные процедуры и функции перечисляемого типа можно использовать и для целых типов.

В языке Pascal сущестует стандартная константа MAXINT. Так как в языке UniPascal есть несколько целых типов, то не очень удобно будет, если введем еще несколько соответстующих констант для каждого из них. Для этой цели можно использовать стандартную (в языке UniPascal) функцию MAX. Например, MAX(INTEGER), MAX(CARDINAL) и т.н.

3.1.4. Тип диапазона

Часто заранее известно, что значения данной переменной будут только из определеного диапазона (или точнее, она должна принимать значения из этого диапазона). В этом случае, если эта информация будет передана компилятору, он сможет выбрать подходящее представление данных и способ проверки. Для этой цели в языке Pascal дефинирован тип диапазона:

SubRange= Constant '..' Constant.

В действительности все целые можно рассматривать как диапазоны единственого типа LONGINT.

Новый тип принимает (по наследству) все операции, дефинированные над базовым типом (тем, которому является подобластью), но принадлежащие ему значения находятся в закрытом интервале, определяемом двумя константами. Пример:

```
type WorkDays = Monday..Friday;          (* предполагаем, что уже *)
     WeekEnd =  Saturday..Sunday;        (* описан тип DayOfWeek *)
     digit =    0..9;
```

3.1.5. Символьный тип

Значения, которые могут принимать объекты символьного типа (CHAR) - это все символы 8-битового ASCII набора. К ним добавлена кириллица. Для 8-битового ASCII

набора следующие подмножества упорядочены и связаны (условие удовлетворяется болгарским стандартом):

- Арабские цифры (от 0 до 9)
- Прописные буквы латинского алфавита (от A до Z)
- Строчные буквы латинского алфавита (от a до z)
- Прописные буквы кириллицы (от А до Я)
- Строчные буквы кириллицы (от а до я)

Все стандартные процедуры и функции описанных выше типов, кроме вещественного, применимы и для символьного типа. Кроме них в Pascal-е определена стандартная функция CHR с аргументом целого типа, в результате выполнения которой выдается буква, порядковый номер которой (ASCII код) равен аргументу функции. В языке UniPascal необходимость в использовании этой функции отпадает из-за наличия возможности явного преобразования типа.

3.1.6. Вещественный тип

Вещественный тип (Real) в UniPascal-е включает вещественные числа. Точность представления - 7 цифр. Это означает, что только первые 7 значащих цифр любого вещественного числа определяют величину мантиссы. Порядок вещественных чисел в диапазоне [-37, 37].

Для вещественных чисел определены следующие операции: сложение (+), вычитание (-), умножение (*), деление(/), унарная операция (-). Если при операции деления (/) оба операнда целого типа, они приводятся перед выполнением к вещественному типу. Следующие стандартные функции определены над вещественным типом и результат всегда вещественного типа:

- ABS(x) абсолютная величина аргумента (x: Real);
- SQR(x) аргумент в степени 2 (x * x) (x: Real);

Следующие функции могут иметь параметр как вещественного так и целого типа, но результат всегда вещественного типа.

- SIN(X) - синус, аргумент функции задан в радианах;
- COS(X) - косинус, аргумент функции задан в радианах;
- LN(X) - натуральный логарифм;
- EXP(X) - экспоненциальная функция;
- SQRT(X) - квадратный корень аргумента;
- ARCTAN(x) - арктангенс, результат получается в радианах;
- INT(x) - целая часть аргумента;
- FRAC(x) - дробная часть аргумента.

Следующие функции имеют вещественный параметр, но результат всегда целого типа:

- TRUNC(X) - целая часть аргумента без округления;
- ROUND(X) - целая часть аргумента с округлением.

Переменной вещественного типа можно присвоить значение переменной целого типа, но не наоборот. Для этой цели необходимо использовать одну из функций TRUNC или ROUND.

3.1.7. Стандартные типы BYTE, WORD, LONGWORD

В UniPascal-е дефинированы еще следующие стандартные типы данных: BYTE, WORD, LONGWORD. Они не являются целыми типами. Их аналоги можно найти в языке Modula-2. Эти три типа данных вполне совместимы с другими типами, представление принадлежащих им значений которых занимает 1, 2 и 4 байта, соответственно. Над этими типами данных дефинированы следующие операции:

- AND - побитовая (byte($c3) and byte($66) = byte($42));
- OR - побитовая (byte($a5) or byte($5a) = byte($ff));
- XOR - побитовая (byte($c3) xor byte($66) = byte($a5));
- = - равенство;
- <> - неравенство;
- := - присваивание значения.

Другие операции над этими типами данных не определены и при попытке использовать такие, индицируется ошибка в процессе компиляции. Если эти типы данных используются в стандартной процедуре WRITE (для текстового файла), такие значения выводятся в шестнадцатеричной системе. Их использование в стандартной процедуре READ запрещается.

3.2. Составные типы данных

Простые типы - это типы, значения которых не имеют выраженной структуры. В Pascal-е предусмотрены и другие типы: составные (структурированные, сложные) и указатели (ссылки).

Составной тип характеризируется своим методом структурирования и типом своих компонент. Если компоненты составного типа тоже структурированные, тогда говорим, что тип имеет больше одного уровня структурирования. Над всеми составными типами, кроме файлового типа, можно применять операцию присваивания.

```
StructuredType=     ['packed'] (ArrayType  |
                                StringType |
                                RecordType |
                                SetType    |
                                FileType   ).
```

Перед определением типа можно поставить префикс "packed" (упакованный), указывающий транслятору на необходимость экономить память даже за счет неэффективного доступа к его компонентам.

3.2.1. Массивовый тип

```
ArrayType=          'array' '[' IndexType {',' 
                                IndexType } ']' 'of' Type.

IndexType=          OrdinalType.
```

Массив является регулярной (гомогенной) структурой. Любой объект типа массива (ArrayType) состоит из фиксированного числа компонент. Все компоненты относятся к одному типу, называемому базовым типом компонент (Type). Число компонент определяется при описании этого типа. Каждая компонента может быть явно обозначена с помощью имени переменной-массива, за которым в квадратных скобках

следует индекс. Индекс специфицирует компоненту массива. Его тип называется тип индекса (IndexType). Индексы можно вычислять. Они не так строго фиксированы как в FORTRAN-е или BASIC-е. Программист может задавать их. Единственное требование к индексу относится к его типу - он должен быть ординальным, т.е. он не может быть вещественным. Например, если даны дефиниции:

```
const AreaLimit = 100;
type AreaIndex = 0..AreaLimit;
      AreaType = array[AreaIndex] of integer;
var I: AreaIndex;
     Area: AreaType;
```

то Area[I] обозначает I-тую компоненту массива Area. Время доступа к любой компоненте массива не зависит от значения индекса этой компоненты массива. Кроме того, чтобы достичь I-той компоненты массива, не нужно проходить через все I-1 предыдущие компоненты. Поэтому говорим, что массив является структурой данных, которая допускает прямой доступ. Поскольку базовый тип компонент массива (Type) может быть любой тип кроме файлового, то компоненты массива могут быть и составного типа. В частности, если базовый тип компонент тоже массивовый, то исходный массив называется многомерным. Например:

```
var A: array [LoInd1..HiInd1] of
            array [LoInd2..HiInd2] of SomeType;
```

В этом случае через A[I][J] обозначаем J-тую компоненту (типа Type) I-той компоненты массива A. Синтаксис языка позволяет записать этот пример короче:

```
var A: array [LoInd1..HiInd1, LoInd2..HiInd2] of Type;
```

Компоненты массива можно обозначать и через A[I, J]. Если при описании массива задано n индексов, то массив называется n-мерным, а его компоненты обозначаются с помощью n индексных выражений.

Компонентам массива можно присваивать значения того же типа.

Если имеем две переменные-массивы одного типа, то можно одной присвоить значение другой. Например, если:

```
var A, B: AreaType;
```

то возможно присваивание A:= B; и оно означает:

"Для всех I из диапазона IndexType: A[I]:= B[I]"

Над объектами массивого типа определены и отношения "равно" (=) и различно (<>). Они применяются для всех компонент объектов и возвращают результат типа Boolean.

При работе с упакованными массивами с компонентами типа Char существуют следующие дополнительные возможности. Можно применять все операции сравнения: =, <>, <, <=, >, >=. Упорядоченность начинается с первой компоненты массива и определяется порядком, существующим для типа Char.

3.2.2. Стандартный тип STRING

Специфичным одномерным символьным массивом является тип string (строка, символьная строка). Его длина динамична. Она изменяется от 0 до некоторого максимального значения, определенного при описании типа и не превышающего 255.

```
StringType=          'string' [ '[' Constant ']' ].
```

Где Constant в диапазоне [1..255].

3.2.3. Записные типы

```
RecordType=        'record' FieldList 'end'.

FieldList=         (FixedPart [';']) | (VariantPart [';']) | (FixedPart ';'
                   VariantPart [';']).

FixedPart=         IdentList ':' Type { ';'
                   IdentList ':' Type }.

VariantPart=       'case' TagField 'of'
                      CnstList ':' '(' [FieldList] ')' { ';'
                      CnstList ':' '(' [FieldList] ')' }.

TagField=          [Ident ':'] OrdinalTypeIdent.

OrdinalTypeIdent=  Ident.
```

Запись (RecordType) в языке программирования Pascal - это структура данных, состоящая из фиксированного числа наименованных компонент, называемых полями. При том поля могут быть разного типа. Эта особенность делает запись наиболее общим и гибким типом данных. При определении записного типа для каждого поля задаются его имя и тип. Область действия имени поля - самая внутренняя запись, в которой оно определяется. Имя поля может совпадать как с идентификаторами блока, где описана запись, так и с именами полей других записей, но не может совпадать с именем поля на том же уровне структурирования записи.

Запись используется для описания упорядоченной последовательности величин. Поля записи могут быть разных типов. В связи с этим каждая компонента должна иметь свое имя. Как в случае массивов, так и в случае записей осуществляется прямой доступ к компонентам. В случаях ссылки к компоненте записи следом за именем переменной ставится точка, а затем - имя соответствующего поля.

Иногда в запись необходимо включить информацию, зависящую от другой, уже включенной в этой записи, информации. Синтаксис записного типа предусматривает вариантную часть (VariantPart), рассчитанную на то, что можно задавать тип, содержащий определение нескольких вариантов структуры. Это означает, что разные переменные, хотя и одного типа, могут иметь различные структуры. Различие может быть как в типах компонент, так и в их числе.

Вариантная часть записи состоит из селектора варианта (TagField) и одного или более вариантов. Структура и значения каждого варианта задаются его списком полей. Варианты специфицируются селектором варианта (TagField). Здесь OrdinalTypeIdent - тип, значения которого определяют варианты, а Ident - имя поля признака, в котором записывается значение, соответствующее определенному варианту. TagFieldType может быть имя ординального типа (OrdinalType).

Каждый вариант состоит из списка констант типа TypeIdent и соответствующего списка описаний компонент, заключенного в скобках. Каждая константа должна встречаться только один раз в данной вариантной части.

Если существует поле признака, то активным будет вариант, которому предшествует константа, равная значению поля признака.

Над объектами записного типа определены операции сравнения на равенство (=) и неравенство (<>).

Например:

```
type BookType = record
        Name: string[31];
        case AuthorKnown: boolean of
           true: (Author: string[33]);
           false: ();
        end;
var Book: BookType;
```

Теперь можно записать:

```
Book.Name = 'Pascal User Manual and Report';
Book.AuthorKnown:= TRUE;
Book.Author:= 'K. Jensen, N. Wirth';
 (* ... текст программы ... *)
write(Book.Name, ' written by ');
if Book.AuthorKnown then
   writeln(Book.Author)
else
   writeln('Unknown author.');
```

ПРИМЕЧАНИЯ:

- все имена полей должны быть уникальными, даже если они встречаются в разных вариантах одной записи;
- если некоторому значению не соответствует вариант, его можно не указывать, т.е. не все значения типа признака обязательны (поле опускается);
- любой список полей может иметь только одну вариантную часть, которая должна следовать за фиксированной частью записи;
- каждый вариант может содержать в себе вариантную часть, т.е. допускаются вложенные варианты;
- область действия идентификаторов констант перечисляемого типа, вводимых в записном типе, расширяется на вложенные блоки.

3.2.4. Множественный тип

SetType= 'set' 'of' OrdinalType.

Множественные типы обеспечивают компактную структуру, в которой сохраняется информация о группах значений, относящихся к ординальному типу. Множество значений переменной множественного типа включает в себя все (включая пустое) подмножества элементов базового типа. Значение переменной множественного типа является множеством элементов типа, совпадающего с базовым типом.

В UniPascal-е число элементов не должно превышать 256 с порядковыми номерами в диапазоне от 0 до 255.

Над объектами множественного типа можно применять следующие операции:

- равенство множеств =
- неравенство (различие) множеств
- включение одного множества в другое >=, <=

- объединение множеств +
- пересечение множеств *
- разность множеств -
- принадлежность к множеству . IN.
  Левый операнд должен быть базового типа, а правый - множественного типа. Операция возвращает результат TRUE, если первый операнд является элементом второго, и FALSE - если не является.

В UniPascal-е над объектами множественного типа определены и следующие стандартные процедуры: Incl(s, x) и Excl(s, x), где параметр s является множеством, а параметр x - элементом базового типа множества. Эти процедуры включают в множество или исключают из множества элемент x, соответственно. Результатом действия процедуры Incl(s, x) является s:= s + [x], а процедуры Excl(s, x) - s:= s - [x]. Множество не изменится включением элемента, уже принадлежащего ему, и исключением элемента, непринадлежащего ему.

Множественное значение можно задать с помощью конструктура множества (Set), в котором содержатся описания элементов множества, отделенные друг от друга запятыми и заключенные в квадратных скобках.

SetConstructor= '[' [SetElement {',' SetElement}] ']'.

SetElement= Expression ['..' Expression].

Пустое множество (обозначаемое через [ ]) принадлежит всем множественным типам. Обозначением X..Y, где X и Y являются выражениями, дефинируется множество, состоящее из всех элементов базового типа из закрытого интервала [X, Y]. Если X > Y, тогда [X..Y] определяет пустое множество. Когда значение элемента при конструировании множества выходит вне возможностей реализации (порядковый номер элемента вне интервала [0, 255]), получается ошибка и о ней сообщается в процессе компиляции (если этот элемент является константой) или в процессе выполнения (если он является переменной).

3.2.5. Файловый тип

FileType= 'file' ['of' Type].

Последовательность является одной из основных структур данных. Для описания последовательности используется файловый тип. Ее компоненты (компоненты последовательности) должны быть одного типа Type. Здесь Type - имя или описание любого типа, кроме файлового типа.

Файл - это структура последовательного доступа, т.е. в любой момент времени доступна только одна компонента файла. Другие компоненты доступны при последовательном продвижении по файлу. Число компонент (длина файла) не фиксируется при определении файла. Файл, не содержащий ни одной компоненты, называется пустым.

Файл используется для представления последовательности объектов, число которых неизвестно во время создания программы. Поэтому файл не находится целиком в оперативной памяти, а расположен во внешней памяти и только одна часть находится в оперативной памяти.

Если слово "of" и описание типа (Type) пропущенны в описании файлового типа, тогда он называется файлом неопределенного типа и используется для осуществления прямого доступа к внешнему файлу. Его компоненты обрабатываются специальными процедурами.

Над объектами файлового типа можно применять только стандартные процедуры для их обработки. Подробное описание всех процедур можете найти в Приложении B:

- для обработки файлов определенного типа: EoF, Read, Write, Reset, Open, Rewrite, Close;
- дополнительные процедуры для обработки текстовых файлов: EoLn, ReadLn, WriteLn;
- для обработки файлов неопределенного типа: EoF, BlockRead, BlockWrite, Reset, Open, Rewrite, Close.

В UniPascal-е не поддерживаются определенные ISO Pascal-ем стандартные процедуры GET и PUT, а также и буферная переменная, связанная с файлом.

3.2.5.1. Текстовые файлы

Чаще всего используются файлы, компонентами которых являются символы. Это является самым удобным для людей способом общения с компьютером. В Pascal-е дефинирован стандартный тип TEXT. Его описание следующее:

```
type TEXT = packed file of char;
```

Текстовые файлы состоят из символов, которые организованы в строках. Процедуры READLN, WRITELN и EOLN дефинированы для таких целей.

Текстовые файлы являются привилегированными в сравнении с другими файлами и, хотя тип их компонент - символьный (CHAR), в этих файлах можно записывать или из них считывать данные всех стандартных типов (за искючением типа BOOLEAN). До совершения записи в текстовом файле, каждое значение преобразуется (неявным образом) со своего внутреннего представления в символьное представление и тогда записывается в файл. Аналогическим образом при чтении сначала совершается преобразование символьного представления во внутренее представление соответствующего типа и потом присваивается переменной.

3.2.5.2. Стандартные файлы

В UniPascal-е дефинированы следующие пять стандартных файлов:

- Input - стандартный файл ввода, обычно - клавиатура микрокомпьютера, но может быть перенаправлен к внешнему файлу операционной ситемой;
- Output - стандартный файл вывода, обычно - экран микрокомпьютера, но тоже может быть перенаправлен;
- Message - стандартный файл ввода/вывода; всегда работает с клавиатурой и экраном микрокомпьютера и не может быть перенаправлен;
- Auxiliary - стандартный файл ввода/вывода (в зависимости от UniDOS-а); чаще всего - серийный ввод и/или вывод;
- Printer - стандартный файл вывода; обычно - печатающее устройство, но из-за того что он связан с аппаратным модулем параллельного ввода/вывода, эго можно использовать и как файл ввода.

Этих пяти файлов нельзя использовать в качестве параметров процедур RESET, REWRITE и CLOSE (не будет никакого эффекта).

В заголовке программы можно задать несколько из этих 5 файлов как параметры программы. Если параметров нет, по умолчанию подразумеваются Input и Output. Если они будут перенаправленными, программа не будет читать с клавиатуры и/или писать

на экран, а будет выполнять операции над заданными ей при ее стартировании файлами. Иногда необходимо чтобы некоторые из сообщений не могли быть перенаправленными, а так же чтобы чтение совершалось обязательно с клавиатуры (например, когда сообщается об ошибке пользователю программы и ожидаются его указания о продолжении работы). В таком случае нужно явно указать, что необходимо использовать файл Message. С другой стороны может быть программе будет необходимо работать всегда с файлом Message и реже с файлами Input и Output. Необходимо ли каждый раз указывать явным образом использование файла Message в каждом вызове процедур READ и WRITE? В UniPascal-е реализована возможность указывать который из файлов будет подразумеваться как параметр стандартных процедур для работы с текстовыми файлами, когда параметр будет опущен.

Задание параметров программы перечислением используемых внешних файлов имеет существенное значение для программы. Точнее, особое значение имеет порядок перечисления этих файлов. Когда используется некоторая из процедур READ, READLN, ... без никакого файла, подразумевается использование первого из перечисленных внешних файлов с необходимым доступом для чтения или записи. Например, пусть рассмотрим следующую программу:

```
program Example1(input, output);
   var s: string;
begin
   readln(s);
   writeln(s);
end.
```

Эта программа считывает строку со стандартного входного файла Input и потом записывает ее в стандартный выходной файл Output. Если заголовок программы имеет вид:

```
program Example2(input, message);
```

процедура READLN будет использовать по умолчанию файл Input, но WRITELN будеть использовать файл Message. Если заголовок имеет вид:

```
program Example3(message, input);
```

то READLN и WRITELN будут использовать файл Message, потому что он удовлетворяет обоим типам доступа (для чтения и записи) и задан первым в списке внешних файлов. Задание других файлов после Message в списке не укажет влияния на программу. Считается признаком хорошего стиля программирования, если в заголовке программы перечисляются только все внешние файлы, которые используются программой. С этой точки зрения перечисления файла Input в заголовке программы Example3 не только ненужно, но и не хорошо, потому что этот файл вообще не используется программой.

Существует еще одна разница между файлами Output и Message. Если происходит запись символов, выходящая за правой границей строки, при файле Output текст переносится автоматически в следующую строку, а при файле Message такой перенос не происходит, т.е. все символы пишутся только в последней позиции строки экрана (тем самым остается только последний символ). Работа с стандартными файлами Input и Output описывается более подробно в Приложении D.

3.2.6. Упаковка в UniPascal-е

Как уже было сказано, в Pascal-е префикс packed может предшествовать описанию каждого составного типа. Это значит, что внутреннее представление будет упакованным для экономии памяти. К сожалению, такое представление может привести

в некоторых случаях к большим затратам памяти из-за усложненного доступа к таким упакованным компонентам. Другими словами, съэкономленная упаковкой данных память уйдет на увеличение кода программы из-за того, что данные упакованы.

Поэтому в UniPascal-е воспринято компромиссное решение. Упаковка делается только на границе байта. Этим способом поддержка упакованных компонент делается легче.

При использовании упакованных типов надо иметь ввиду, что их компоненты не могут быть параметрами-переменными (см. п. 7).

3.3. Ссылочные типы

Ссылочный тип отличается от простых и составных типов тем, что его множество значений динамическое - значения любого ссылочного типа порождаются и уничтожаются в процессе выполнения программы. Разница между статическими и динамическими структурами состоит не только в моменте зарезервирования памяти, но и в том, что динамические структуры обычно используются для представления рекурсивных структур данных, т.е. структура, включающая явным или неявным образом себя в свое определение. Ее длина не известна в процессе компиляции и, обычно, меняется во время выполнения программы.

PointerType= '^' TypeIdent.

Каждый ссылочный тип (PointerType) состоит из неограниченного множества значений, указывающих на элементы базового типа (TypeIdent). Значением переменной ссылочного типа является адрес динамической переменной в оперативной памяти. Если базовый тип неопределен во время описания ссылочного типа, то он должен быть описан до конца текущего раздела типов. В противном случае выдается сообщение об ошибке.

Над указателями определены только операции присваивания (:=) и проверки на равенство (=) и различие (<>). Значение NIL (пустой указатель) принадлежит всем ссылочным типам. Оно не указывает ни на какой элемент. Для обработки объектов ссылочного типа используются следующие стандартные процедуры: New - создает динамическую переменную базового типа, резервируя часть оперативной памяти; Dispose - освобождает резервированную для некоторой динамической переменной память; Mark - отмечает память некоторой динамической переменной для последующего освобождения; Release - освобождает динамическую память, отмеченную процедурой Mark.

Процедуры MARK и RELEASE являются заместителями процедуры DISPOSE. Который из этих двух методов MARK-RELEASE или DISPOSE будеть использован для освобождения занятой памяти - вопрос конкретной необходимости и организации данных в программе. Не рекомендуется одновременное использование обоих методов в программе, потому что это может привести к неосвобождению памяти (см. п. 12).

3.4. Идентичность и совместимость типов данных

В процессе составления программы необходимо знать отношения переменных разного типа. Программисту, обычно, интуитивно ясна совместимость типов данных (т.е. переменные каких типов могут участвовать в одном выражении). Компилятору, с другой стороны, совместимость строго определена.

В Pascal-е введена сильная типизация. Типизация бывает структурной и именной. Вообще говоря, при структурной типизации два типа совместимы, если их структуры

одинаковые. При именной - не существуют два различные типа (с разными именами), которые были бы совместимы. В ISO Pascal-е, к которому придерживается и UniPascal, типизация представляет собою что-то среднее между обоими видами с уклоном к именной.

3.4.1. Идентичность типов данных

Идентичность типов (type identity) обязательна только при совместимости между фактическими и формальными параметрами-переменными процедур и функций. Одно название подсказывает, что если два типа идентичны, они не отличимы друг от друга.

Оба типа T1 и T2 называются идентичными, если справедливо одно из следующих условий:

- T1 и T2 представляют один и тот-же тип (например, тип integer идентичен себе);
- если T2 определен декларацией type T2 = T1 (или type T1 = T2).

Для идентичных типов справедливы и следующие условия (реляции эквивалентности):

- T1 идентичен сам себе;
- если T1 идентичен типу T2, то и T2 идентичен типу T1;
- если T1 идентичен типу T2 и T2 идентичен типу T3, то T1 идентичен типу T3 (транзитивность).

При именной сильной типизации T1 идентичен только себе и никому другому.

Примечание: При передачи параметров типа STRING не требуется идентичность типов. Достаточно только, чтобы фактический параметр имел длину, не меньшую максимально допустимой длины формального параметра.

3.4.2. Совместимость типов данных

Сместимость типов данных (type compatibility) необходима при использовании переменных разных типов в выражениях (арифметических и сравнениях).

Два типа называются совместимыми, если удовлетворено любое из следующих условий:

- оба типа - идентичные;
- оба типа - целые;
- один представляет собою диапазон другого;
- оба типа - диапазоны некоторого исходного типа и их множества значений имеют общее пересечение;
- оба типа - множественные, причем их базовые типы - совместимы;
- оба типа - упакованные массивы символов (CHAR) с одинаковым числом компонент;
- один представляет собою строка (STRING), а другой - или тоже строка или стандартный тип CHAR;
- один представляет собою стандартный тип POINTER, а другой - любой ссылочный тип;

- один представляет собою стандартный тип BYTE (WORD или LONGWORD), а другой - любой тип, для реализации которого необходимы 1 (2 или 4) байт памяти.

3.4.3. Совместимость по присваиванию

Совместимость по присваиванию (assignment compatibility) необходима при присваивании значения переменной или при передачи фактического параметра по значению.

Значение, относящееся к типу T2, называется совместимым по присваиванию с типом T1, если справедливо любое из следующих условий:

- T1 и T2 - идентичные;
- T1 и T2 - совместимые ординальные типы и значение, относящееся к типу T2, находится в дипазоне допустимых значений типа T1;
- T1 - вещественный, а T2 - целый тип;
- T1 и T2 - строки (STRING);
- T1 - строка (STRING), а T2 типа CHAR;
- T1 и T2 - совместимые упакованные массивы с базовым типом CHAR;
- T1 и T2 - совместимые множественные типы, при том все элементы, принадлежащие множеству типа T2, находятся в диапазоне элементов, которые могут принадлежать типу T1;
- T1 и T2 - совместимые ссылочные типы;
- один представляет собою стандартный тип BYTE (WORD или LONGWORD), а другой - любой тип, для реализации которого необходимы 1 (2 или 4) байт памяти.

4. Переменные

Переменная обладает типом, определяемым ее описанием и может принимать значения только этого типа. Любая статическая переменная, описанная в некотором блоке, порождается при активизации блока и уничтожается по его окончанию. Динамические переменные порождаются или уничтожаются при помощи процедур NEW и DISPOSE.

Обращение к переменной означает одно из следующих:

- полная переменная;
- переменная-компонента - средство обращения к компоненте составного (массивого или записного) типа;
- динамическая переменная.

5. Выражения

Любое выражение представляет собою правило вычислений, при выполнении которых выражению присваивается некоторое значение. Выражение состоит из операций и операндов. Значение выражения зависит от значений констант, границ типов, значений переменных, операций и функций, включенных в выражение. Тип полученного результата зависит от типов операндов и операций, из которых состоит выражение. Этот тип однозначно определен и известен еще в процессе компиляции.

```
Expression=          (SimpleExpression [relationOp
                       SimpleExpression]) |
                     ExpTypeCast.

SimpleExpression=    ['+' | '-'] Term {AdditiveOp Term}.

Term=                Factor {MultiplicativeOp Factor}.

Factor=              Constant       |
                     VariableRef    |
                     SetConstructor |
                     FunctionCall   |
                     'not' Factor   |
                     '(' Expression ')'.
```

5.1. Операнды

Операндами могут быть константы, переменные, обращения к функциям и конструкторы множеств.

Обращение к функции:

```
FunctionCall=        QualIdent [ ActualParamList ].
```

Не рекомендуется использовать функции, вызывающие побочные эффекты, потому что операнды могут быть вычислены в порядке, отличающемся от заданного текстом программы.

Использование переменных, которым не присвоено значение до начала вычисления выражения, является ошибкой. Если она не приводит к другим ошибкам, ее нельзя регистрировать ни в процессе компиляции, ни в процессе выполнения программы. Такая ошибка чаще всего приводит к получению неопределенного результата.

При вычислении арифметических выражений возможно возникновление ошибки. Речь идет о переполнении. Оно возникает, когда результат арифметической операции не принадлежит диапазону допустимых значений. Когда тип результата - целый, такая ошибка не всегда может быть регистрирована.

5.2. Операции

Операции в Pascal-е можно разделит в три группы: мультипликативные, аддитивные и операции отношений.

```
relationOp=          '=' | '<>' | '<' | '<=' | '>' | '>=' | 'in'.

AdditiveOp=          '+' | '-' | 'or' | 'xor' | '|'.
```

MultiplicativeOp= '*' | '/' | 'div' | 'mod' | 'and' | '&'.

Когда порядок их выполнения не указан явным образом при помощи скобок, подразумевается следующий приоритет операций (дается по убывающему приоритету):

- 1) логическое отрицание (not)
- 2) мультипликативные операции (*, /, div, mod, and, &)
- 3) аддитивные операции (+, -, or, xor, |)
- 4) операции отношений (=, <>, <, <=, >, >=, in)

Последовательность операций с одинаковым приоритетом выполняется слева направо. Но порядок вычисления операндов не фиксирован, т.е. возможно, чтобы сначала будет вычислен правый (а не левый) операнд данной операции. Только вычисление операндов логических операций AND и OR всегда начинается с левого операнда и, если его значением определяется результат, правый операнд не вычисляется.

5.2.1. Арифметические операции

Арифметические операции выполняются над целыми или вещественными операндами и порождают целый или вещественный результат.

Унарные арифметические операции (знаки):

Операция	Действие	Тип операнда	Тип результата
+	тождественное	Integer или Real	тип операнда
-	изменение знака	Integer или Real	тип операнда

Бинарные арифметические операции:

Операция	Действие	Тип операндов	Тип результата
*	умножение	Integer или Real	Integer или Real
/	деление	Integer или Real	Real
div	деление	Integer	Integer
mod	остаток	Integer	Integer
+	сложение	Integer или Real	Integer или Real
-	вычитание	Integer или Real	Integer или Real

ПРИМЕЧАНИЯ:

- символы '+', '-', '*' применяются и для обозначения операций над множествами;
- здесь Integer обозначает все разновидности целого типа: Integer, ShortInt, Cardinal, Natural, ShortCard, LongInt.

Если оба операнда одного и того-же типа, то результат тоже этого типа. Если типы операндов различаются, тогда результат будет наименьшего типа, включающего значения типов операндов.

Например: Если тип одного из операндов ShortInt, а тип другого - Natural, то тип результата будет Integer, потому что оба типа являются его диапазонами.

Если в случае целочисленных операций оба операнда и верный результат находятся в диапазоне Min(LongInt)..Max(LongInt), то получается правильный результат. Но результат выражения зависит еще от типа переменной, которой он присваивается. Если результат выходит за границами ее типа, то ей присваивается неверное значение. Чтобы этого не получалось, необходимо знать ожидаемый результат вычисления выражения.

Вычисление терма x/y будет ошибкой, если y равен нулю. Результат операций сложения, вычитания и умножения будет целого типа, если оба операнда - целого типа. Если тип одного операнда вещественный, то получается результат вещественного типа. Операции и стандартные функции, дающие вещественный результат, всегда следует считать приближенными, а не точными. Их точность зависит от реализации.

5.2.2. Логические операции

Операция	Действие	Тип операндов	Тип результата
not	логическое "не"	Boolean	Boolean
and	логическое "и"	Boolean	Boolean
&	логическое "и"	Boolean	Boolean
or	логическое "или"	Boolean	Boolean
|	логическое "или"	Boolean	Boolean
xor	логическое исключающее "или"	Boolean	Boolean

ПРИМЕЧАНИЕ: Если после вычисления одного операнда операции AND или OR результат уже известен, то значение второго операнда не влияет на результат и поэтому не вычисляется. Если необходимо чтобы значение второго операнда всегда вычислялось, нужно применять операции '&' и '|', соответственно. Значения обоих операндов указанных операций вычисляются до конца.

Логические операции выполняются над операндами логического типа и тип их результата - логический. В некоторых реализациях (но не в ISO Pascal-е) логические операции применяются над операндами целого типа. Тогда они выполняются над парами соответствующих битов операндов (побитовые операции). В UniPascal-е не разрешается применение логических функций над операндами целых типов. Но их можно применять над операндами стандартных типов BYTE, WORD, LONGWORD.

5.2.3. Операции над множествами

Над объектами множественного типа можно применять следующие операции:

- равенство множеств =
- неравенство (различие) множеств
- включение одного множества в другое >=, <=
- объединение множеств +
- пересечение множеств *
- разность множеств -
- принадлежность к множеству IN

Оба операнда должны всегда относиться к совместимым типам.
Получение результата подчиняется множественной логике.

5.2.4. Операции отношения

Операции сравнений применяются над операндами любого типа (за исключением файлового), но результат всегда логического типа (Boolean).

= равенство, операция применяется над операндами любых совместимых типов за исключением файлового;

неравенство, операция применяется над операндами любых совместимых типов за исключением файлового;

< меньше, операция применяется над операндами совместимых простых типов, операндами типа STRING и упакованными массивами с базовым типом CHAR;

<= меньше или равно, операция применяется над операндами совместимых простых типов, операндами типа STRING и упакованными массивами с базовым типом CHAR. Операция применяется и над операндами совместимых множественных типов, но тогда она интерпретируется как включение левого операнда-множества в правом операнде-множестве;

> больше, операция применяется над операндами совместимых простых типов, операндами типа STRING и упакованными массивами с базовым типом CHAR;

>= больше или равно, операция применяется над операндами совместимых простых типов, операндами типа STRING и упакованными массивами с базовым типом CHAR. Операция применяется и над операндами совместимых множественных типов, но тогда она интерпретируется как включение правого операнда-множества в левом операнде-множестве;

IN принадлежность элемента множеству. Левый операнд должен быть ординального типа, а правый - множественного типа. Операция возвращает результат TRUE, если левый операнд принадлежит правому, и FALSE - если не принадлежит.

5.3. Переопределение типа данных (Type Cast)

Переопределение типа данных является расширением ISO Pascal-я. В UniPascal-е идентификатор каждого типа может быть использован в качестве идентификатора функции с единственным параметром. Эта функция возвращает в качестве результата значение заданного ей параметра, но уже принадлежит другому типу, определенному именем функции. Переопределением типа явным образом требуется изменение типа. Неявным образом совершается изменение типа при вычислении выражения, содержащего операнды разных типов.

ExpTypeCast= TypeIdent '(' Expression ')'.

VarTypeCast= TypeIdent '(' VariableRef ')'.

Явное переопределение типа применяется в двух случаях:

- переопределение типа переменной. Временно (только в этом месте и нигде больше) изменяется множество ее допустимых значений и множество допустимых операций. Размер переменной и размер нового типа должны совпадать или переменная должна быть нетипизированным параметром. В этом случае компилятор не создает дополнительный код, а на этом месте он считает ее декларированной как переменная переопределяемого типа;
- переопределение типа значения. Здесь компилятор генерирует код, который переобразует значение старого типа в значение нового типа.

Переопределение типа переменной или значения описывается единственным синтаксическим правилом. Компилятор определяет в зависимости от контекста семантику этого правила и предпринимает необходимые действия.

Примеры:

1) Использование побитовых логических операций над операндами целых типов. Так как эти операции дефинированы только над данными типа BYTE, WORD и

LONGWORD, используем необходимый тип для отдельных случаев. Пусть имеем определения

```
var i, i1, i2: integer;
    b, b1, b2: shortcard;
    l, l1, l2: LongInt;
```

тогда следующие операторы - правильные:

```
i:= word(i1) and word(i2);
b:= byte(b1) and byte(b2);
l:= longword(l1) and longword(l2);
```

Следующий оператор - синтаксически и семантически правильный, но результат ошибочный, т.е. он алгоритмически неправилен:

```
l:= word(l1) and word(l2);
```

Так как в случае происходит переопределение типа выражения, то значения l1 и l2 сначала будут преобразованы из LONGINT в WORD (отбрасыванием старшего слова) и тогда будет применена операция AND над словами, т.е. оператор эквивалентен следующему:

```
l:= (longword(l1) and $ffff) and (longword(l2) and $ffff);
```

2) Применение целых арифметических операций над операндами типа LONGINT когда их тип не LONGINT. Например, i1 * i2 различается от LONGINT(i1 * i2), если i1 = 300, i2 = 1000.

Обычно, компилятор сам определяет как организовать вычисления в зависимости от ожидаемого типа следующим образом:

- если левый операнд выражения принадлежит типу LONGINT, вычисление правого операнда выполняется при помощи LONGINT арифметических операций;
- при присваивании применяется аналогическим образом предыдущее правило: если переменная, которой будет присвоено значение, принадлежит типу LONGINT, вычисление выражения, выполняется при помощи LONGINT арифметики;
- передача фактических параметров-значений процедурам (функциям) аналогична оператору присваивания; если формальный параметр типа LONGINT, вычисления выполняются при помощи LONGINT арифметики;
- если оба операнда целого типа и минимальный диапазон, включающий в себя диапазоны обоих операндов, принадлежит типу LONGINT, то вычисления выполняются при помощи LONGINT арифметики;
- во всех остальных случаях, если не использовано явное переопределение типа (Type Cast), целые операции выполняются при помощи INTEGER или CARDINAL арифметики.

Ясно, что в некоторых случаях компилятор не сможет сам предпринять правильное решение относительно типа используемых операций. Например, если i = 10, то оператор

```
i:= i1 * i2 div i;
```

при i1 = 300 и i2 = 1000 получит ошибочный результат. Так как вычисления будут выполняться при помощи INTEGER арифметики, для этой цели необходимо явно указать на необходимость в переопределении типа. Но результат выполнения оператора

```
l:= i1 * i2 div i;
```

не будет ошибочным, так как ожидаемый тип LONGINT.

6. Операторы

Операторы определяют алгоритм действия. Перед любым оператором допускается метка, на которую можно ссылаться с помощью оператора перехода. Операторы объединяются в разделе операторов каждого блока. Раздел операторов является составным оператором (Compound Statement).

```
Statement=          [Label ':'] ( SimpleStatement |
                                  StructStatement ).
```

где Label может быть идентификатор или целое число, описанное в разделе описания меток.

6.1. Простые операторы

Простым называется оператор, в котором не включены как составные части другие операторы.

```
SimpleStatement=    EmptyStatement | Assignment |
                    ProcedureCall | GotoStatement.
```

6.1.1. Пустой оператор

Пустой оператор (EmptyStatement) не содержит никаких символов и не определяет никаких действий.

```
EmptyStatement=     .
```

6.1.2. Оператор присваивания

Оператор присваивания (AssignmentStatement) предназначен для замены текущего значения переменной на значение, полученное при вычислении выражения, или для спецификации выражения, результат вычисления которого возвращается функцией.

```
Assignment=         (VariableRef | FuncIdent) ':=' Expression.

VariableRef=        VarTypeCast |
                    (QualIdent {'.' Ident | '^' |
                    '[' Expression {',' Expression} ']'}).

FuncIdent=          Ident.
```

Значение выражения должно быть совместимо с типом переменной или типом идентификатора функции.

6.1.3. Оператор активизации процедуры

Оператор активизации процедуры (ProcedureStatement) предназначен для вызова процедуры, обозначенной идентификатором процедуры (ProcIdent). Если описание процедуры содержит список формальных параметров, то оператор процедуры должен иметь соответствующий список фактических параметров (Actual-ParameterList).

ProcedureCall= QualIdent [ActualParamList].

6.1.4. Оператор перехода

Оператор перехода (GotoStatement) указывает, что процесс выполнения программы должен быть продолжен с оператора, помеченного меткой Label.

GotoStatement= 'goto' Label.

При применении операторов перехода необходимо соблюдать следующие правила:

- метка (Label) в операторе перехода должна быть описана в тот же самом блоке, где встречается оператор перехода. Невозможно передать управление извне в процедуру или функцию. Так же невозможно оператором перехода передать управление вне процедуры или функции;
- передача управления оператору, который является частью сложного оператора, может привести к неожиданным эффектам;

6.2. Сложные операторы

Сложные операторы (Structured statements) - это конструкции, состоящие из других операторов, причем эти операторы либо выполняются последовательно (составные операторы), либо - в зависимости от условия (условные операторы и операторы выбора), либо повторяются (операторы цикла), либо выполняются в некоторой расширенной области действия (scope) (оператор присоединения).

StructStatement= CompoundStatement |
IfStatement |
CaseStatement |
RepetativeStat |
WithStatement.

6.2.1. Составной оператор

Составной оператор (CompoundStatement) задает выполнение последовательности операторов. Слова BEGIN и END выполняют роль операторных скобок.

CompoundStatement= 'begin' Statement { ';' Statement } 'end'.

Составной оператор, обычно, используется там, где синтаксисом Pascal-я допускается только один оператор, а необходимо использование нескольких. Например:

```
if OK then begin
   ReadData; Calculate; WriteData;
end { if };
```

6.2.2. Условный оператор (IF)

```
IfStatement=        'if' Expression
                      'then' Statement [
                      'else' Statement ].
```

где Expression должно быть логическое выражение стандартного типа Boolean.

Если логическое выражение имеет значение истина (TRUE), то выполняется оператор (Statement), следующий за словом THEN. Если логическое выражение имеет значение ложь (FALSE), то выполняется (если существует) оператор (Statement), следующий за словом ELSE.

Примечание: Синтаксическая двусмысленность конструкции:

```
if Exp1 then if Exp2 then Stat1 else Stat2
```

разрешается, считая, что она эквивалентна конструкции:

```
if Exp1 then begin
   if Exp2 then
      Stat1
   else
      Stat2
end
```

6.2.3. Оператор варианта (CASE)

Оператор варианта (CaseStatement) содержит ординальное выражение (индекс варианта) (selector) и список операторов. Перед каждым оператором списка стоит одна или несколько констант, относящиеся к типу индекса варианта, или слово 'else', последованное двоеточием.

```
CaseStatement=      'case' Selector 'of'
                      CnstList ':' Statement {';'
                      CnstList ':' Statement } [';'] [
                    'else' ':' Statement { ';'
                      Statement } [';'] ]
                    'end'.

Selector=           Expression.

CnstList=           Constant {',' Constant }.
```

Оператор варианта указывает, что необходимо выполнить оператор, перед которым стоит константа, равная значению индекса варианта. Если ни перед одним оператором не стоит константа, равная этому значению, то выполняется оператор, который находится в ELSE части, если она существует; если нет ELSE части, то управление передается оператору, следующему за телом оператора варианта.

6.2.4. Циклические операторы

Циклические операторы (RepetitiveStat) указывают на то, что выполнение некоторых операторов необходимо повторять. Число повторений может быть известно заранее или определяется предусловием или постусловием.

RepetativeStat= ForStatement |
WhileStatement |
RepeatStatement.

6.2.4.1. Цикл с предусловием (WHILE)

WhileStatement= 'while' Expression 'do' Statement.

Expression является выражением логического типа. Оператор (Statement) повторно выполняется до тех пор, пока выражение не даст значение ложь (FALSE). Каждый раз вычисляется значение логического выражения перед выполнением оператора. Если в самом начале значение логического выражения - ложь (FALSE), то оператор не выполняется вовсе.

6.2.4.2. Цикл с постусловием (REPEAT)

RepeatStatement= 'repeat' Statement {';'
Statement }
'until ' Expression.

Expression является выражением логического типа. Последовательность операторов повторно выполняется до тех пор, пока выражение не даст значение истина (TRUE) (по крайней мере один раз она выполнится). Значение логического выражения вычисляется после выполнения последовательности операторов.

6.2.4.3. Цикл с шагом (FOR)

Цикл FOR указывает, что необходимо повторять выполнение оператора и одновременно присваивать переменной, называемой управляющей переменной цикла, последовательно возрастающие (убывающие - если использовано зарезервированное слово DOWNTO) значения.

ForStatement= 'for' Ident ':=' Expression ('to' | 'downto') Expression 'do'
Statement.

Управляющая переменная (Ident) должна быть описана в разделе описаний переменных блока, в разделе операторов которого находится данный оператор FOR. Эта переменная должна относиться к ординальному типу, совместимому с типами начального и конечного значения.

Начальное и конечное значения вычисляются перед началом цикла и не изменяются во время его выполнения.

Оператор, содержащийся в цикле, выполняется один раз для каждого значения управляющей переменной. Если в цикле FOR использовано TO, значение управляющей переменной увеличается на единицу после каждого повторения оператора. Если начальное значение превышает конечное значение, то оператор (Statement) не выполняется ни один раз. Если в цикле FOR использовано DOWNTO, значение управляющей переменной уменьшается на единицу после каждого повторения оператора. Если конечное значение превышает начальное значение, то оператор (Statement) не выполняется ни один раз.

Никакой оператор внутри цикла не должен изменять значение управляющей переменной.

После нормального окончания цикла FOR значение управляющей переменной неопределено.

6.2.5. Оператор присоединения (WITH)

WithStatement= 'with' VariableRef {',' VariableRef } 'do' Statement.

Оператором присоединения (WithStatement) можно сократить описание обращений к полям переменной записного типа. Он открывает область действия, содержащую имена полей указанной переменной записного типа, так что эти имена могут фигурировать как имена обычных переменных. Внутри этого оператора поля переменной записного типа обозначаются с помощью только их имен. При трансляции каждый идентификатор проверяется на принадлежность к именам полей переменной записного типа (VariableRef). Если это так, то это интерпретируется как стандартное обращение к полю этой переменной.

Примечание: Если имя некоторой переменной совпадает с именем поля переменной записного типа, то в рамках оператора присоединения WITH эта переменная становится недоступной. Например:

```
var Field: integer;
        Rec: record
        Field, b: real;
      end;
(* ..................... *)
with Rec do begin
   Field:= 1;
end;
(* ..................... *)
```

В этом фрагменте программы значение 1 присваивается полю Field типа Real переменной Rec, а не целой переменной Field. После выхода из оператора with целая переменная Field снова становится доступной.

Для сокращенной записи множества вложений операторов with разрешается задавать список имен переменных или полей переменных. Оператор:

```
with r1, r2, ...., rn do .......
```

эквивалентен следующему:

```
with r1 do
   with r2 do
      ...........
         with rn do ........ .
```

Если при задании записи используется указатель или индексация массива, то эти действия совершаются только раз в начале оператора WITH. Например, следующие два фрагмента программы не эквивалентны. При том вероятнее всего выполнение второго приведет к непредсказуемым результатам, если значение переменной I не определено в начале оператора WITH. А если переменная I определена, то оператор WITH сделает 10 присваиваний одному и тому же элементу массива (в результате этот элемент будет иметь последнее присвоенное ему значение). Выполнением первого фрагмента программы присваиваются значения всем 10 элементам массива. Предположим, что сделано следующее определение:

```
var MyArray: array [1 .. 10] of record
        X, Y: integer;
      end { MyArray};
```

фрагмент 1

```
for i:= 1 to 10 do begin
  with MyArray[i] do begin
    X:= i; Y:= 10 - i;
  end { with };
end { for };
```

фрагмент 2

```
    with MyArray[i] do begin
for i:= 1 to 10 do begin
     X:= i; Y:= 10 - i;
   end { for };
    end { with };
```

7. Процедуры и функции

Процедура или функция - именованная часть программы, вызываемая оператором активизации процедуры или, соответственно, обозначением функции. Программист может описать новые процедуры и функции. Описания процедур и функций объединяются и составляют раздел описания процедур и функций. Именами процедур и функций являются идентификаторы, которые подчиняются правилам видимости идентификаторов.

7.1. Описание процедуры

Описание процедуры вводит идентификатор процедуры и ставит в соответствие этому идентификатору некоторый блок и, необязательно, список формальных параметров. Этот идентификатор называется именем процедуры.

ProcDeclaration= ProcHeading ';' (Block | Directive) ';'.

ProcHeading= ['segment'] 'procedure' Ident [FormalPList].

Directive= 'forward' | 'external' |
('code' IntConst {',' IntConst}).

Имя процедуры и список формальных параметров задаются в заголовке процедуры (ProcHeading). Тело процедуры является настоящим блоком. Описание списка формальных параметров дается ниже. Область действия имени процедуры и идентификаторов, описанных в списке формальных параметров - блок, непосредственно содержащий данное описание процедуры.

Процедура может быть описана с помощью одного единственного описания, состоящего из заголовка процедуры и ее тела. Это самый распространенный способ. Но есть и другие способы - опережающее описание и описание внешней процедуры.

Когда используется способ опережающего описания, процедуре дается два описания: первое описание состоит из заголовка процедуры и директивы FORWARD, а второе описание, которое должно находиться в том же разделе описаний процедур и функций, лишь идентифицирует процедуру и содержит ее тело. Во втором описании список формальных параметров можно не указывать. Но если указан, он должен совпадать со списком формальных параметров первого описания.

Описание внешних процедур позволяет программисту использовать процедуры и функции, составленные на другом языке программирования и оттранслированные отдельно. Внешний код должен быть связан с программой при помощи {$L filename } - директивы.

Применение имени процедуры в операторе процедуры внутри блока ее собственного описания предполагает рекурсивное использование этой процедуры.

При описании процедуры определяются все объекты, с которыми она работает, и действия над этими объектами. Если в теле процедуры встречаются некоторые идентификаторы, не описанные в блоке процедуры и не являющиеся формальными параметрами, то они считаются глобальными по отношению к данному описанию процедуры. Их описания должны находиться в начале блоков, обхватывающих данное описание процедуры.

Использованием префикса SEGMENT перед описанием процедуры указывается компилятору, что процедура сегментирована (оверлейная). В UniPascal-е этим способом организуются оверлейные программы. Это означает, что код, генерируемый для тело процедуры (вместе с вложенными в ней процедурами), будет

находиться в памяти компьютера только пока она (или некоторая из вложенных в ней процедур) не прекратит быть в состоянии активной. Если в программе (модуле) есть оверлейная процедура, в процессе выполнения программы не закрывается файл с кодом программы (модуля).

7.2. Описание функции

Описание функции задает часть программы, после выполнения которой явно возвращается результат. Описание определяет имя функции, ставит в соответствие с ним тип результата, блок и список формальных параметров (если список существует).

FuncDeclaration= FuncHeading ';' (Block | Directive) ';'.

FuncHeading= ['segment']
'function' Ident [FormalPList] ':' TypeIdent.

Заголовок функции (FuncHeading) определяет имя функции, тип результата и список формальных параметров.

Тип результата (TypeIdent) определяет тип значения, возвращаемого функцией после ее выполнения. Этот тип может быть простой (ординальный или вещественный) или ссылочный тип.

Блок любого описания финкции должен содержать по крайней мере один оператор присваивания, где имени функции присваивается значение, или одно использование стандартной функции RETURN. Использование имени функции в другом операторе кроме в левой части оператора присваивания предполагает рекурсивное выполнение этой функции.

7.3. Формальные параметры

Параметры позволяют при каждом вызове процедуры или функции работать с объектами (значениями, переменными), задаваемыми в точке активизации через список фактических параметров. Список формальных параметров в заголовке процедуры или функции определяет имена, под которыми эти объекты известны в блоке процедуры или функции, а также вид и тип требуемых фактических параметров.

FormalPList= '(' [Parameter { ',' Parameter }] ')'.

Parameter= (['var'| 'const'] IdentList ':' TypeIdent) |
('var' | 'const') IdentList.

Параметры, перечисленные в одной секции формальных параметров (Parameter), представляют собой либо параметры-значения (слово var им не предшествует), либо параметры-переменные (слово var предшествует им и их тип определен), либо параметры-константы (слово const предшествует им и их тип определен), либо нетипизированными параметрами - переменными (слово var или const предшествует им, но их тип не задан).

7.3.1 Формальные параметры-значения

Формальный параметр-значение (параметр передается по значению) действует как локальная переменная в процедуре или функции, исключая то, что получает начальное значение от соответствующего фактического параметра перед вызовом

процедуры или функции. Изменения этого формального параметра в теле процедуры или функции не отражаются на значение соответствующего фактического параметра.

Соответствующий формальному параметру-значению фактический параметр в операторе процедуры или при обращении к функции может быть выражением и не может быть файлового типа.

Фактический параметр должен соответствовать по типу формальному параметру-значению (совместимость по присваиванию).

7.3.2. Формальные параметры-переменные

Формальные параметры-переменные (параметры передаются по адресу) используются когда необходимо передать значение из процедуры или функции к тому, кто обращался к ним. Соответствующий фактический параметр из списка фактических параметров должен быть именем переменной. Формальный параметр-переменная представляет собою фактический параметр во время вызова процедуры или функции, так что каждое изменение значения формального параметра отражается на значение фактического параметра.

Тип фактического параметра должен быть идентичным типу формального параметра. Переменные типа файл можно передавать в качестве параметров только как параметры-переменные.

Компоненты упакованного типа нельзя сопоставлять формальным параметрам-переменным.

7.3.3. Формальные параметры-константы

Параметры-константы в UniPascal-е являются расширением ISO Pascal-я. Компилятор не разрешает изменения их значений в теле процедуры (т.е. им невозможно присвоит значение и их невозможно передать в качестве параметров-переменных другой процедуре). Когда они простого типа, параметры-константы передаются по значению, а если их тип составной, они передаются по адресу. Тем самым экономится память и время.

В качестве примера рассмотрим следующую функцию, вычисляющую детерминант матрицы 3x3, тип которой дефинирован как matrix_3x3.

```
 var matrix_3x3 = array [1..3, 1..3] of real;
function determinant(const a: matrix_3x3): real;
begin
   determinant:= a[1, 1] * a[2, 2] * a[3, 3] +
                 a[2, 1] * a[3, 2] * a[1, 3] +
                 a[1, 2] * a[2, 3] * a[3, 1] -
                 a[1, 3] * a[2, 2] * a[3, 1] -
                 a[2, 1] * a[1, 2] * a[3, 3] -
                 a[3, 2] * a[2, 3] * a[1, 1];
end { determinant };
```

Если перед формальным параметром не стоит зарезервированное слово CONST, фактический параметр будет передаваться по значению. А значение массива состоит из значений всех его элементов, т.е. функции будут переданы 9 вещественных чисел. На выполнение этой передачи уходит время и память. Обычно в таких случаях программисты продпочитают использовать параметр-переменную. Если параметр стандартного типа STRING, то возникают неприятности, связанные с использованием

зарезервированного слова VAR. В качестве примера рассмотрим программу, печатающую заданную ей строку. При том строка печатается заключенной в кавычках и вместо каждой содержащейся в ней кавычки печатаются две кавычки (как строка задается в тексте программы на Pascal-е):

```
procedure write_quote(const s: string);
   var i: integer;
begin
   write('''');
   for i:= 1 to length(s) do begin
      write(s[i]); if s[i] = '''' then write('''');
   end { for };
   write('''');
end { write_quote };
```

Эту процедуру можно вызвать с фактическим параметром, представляющим собою переменную типа STRING или симольную константу. Если вместо CONST напишем VAR, фактическим параметром может быть только переменная типа STRING. Если удалим слово CONST, ее фактическим параметром может быть и константа, но за счет времени и памяти.

7.3.4. Формальные нетипизированные параметры

Нетипизированные параметры (переменные и константы) в UniPascal-е являются расширением ISO Pascal-я. Они передаются всегда по адресу. Когда тип формальных параметров не указан в их декларации, соответствующим фактическим параметром может быть любое имя переменной, независимо от ее типа.

Например, если дано описание:

```
function Equal(var Source, Dest; Size: Cardinal): boolean;
   const MaxInt = Max(Integer);
   type Bytes = packed array [0..MaxInt] of byte;
   var N: Cardinal;
begin N:= 0;
   while (N < Size) and
            ( Bytes(Dest)[N] = Bytes(Source)[N]) do
      inc(N);
   Equal:= N = Size;
end; { Equal }
```

эту функцию можно применять для сравнения любых переменных любой длины.

Если даны описания:

```
type Vector = array [1..10] of integer;
      Point = record
         x, y: integer;
      end;
var v1, v2: Vector;
      p:       point;
      i:       integer;
```

то можно записать:

```
if Equal(v1, v2, SizeOf(Vector)) then
   { ..... }
   { Здесь сравниваются все компоненты переменных. }
if Equal(v1[1], v1[6], SizeOf(Integer)*5) then
   { ..... }
   { Здесь сравниваются первые 5 компонент переменной v1 с
     вторыми 5 компонентами }
if Equal(v1, p, SizeOf(point)) then
   { ..... }
   { Здесь сравниваются переменные разного типа и разной длины }
```

7.3.5. Формальные параметры типа STRING

Стандартный (в UniPascal-e) тип STRING занимает привилегированное положение по отношению к другим типам. При передачи параметров типа STRING не обязательно, чтобы формальный и фактический параметры имели идентичные типы. Достаточно только, чтобы максимальная длина фактического параметра была не меньше максимальной длины формального. Поэтому в определении формального параметра может стоять спецификатор максимальной длины. Например:

```
procedure ExpandTabs(var s: string[79]);
...
```

Так описанной процедуре ExpandTabs можно передавать переменные типа string[x], где x должно быть не менше 79.

Если формальный параметр-переменная типа STRING не имеет спецификатор длины, его тип не воспринимается как STRING[255] по умолчанию, как это было бы, если речь шла о переменных этого типа. Например, пусть процедура определена следующей декларацией

```
procedure ExpandTabs(var s: string);
...
```

В этом случае спецификатор длины фактического параметра передается процедуре неявным образом. Поэтому максимальная длина так дефинированного параметра типа STRING динамически изменяется, т.е. она неизвестна в процессе компиляции и варьирует для разных фактических параметров. Например:

```
procedure ExpandTabs(var s: string);
   var i: shortcard;
begin              {Эта процедура заменяет все символы <TAB> в данной строке}
   i:= 0;                                  {необходимым числом пробелов; каждая}
   repeat i:= i +1;                {табуляция заменяется таким числом пробелов,}
      if s[i] = #9 then begin                    {чтобы следующий символ оказался}
         s[i]:= ' ';                    { в позиции 8*K +1, где K= 0, 1, ... и 8*K +1}
         while i mod 8 <> 0 do begin                {больше текущей позиций}
            i:= i +1;          {<TAB> из 1-вой позиции замещается 8 пробелами}
            insert(' ', s, i);                  {из 2-рой позиции - 7 пробелами}
         end { while };                         {из 3-ей - 6, ..., из 8-ой - только 1}
      end { if };
   until i = length(s);
end { ExpandTabs };
```

Эта процедура сообщит об ошибке в процессе ее выполнения, если ей дадим такой фактический параметр, что если табуляции будут заменены необходимым числом пробелов, получится строка, длиннее максимальной (заданной как фактический параметр). Если подобная вероятность существует, а появление ошибки нежелательно, можем добавит дополнительное условие в начале цикла WHILE. Проверкой этого условия будем констатировать есть ли место для введения пробелов, т.е. действительная длина (SizeOf(s)) фактического параметра меньше текущей (максимальная длина строки равна SizeOf(s) -1, так как на представление строки в памяти уходить на один байт больше, который содержит ее длину - см. п. 12).

7.4. Фактические параметры

Список фактических параметров в операторе процедуры или описании функции определяет фактические параметры, которые при выполнении процедуры или функции должны быть подставлены в качестве формальных параметров. Если процедура или функция не имеет списка формальных параметров, то не должен быть и список фактических параметров.

ActualParamList= ['(' [Expression {',' Expression}] ')'].

Соответствие между фактическими и формальными параметрами устанавливается путем позиционного сопоставления параметров из соответствующих списков.

При сопоставлении компилятор следит о том, чтобы соответствующие типы сопоставляемых параметров были совместимы. При том, если параметр передается по значению, оба сопоставляемые типы должны обладать совместимость по присваиванию, а если параметр передается по адресу, оба типа должны быть идентичными. Исключения от этого правила состовляют следующие случаи:

- если формальный параметр нетипизированный, фактический параметр может быть переменной любого типа;
- при формальном параметре стандартного типа BYTE, WORD или LONG-WORD, фактический параметр может иметь любой тип лишь только его представление занимало в памяти 1, 2 или 4 байта, соответственно;
- если формальный параметр принадлежит стандартному типу POINTER, то фактическим параметром может быть любая переменная ссылочного типа;
- если формальный параметр-переменная типа STRING имеет спецификатор длины, достаточно чтобы фактический параметр имел не меньшую максимальную длину по отношению формального;
- если формальный параметр-переменная типа STRING не имеет спецификатор длины, максимальной длине фактического параметра не накладываются никакие ограничения;
- компоненты упакованного типа нельзя сопоставлять формальным параметрам-переменным.

Примечание: Некоторые стандартные процедуры и функции отступают от этих правил. Они позволяют при обращении к ним связывать с одним формальным параметром фактические параметры разного типа и разного вида. Поэтому лучше было бы называть их не процедурами или функциями, а более общим понятием - конструкциями языка. В настоящем документе будем придерживаться к установленному названию в литературе - процедуры и функции.

8. Модули

В UniPascal-е предусмотрена возможность, которой нет в стандартном Pascal-е - возможность раздельно-модульной компиляции (будем писать и только раздельной).

Некоторые части разных программ иногда совпадают. Если каждая программа должна быть единым целом при трансляции, то некоторые части с одинаковым предназначением должны фигурировать столько раз, сколько раз использованы в программе.

Для устранения указанного недостатка в UniPascal-е предусмотрена еще одна конструкция языка - модули (units).

Идея состоит в том, что модули могут храниться в "библиотеке" программ и к ним должно производиться автоматическое обращение при загрузке и выполнении программ. Таким образом, становится возможным заранее заготовить набор часто используемых программных фрагментов и тем самым избежать многократного программирования. Этот метод называется методом раздельной компиляции.

Модули сохраняются в библиотеке программ не в виде исходных текстов, а в компилированном виде. При загрузке главная программа, которая использует возможности одного или нескольких модулей, связывается с ранее компилированными модулями, из которых она импортирует объекты.

Во время трансляции импортирующей программы компилятор должен иметь доступ к описаниям объектов ранее откомпилированных модулей, из которых происходит импорт. Этот механизм отличает раздельную компиляцию от независимой компиляции. Любой вспомагательнный модуль может, в свою очередь, импортировать объекты из других модулей. Следовательно, программа может представлять собою целую йерархию модулей. Главная программа имеет высший уровень, а модули, которые не импортируют объектов - низший.

Чтобы компилятор знал об используемых модулях, список их идентификаторов вводится как UsesClause в текст главной программы.

```
Unit=                 'unit' Ident ['('IntConst')']';' 'interface'
                        InterfacePart (
                      'implmentation'
                        ImplmntPart |
                      'end') '.'.

InterfaceUnit=        'interface' 'unit' Ident ['('IntConst')'] ';'
                        InterfacePart
                      'end' '.'.

ImplmntUnit=          'implementation' 'unit' Ident ';'
                        ImplmntPart '.'.
```

От синатксических описаний видно, что за идентификатором модуля (в обычном модуле или в описательной части модуля) может стоять целая константа, заключенная в скобках. Этой константой указывается версия модуля (см. п. 12.6).

В UniPascal-е текст модулей состоит из двух разделов - раздел описаний (InterfacePart) и раздел реализации (ImplementationPart).

Чтобы не компилировать всегда обе части одновременно, предусмотрена возможность их раздельной компиляции. Эти два раздела обособляются как два модуля - модуль описаний (InterfaceUnit) и модуль реализации (ImplementationUnit). Поэтому существуют модули четырех видов:

- модуль-описание, содержащий описательную часть;

- модуль-реализация, содержащий только реализационную часть;
- обычный модуль, содержащий и обе части;
- модуль-только-описание (interface only unit), не содержащий реализации и ненуждающийся в такой части.

8.1. Раздел описания (interface part)

Раздел описания содержит описания экспортируемых объектов модуля. При импорте модуля достаточно иметь доступ только к его разделу описаний. Поэтому модуль разделяется на модуль описаний и модуль реализации. Оба модуля должны иметь один и тот же идентификатор. Модуль реализации остается недоступным для тех, кто импортирует объектов из него. Это используется для исключения неправомерного доступа к модулю реализации. Только тот, кто писал раздел реализации может здесь что-то изменять. Пока разработчик меняет лишь раздел реализации, ему не нужно сообщать о своих действиях тем, кто использует модуль.

```
InterfacePart=      [UsesClause] {
                    ConstDeclaration |
                    TypeDeclaration |
                    VarDeclaration |
                    PFDeclaration }.
```

Модуль - часть полного текста программы. Следовательно, объекты, описанные в разделе описаний, являются глобальными в том смысле, что они существуют в течении всего периода выполнения программы. Описания процедур и функций в разделе описаний состоят только из одних заголовков.

Следует пример модуля, реализующий стек для целых чисел. Для работы стека необходимы следующие четыре операции:

- втолкнуть элемент в стек;
- вытолкнуть элемент из стека;
- проверка на пустой стек;
- проверка на полный стек.

Проверка на полный или пустой стек реализуем логическими функциями.

```
interface unit STACK;
  function  full: boolean;
  function  empty: boolean;
  procedure push(x: integer);
  function  pull: integer;
end { STACK }.
```

Так описанный модуль STACK может быть использован любой программой, при том ее составитель не знает заранее размер стека и метод его реализации.

8.2. Раздел реализации (implementation part)

```
ImplmntPart=        [UsesClause]
                    Block.
```

Раздел реализации мало в чем отличается от обычной программы. Отличия следующие:

- не нужно и не надо декларировать снова объекты, описанные в описательной части. По правилам видимости их видно как глобальные идентификаторы;
- для всех процедур и функций, описанных в описательной части, необходимо задать реализацию. Описания в interface части рассматриваются компилятором как опережающие (forward) описания;
- все глобальные переменные модуля, описанные в обеих частях (interface и implementation), существуют все время с начала до конца выполнения программы (или модуля), использующей этот модуль;
- тело модуля (операторы, заключенные в BEGIN и END, implementation части модуля) выполняется до передачи управления первому оператору использующего его модуля и выступает в роли инициализирующей части;
- если в теле модуля будет использована метка EXIT, в процессе инициализации модуля выполняются только операторы, которые предшествуют этой метке. По окончанию выполнения программы (модуля), использующей этот модуль, управление передается помеченному этой меткой оператору. Этим способом можно предусмотреть выполнение необходимых действий для завершения работы этого модуля (закрытие файлов, очистка экрана и т.д.).

```
implementation unit STACK;
  const StackSize = 100;
  type StackIndex = 0..StackSize;
  var index: StackIndex;
        IsFull, IsEmpty: boolean;
        StackArray: array [StackIndex] of integer;
  procedure push(x: integer);
  begin
     if not IsFull then begin
        StackArray[index]:= x;
        inc(index);
     end { if };
     IsFull:= StackSize <= index;
     IsEmpty:= false;
  end { push };
  function pull: integer;
  begin
     if not IsEmpty then begin
        dec(index);
        pull:= StackArray[index];
     end { if };
     IsEmpty:= index = 0;
     IsFull:= false;
  end { pull };
  function full: boolean;
  begin
     return(IsFull);
  end { full };
  function empty: boolean;
  begin
```

```
    return(IsEmpty);
  end { empty };
begin { инициализация модуля STACK }
  writeln('Начало инициализации модуля STACK');
  index:= 0; IsFull:= false; IsEmpty:= true;
  writeln('Конец инициализации модуля STACK');
EXIT:
  writeln('Конец работы модуля STACK');
end { Stack }.
```

8.3. Модуль-только-описание (interface only unit)

Иногда нет необходимости в том, чтобы модуль имел реализационную часть. Типичным примером является модуль, который содержит только описания типов. В таком случае можно создать пустую реализационную часть. Единственная проблема состоит в том, что хотя и пустая, на ее создание, связывание и сохранение уходит время и память (оперативная и внешняя). Чтобы не делать этих лишних затрат времени и памяти, модуль можно описать как модуль-только-описание (без реализационной части). Синтаксическое описание модуля-только-описания отличается от синтаксического описания обычного модуля только тем, что в нем не присутствует реализационная часть (участок с IMPLEMENTATION до END). При создании программы, состоящей из нескольких модулей, удобнее будет все используемые в программе типы описать в одном модуле-только-описании. Таким является, например, модуль UniLEX с дистрибутивного дискета.

В модуле-только-описании можно декларировать типы, простые константы (без символьных строк), процедуры или функции, описание которых дается при помощи директивы CODE. Не разрешается декларирование переменных или символьных констант.

Следует пример модуля-только-описания, в котором даются описания типов дня недели, цвета и цифры:

```
unit MISC; interface
  type DayOfWeek = (Sunday, Monday, Tuesday, Wednesday,
                    Thursday, Friday, Saturday);
       color = (Black, Blue, Green, Cyan,
                Red, Magenta, Brown, LightGray,
                DarkGray, LightBlue, LightGreen, LightCyan,
                LightRed, LightMagenta, Yellow, White);
       CharDigit = '0' .. '9';
       IntDigit = 0 .. 9;
end { MISC }.
```

8.4. Использование модулей

Использование модулей программами или другими модулями делается при помощи клаузы USES. Зарезервированное слово USES предшествует списку используемых модулей.

```
UsesClause=          { 'uses' IdentList';' }.
```

Все экспортируемые модулями идентификаторы автоматически становятся видимыми как глобальные. При совпадении некоторого идентификатора с другим, описанным в главной программе, доступным будет второй. Доступ к первому становится возможным только использованием квалифицированного идентификатора, т.е. имя модуля и точка '.' должны предшествовать экспортированному идентификатору. Например, MyUnit.MyProc или MyUnit.MyVar.

Все стандартные идентификаторы в UniPascal-е декларированы в модуле STANDARD. Все они могуть быть использованы при помощи квалифицированных идентификаторов. Например, STANDARD.NEW, STANDARD.DISPOSE и т.д.

Если главная программа использует два модуля и в их разделах описаний описан один и тот же идентификатор, который не описан в главной программе, и в тексте главной программы компилятор встретит его имя, то он выдаст сообщение об ошибке, требуя у программиста определить модуль, которому принадлежит идентификатор.

Приведенный ниже пример поясняет существенные свойства модулей. Модуль stack экспортирует процедуру и функцию, которые, соответственно, вталкивают в стек или выталкивают из стека данные. Доступ к стеку можно осуществлять только через эти процедуры и функции, чтобы гарантировать его правильное функционирование.

```
program UseStack(message);
   uses Stack;
   var i: integer;
begin
   writeln('Начало программы.');
   writeln('Вталкивание целых чисел в стек...');
   i:= 0;
   while not full do begin
      writeln('Вталкивание ', i, ' в стек.');
      push(i); inc(i);
   end { while };
   writeln('Выталкивание чисел из стека...');
   while not empty do begin
      writeln('□ытолкнули ', i, ' из стека.');
   end { while };
   writeln('Конец программы.');
end { UseStack }.
```

В примере программе не нужен стек большого размера. Однако, если напишем программу, для которой необходимо увеличить его размер, достаточно будет на месте константы в реализационную часть поставить новую. При том не надо перекомпилировать программу, использующую стек. Она не изменится, даже если реализация стека будет использовать внешнюю память.

9. Компиляция и управление компиляцией

Compilation= Program | Unit | InterfaceUnit | ImplmntUnit.

Компилятор UniPascal-я может обрабатывать следующие объекты (Compilation):

- программа (Program);
- модуль (Unit);
- модуль-описание (InterfaceUnit);
- модуль-реализация (ImplmntUnit).

Процесс компиляции исходного текста управляется при помощи директив компилятору. Директива вводится в исходный текст в виде комментария со следующим специальным синтаксисом:

CompilerOption= '{$' directive '}' | '(*$' directive '*)'.

Другими словами, первым символом комментарной формы записи должен быть знак доллара "$".

ВНИМАНИЕ! В одной строке исходного текста (в одной комментарной форме) может быть записана только одна директива. Все, находящееся после первого пробела в комментарной форме, считается нормальным комментарием. Исключением составляют так называемые "переключающие" директивы (см. ниже). Для них допускается ввод более одной директивы в одной форме, но при условии, что директивы разделены запятыми БЕЗ ПРОБЕЛОВ. Каждая директива должна быть записана полностью в одной строке (в одной комментарной форме).

Если в исходном тексте найден текст несуществующей директивы, компилятор UniPascal выдает соответствующее сообщение, но не возникает ситуация ОШИБКА.

Директивы UniPascal-я можно разделить на три группы:

- ПЕРЕКЛЮЧАЮЩИЕ директивы - директивы включения или выключения определенных действий, определенных возможностей компилятора. Переключающие директивы состоят из однобуквенного идентификатора директивы и знака "+" (плюс - включить) или "-" (минус - выключить). Между идентификатором и знаком не должны находиться другие символы. Как уже было подчеркнуто, в одной директивной форме можно записать несколько переключающих директив, разделенных запятыми. Например, строки {$I+} {$R-} можно заменить строкой {I+,R-}
- директивы С ПАРАМЕТРОМ - после однобуквенного идентификатора директивы следует ввести (через пробел) соответствующий параметр выполнения директивы;
- директивы УСЛОВНОЙ КОМПИЛЯЦИИ - в качестве идентификатора применяется зарезервированное слово. Часть директив этой группы предусматривает ввод соответствующих параметров.

9.1. Переключающие директивы

Внимание! При описании переключающих директив в заголовке указан знак ("+" или "-" - "включить" или "выключить", соответственно), который является значением ПО УМОЛЧАНИЮ.

9.1.1. Проверка результата ввода/вывода (*$I+*)

Если генерация кода проверки результата выполнения операции ввода/вывода выключена, выполнение программы будет продолжено независимо от результата операции. Предполагается, что программист сам принял необходимые меры анализа хода выполнения программы.

При включенной генерации кода для проверки операций ввода/вывода, применение функции IOresult бессмысленно, потому что в случае ошибки выполнение программы будет остановлено до выполнения функции.

9.1.2. Modula-2 в UniPascal-е (*$M-*)

Этой директивой можно настроить компилятор так, чтобы он принимал Modula—2 подобные операторы. При выключенном состоянии компилятор будет допускать только операторы, описанные до сих пор. При включенном состоянии компилятор считает, что входной поток содержит операторы, принадлежащие языку, подобному Modula-2. Здесь даются только синтаксические определения операторов UniPascal-я при включенном состоянии директивы (*$M+*).

```
StatementList=        Statement {';' Statement }.

IfStatement=          'if' Expression 'then'
                        StatementList {
                      'elsif' Expression 'then'
                        StatementList } [
                      'else'
                        StatementList ]
                      'end'.

CaseStatement=        'case' Selector 'of'
                       ['|']{ CnstList ':' StatementList '|'}
                      ['else' ':' StatementList ]
                      'end'.

WhileStatement=       'while' Expression 'do'
                        StatementList
                      'end'.

RepeatStatement=      'repeat' Statemen {';'
                        Statement }
                      'until ' Expression.

ForStatement=         'for' Variable ':=' Expression ('to'| 'downto') Expression
                      'do'
                       StatementList
                       'end'.

WithStatement=        'with' VariableRef {',' VariableRef } 'do'
                          StatementList
                      'end'.

Block=                [ Declarations ]
                      'begin'
                        Statement { ';'
```

```
                Statement }
            'end' ident.
```

Как видно, уже нет необходимости в составном операторе (BEGIN ... END). Любой сложный оператор заканчивается словом END. Оператор REPEAT UNTIL не изменяется, но присутствует для полноты описания. Кроме изменения синтаксиса использование директивы (*$M+*) приводит и к использованию нового зарезервированного слово ELSIF. И последнее изменение: за зарезервированным словом END в конце тело процедуры, функции, модуля или программы должно стоять имя (идентификатор) процедуры, функции, модуля или программы, соответственно, т.е. изменяется синтаксическое описание блока.

9.1.3. Включение Modula-2 расширения или проверки имен (*$N-*)

Эта директива используется, обычно, в выключенном состоянии. При включенном состоянии предлагаются две возможности, которые выбираются состоянием переключателя M:

- при выключенном состоянии переключателя M компилятор требует, что в конце процедуры (функции) ее идентификатор следовал за словом END. Остальные операторы имеют присущий языку Pascal-я синтаксис;
- при включенном состоянии переключателя M компилятор считает, что правилен синтаксис, введенный переключателем (*$M+*) и, кроме того, он требует, что за каждым словом END стояло зарезервированное слово, с которого начинается сложный оператор.

Например:

```
{$N+,M-}                        {$N+,M+}
procedure TEST;                 procedure TEST;
begin                              begin
   if EOF then begin            if EOF then
      writeln('EOF');             writeln('EOF');
   end { if };                  end if;
end TEST;                       end TEST;
```

9.1.4. Автоматическая упаковка (*$P-*)

При включенном состоянии компилятор рассматривает как упакованные все массивы и записи. При выключенном состоянии компилятор рассматривает как упакованные только те массивы и записи, описания которых содержат зарезервированное слово PACKED.

9.1.5. Молчаливая компиляция (*$Q-*)

При выключенном состоянии переключателя компилятор сопровождает каждое сообщение об ошибке звуковым сигналом, а при включенном не сопровождает ("работает молча").

9.1.6. Проверка границ диапазона (*$R-*)

При включенном состоянии генерируется код, которым обеспечивается проверка на соблюдение границ диапазона индексов массивов, присваиваемых значений переменных типа диапазона и передаваемых значений параметров, если формальные параметры диапазонного типа.

При выключенном состоянии компилятор проверяет только значения констант и константных выражений.

9.1.7. Предупредительные сообщения (*$W-*)

Предназначение директивы - выключить диагностические сообщения на время компиляции определенного оператора. Такая необходимость возникает в случаях некоторых отладок и специальных действий, когда программист уверен в правильности программного текста.

Действие директивы распространяется только на один - следующий за директивой - оператор. После его компиляции выдача сообщений включается автоматически. В случае необходимости выключить сообщения о нескольких последовательных операторах, рекомендуем объединить их конструкцией begin ... end в составной оператор и записать директиву {$W-} перед ним.

Этот переключатель обычно используется при заполнении массива с данного его элемента определенным значением при помощи стандартных процедур FILL-CHAR/FILLWORD или при перемещении элементов одной части массива в другой массив при помощи стандартных процедур MOVE/MOVEWORDS. Например, пусть s определено так - var s: string и хотим записать 5 пробелов на места символов с третьего до девятого. Можем использовать цикл:

```
for i:= 3 to 3 +5 -1 do s[i]:= ' ';
```

Но обычно используется стандартная процедура FILLCHAR, потому что она работает быстрее и занимает меньше памяти:

```
fillchar(s[3], 5, ' ');
```

Компилятор выдаст предупредительное сообщение, потому что размер одного элемента - один байт, а заказано заполнение 5 байтов. В таком случае директива {$W+} должна предшествовать оператору активизации процедуры FILLCHAR;

9.1.8. Условная компиляция специального вида (*$Y+*)

Инструкция включена специально для решения некоторых проблем переносимости программ между разными Pascal-компиляторами путем реализации особого вида условной компиляции. Поясним идею на примерах.

Допустим, что в исходном тексте существует фрагмент

```
(*) writeln; (*)
```

Если выполнена директива Y-, то для компилятора фрагмент представляет собою комментарий ") writeln (". Если выполнена директива Y+, то фрагмент воспринимается как оператор writeln между двумя комментариями. Таким образом получаем возможность осуществить условную компиляцию типа:

```
{$ifOpt Y+} .... {$endif}
```

Рассмотрим фрагмент

```
(*) writeln; {*) readln; {*}
```

Если выполнена директива Y-, то фрагмент будет интерпретирован в качестве последовательности:

```
комментарий      ") writeln; {"
оператор           readln;
комментарий          "*"
```

Если выполнена директива Y+, то фрагмент будет интерпретирован в качестве последовательности:

```
комментарий          (*)
оператор           writeln;
комментарий      "*) readln; {*"
```

Таким образом директива позволяет осуществить условную компиляцию типа:

{$ifOpt Y+} ... {$else} ... {$endif}

При использовании условной компиляции этого вида нужно быть осторожными при составлении комментария, заключенного в (*), {*) и {*}.

Легче всего запоминатюся следующие правила:

- в первой части условной компиляции (заключенной в (*) и {*) или в (*) и (*)) можно использовать комментарий, заключенный только в фигурных скобках { };
- во второй части условной компиляции (заключенной в {*) и {*}), можно использовать комментарий, заключенный только в комментарных скобках (* и *).

9.2. Директивы с параметром

В настоящей реализации UniPascal-я директивы этого вида только две - для связывания с внешними процедурами (LINK) и для включения в исходный файл дополнительного файла (INCLUDE). Директивы описываются следующим синтаксическим правилом:

{$L <полное_имя_связываемого_файла>}
{$I <полное_имя_включаемого_файла>}

9.2.1. Включение файла в текст программы (INCLUDE)

Имя файла, которого необходимо включить в исходный файл программы, должно быть полным, т.е. компилятор не вставляет автоматически суффикс в имени файла. При применении директивы включения дополнительного файла следует соблюдать следующие ограничения:

- вложение директивы допускается до четвертого уровня;
- тело каждой процедуры и функции должно находиться полностью в одном файле;
- один комментарий должен находиться полностью в одном файле.

9.2.2. Задание файла для связывания (LINK)

При связывании внешних процедур компилятор должен знать с какого файла он можеть взять генерированный ассемблером код. Файл указывается при помощи директивы (*$L filename *). Имя файла должно быть задано полностью, т.е. компилятор не добавляет по умолчанию никакого суффикса. При применении этой директивы следует соблюдать следующие ограничения:

1) Директива не должна встречаться на последнем возможном (максимальном) уровне вложения INCLUDE файлов;

2) Поиск любой внешней процедуры производится в указанном данной директивой файле. Поэтому компилятор открывает для чтения указанный файл и закрывает его, если произойдет одно из следующих событий:

- встретит другую директиву LINK;
- окончит компилирование;
- начнет компиляцию другого сегмента (оверлея);
- откроет файл для включения в текст программы (INCLUDE);
- встретит директиву LINK без имени файла, т.е. (*$L*). Этим вынуждается компилятор закрыть файл с внешними процедурами.

3) Если компилятор закроет файл с внешними процедурами и тогда появится описание другой внешней процедуры, выдается ошибка.

9.3. Директивы условной компиляции

Идея условной компиляции состоит в том, что определенный фрагмент исходной программы компилируется только в случае выполнения некоторого условия. Идея и соответствующие механизмы хорошо известны программистам, имеющим опыт работы на языке ассемблера или Turbo Pascal для компьютеров типа IBM-PC-XT/AT.

9.3.1. Директивы DEFINE и UNDEF

Директивы предназначены для определения и удаления идентификаторов условной компиляции (ИУК) и имеют синтаксис:

{$define <идентификатор_условной_компиляции>}
{$undef <идентификатор_условной_компиляции>}

Одним из возможных условий компиляции программного фрагмента является определение или неопределение ИУК - т.е. является или не является указанный идентификатор идентификатором условной компиляции.

ИУК рассматриваются компилятором совершенно отдельно от остальных идентификаторов в тексте программы. Поэтому ИУК может совпадать по написанию с другим, нормальным идентификатором в программе.

Для компилятора UniPASCAL по умолчанию определен идентификатор условной компиляции UniPas.

9.3.2. Директивы IFDEF, IFNDEF, IFOPT, ELSE и ENDIF

Условная компиляция осуществляется при помощи конструкции типа:

```
{$IFxxx <условие>}
 {компилируемый в случае выполнения условия фрагмент программы}
{$ENDIF}
```

или

```
{$IFxxx <условие>}
 {компилируемый в случае выполнения условия фрагмент программы}
{$ELSE}
   {компилируемый в случае невыполнения условия фрагмент}
{$ENDIF}
```

где IFxxx одна из директив IFDEF, IFNDEF или IFOPT.

При применении директивы IFDEF условие представляет собой идентификатор условной компиляции и считается выполненным, если идентификатор определен (см. директивы DEFINE, UNDEF).

При применении директивы IFNDEF условие представляет собой идентификатор условной компиляции и считается выполненным, если идентификатор НЕ определен (см. директивы DEFINE, UNDEF).

При применении директивы IFOPT условие представляет собою переключающую директиву. Условие считается выполненным, если директива находится в указанном состоянии.

Поясним применение директив на следующем примере. Допустим, что необходимо выдать сообщение, если выключена генерация кода проверки результата операции ввода/вывода, а также сообщение о том определен или нет идентификатор Turn_on. Тогда возможна следующая реализация:

```
.....
{$IFOPT I-}
   writeln("Проверка результата ввода/вывода не производится!")
{$endif}
{$IFDEF turn_on}
   writeln("Идентификатор Turn_on определен!")
{$else}
   writeln("Идентификатор Turn_on неопределен!")
{$endif}
.....
```

10. Использование UniPascal-я

Первым шагом к выполнению программы на языке UniPascal после ее составления является введение ее текста в микрокомпьютере. Это делается при помощи некоторого текстового редактора, например, UniED. Компилятору UniPascal-я необходимо указать полученный текстовой файл при его стартировании.

Компилятор UniPASCAL, как и любая другая выполнимая в среде операционной системы UniDOS программа, вызывается простым указанием имени программы. Обязательный параметр - имя текстового файла, содержащего исходный текст программы. И так, вызов осуществляется командой:

UPC имя_исходного_файла

Если имя исходного файла не содержит расширение, по умолчанию считается, что расширение .PAS.

Основным результатом работы компилятора является файл, содержащий выполнимый вид компилированной программы. Имя файла соответствует имени паскальской программы с расширением .PGM.

Допустим, что необходимо компилировать исходный файл программы HAN (задача о перемещении ханойских башен), находящейся на дистибутивном дискете. Ниже следует приблизительно то, что увидите на экране в процессе компиляции:

```
A:\>upc han
UniPascal compiler Version 1.60 (c) 1989, 90 Software R&D Lab., Sofia
      han.pas(  29) HAN       s0
      han.pas(  30)  UNICRT    u1
      han.pas(  61)  PREPDISK p2
      han.pas(  73)  MOVEDISK p3
      han.pas(  75)   SHOWMOVE p4
      han.pas( 128)  MOVEDISK
      han.pas( 141)  GETNDISK p5
      han.pas( 144)   HELPUSER p6
      han.pas( 152)  GETNDISK
      han.pas( 172) HAN       (1293)
Successful compilation. 212 lines, 1293 bytes code.
Program compiled as han.pgm
```

Текст программы состоит из 212 строк. Генерированный компилятором код содержит 1293 байта (в этой длине не входит длина кода модуля UniCRT). Компилятор присваивает каждому сегменту уникальный номер, каждой процедуре (функции) другой уникальный (в рамках сегмента) номер. Главной программе сопоставлен сегмент № 0 (s0). Модуль UniCRT используется как сегмент № 1 (u1). Процедура PREPDISK компилирована под № 2 (p2). Она начинается с 61-ой строки программы. Процедуре MOVEDISK сопоставлен № 3. Она начинается с 73-ей строки программы. В нее вложена процедура SHOWMOVE, которой сопоставлен номер № 4. Она начинается с 75-ой строки. Тело процедуры MOVEDISK начинается с 128-ой строки и т.д. Тело главной программы начинается с 172-ой строки. В последней строке находится имя файла, в котором компилятор записал генерированный им код (HAN.PGM).

10.1. Стандартные расширения имен файлов

При разработке компилятора UniPASCAL приняты следующие стандартные (воспринимаемые по умолчанию) имена файлов:

- **.PAS** - расширение имен исходных файлов;
- **.PGM** - расширение имен файлов, содержащих выполнимые программы;

- **.SYM** - расширение имен файлов, содержащих интерфейсные части компилируемых модулей;
- **.BDY** - расширение имен файлов, содержащих тела (implementation) компилируемых модулей.

10.2. Задаваемые с командной линии параметры

При стартировании компилятора с командной линии UniDOS-а необходимо указать по крайней мере один параметр - имя файла, подлежащего компиляции. Можно записать и другие параметры. Тогда каждый параметр должен начинаться с косой черточки (/) или с минуса (-). Компилятор воспринимает следующие параметры:

/Dxxx дефинирование символов, управляющих условной компиляцией. В этом параметре **xxx** является именем символа для условной компиляции, которого хотите дефинировать. Задание каждого такого символа эквивалентно написанию строки {$Define **xxx**}, предшествующей тексту программы;

/Uxxx **xxx** является списком путей (paths), по которым компилятор будет искать компилированные interface части (*.SYM - файлы) использованных программой модулей;

/Ixxx через **xxx** обозначен список путей, по которым компилятор будет искать все файлы, заданные директивой включения файла - {$I fname};

/Oxxx этот параметр аналогичен параметру **/Ixxx**, но здесь задаются пути поиска файлов для связывания - {$L fname};

/Lxxx через **xxx** обозначено имя библиотеки, которую нужно использовать. Имя задается полностью (включительно пути). По умолчанию принимается \SYSTEM.UPL;

/Txxx **xxx** означает путь (path), где компилятор должен создать рабочий файл (необходимый во время компиляции);

/Sxxx **xxx** означает путь, где должен быть создан .SYM файл, который является результатом компиляции interface части модуля;

/Bxxx **xxx** означает путь, где должен быть создан .BDY файл, который является результатом компиляции implementation части модуля;

/Pxxx **xxx** означает путь, где должен быть создан .PGM файл, который является результатом компиляции программы;

/Cxxx этим параметром задаются одновременно три параметра **/Sxxx**, **/Bxxx** и **/Pxxx**;

/W-|+ по умолчанию принимается **/W-**. Если задано **/W+**, компилятор ожидает нажатия клавиши ENTER или ESC после каждой ошибки;

/$x-|+ задание начального состояния переключаемой директивы. Здесь '**x**' является буквой, которая должна быть обозначением одной из переключаемых директив. Начальное состояние директивы устанавливается в зависимости от знака: при '+' она включена, а при '-' она выключена;

@xxx этот параметр не начинается с / или с -, а с @ и задает имя **xxx** файла, который компилятор должен использовать в качестве конфигурирующего. Этого параметра нельзя включать в конфигурирующий файл.

Во всех случаях, если не задан путь или список путей, подразумевается текущая директория. Если задан список путей, компилятор ищет необходимых ему файлов в той последовательности, в которой задан список. Отдельные элементы списка разделяются точкой с запятой ';'. В текущей директории поиск производится, если необходимый файл не найден в заданных списком путей. Если необходимо искать сначала в текущей директории, а потом в других, можете включит ее в список. Например: /I;c:\work;d:\inc... или /I.;c:\work;d:\inc... (пустым именем или '.' обозначается текущая директория).

10.3. Конфигурирующий файл

В начале своей работы компилятор проверяет существует ли конфигурирующий файл **UPC.CFG** в текущей директории. Если его там нет, ищет и в директории, в которой сам компилятор находится. Если обнаружен файл **UPC.CFG**, компилятор вводит с файла параметры (которые можно ввести с командной линии). Конфигурирующий файл должен быть обычным текстовым файлом. Каждый параметр должен быть записан в отдельной строке. После введения параметров с конфигурирующего файла, компилятор проверяет параметры, введенные с командной линии. Они пользуются приоритетом относительно введенных с файла параметров. Это значит, что если некоторый параметр задан компилятору обоими способами (с командной линии и в конфигурирующем файле), он игнорирует заданный файлом параметр.

10.4. Связывание модулей и использование библиотеки

В UniPascal-е связывание программы с использованными ею модулями происходит в процессе ее выполнения, независмо от того где находится их компилированный код. Код каждого модуля физически может находиться в отдельном файле с именем модуля и суффиксом **.BDY** или в библиотеке (SYSTEM.UPL), или в файле, содержащем программу.

Добавление модулей к программам или к библиотеке совершается одной и той же программой **UPL**. Это так, потому что в UniPascal-е программа расматривается как библиотека, первый модуль которой является выполнимым. Поэтому к программе можно связать модули, которые она не использует явным образом или вовсе не использует.

Библиотека (и программа) представляет собою файл, в котором записано множество модулей. В ней можно записать описательные и реализационные части модулей.

Включение (добавление) в библиотеку (программу) и исключение модуля из библиотеки (или программы) совершается при помощи программы **UPL**.

Програма **UPL** имеет следующие параметры: первый из них представляет собою имя библиотеки (или программы). Другие параметры могут быть следующими:

+xxx где **xxx** - имя файла, в котором записан добавляемый к библиотеке модуль. При указанным суффиксе **.SYM**, добавляется только его описательная часть. При **.BDY** - только его реализационная часть. Если нет суффикса, добавляются обе части или только та, которая нужна и существует. Другими словами, если уже в библиотеке находиться описательная часть модуля, добавляется его реализационная часть, и наоборот;

-xxx где **xxx** - имя модуля, который подлежить уничтожению;

***xxx** где **xxx** - имя модуля, подлежащего вынесения из библиотеки. Вынесение осуществлятся следующим образом: создается файл с необходимым суффиксом (**.SYM** для описательной части и **.BDY** для реализационной); в этот новый файл записывается содержимое модуля; старый модуль (в библиотеке) уничтожается;

/m выводит список наличных в библиотеке (программе) модулей;

/s делает то же самое, что и **/m**, но кроме того дает информацию об отдельных сегментах (оверлеях);

/p делает то же самое, что и **/s**, но кроме того дает информацию и о каждой процедуре.

Если программе UPL будет задано только имя библиотеки (или программы), она выдаст список наличных в библиотеке модулей, сегментов и процедур.

Если ей не будет задано никакого параметра, она выдает вспомагательное сообщение о способе ее использования.

Пример: распечатка использования программы UPL относительно ее самой, т.е. выполнения команды

A:\>upl upl

UniPascal Librarian/Linker. version 1.53. (c) 1990 Software R&D Lab., Sofia.

Information for all modules in upl.pgm

module: UPL (1, 3.Feb.1991 14:58:48), 1 seg.

число сегментов в модуле

дата и час компиляции

версия модуля

имя модуля

seg UPL No 0/$00 Sz 7944/$1f08 ($18eb/$061d/$0000), 34 procedures.

число процедур

размер таблицы для перемещения

размер констант

размер Y кода

размер сегмента

номер сегмента

proc No 1/$01 size 121/$0079 lex 0 data 1434/$059a parm 0/$0000

число занятых параметрами байтов

число байтов локальных переменных

вложеность процедуры

размер Y кода для этой процедуры

proc No 2/$02 size 159/$009f lex 1 data 68/$0044 parm 6/$0006

....

10.5. Определение пути для .BDY файлов и для библиотеки

Связывание программы с используемыми ею внешними модулями (которые не добавлены к файлу, содержащему программу) происходит в процессе ее выполнения. Для этого необходимо указать каким путем можно достичь (т.е. где искать) их компилированных реализационных частей (**.BDY** файлы). Присутствие компилированных описательных (interface) частей модулей во время выполнения программы не является необходимым и поэтому не нужно задавать их местоположение.

По умолчанию поиск файлов с суффиксами **.BDY** начинается с текущей директории, а имя библиотеки - \SYSTEM.UPL. При помощи программы BLpath, находящейся на дистрибутивном дискете, можно задать список директорий для поиска **.BDY** файлов, а так же местонахождение библиотеки и/или ее новое имя.

Параметры программы BLpath следующие:

BDY=xxx этим параметром задается список путей, по которым необходимо искать файлов с суффиксом **.BDY**. Отдельные элементы списка разделяются точкой с запятой;

LIB=xxx задает имя библиотеки и полный путь для ее достижения;

? выдает вспомагательное сообщение о способе использования программы.

Если программа BLpath будет активизирована без параметров, она выдает заданный предыдущим вызовом (или установленный по умолчанию) список путей и имя библиотеки.

Поиск реализационной части каждого модуля выполняется следующей последовательностью:

1) В текущем файле, который не всегда является программой, если связывается файл с суффиксом **.BDY**.

2) В файле программы (**.PGM** файл);

3) В библиотеке.

4) Ведется поиск о наличии файла с именем модуля и суффиксом **.BDY** по путям, определенным списком.

10.6. Оптимизация программ на UniPascal-е

Из-за нехватка памяти и времени компилятор UniPascal-я создает не оптимальный код. Его можно оптимизировать использованием программы **YOP**. Она оптимизирует только сгенерированный компилятором код и оставляет без изменений ассемблерские подпрограмы.

Первый параметр программы YOP представляет собою имя программы, подлежащей оптимизацию. Вторым параметром определяется имя файла, в котором будет записан оптимизированный код. При отсутствии второго параметра оптимизированный код записывается на место неоптимизированного. Другие параметры относятся к методе оптимизации: по времени (параметр **/t**), по длине кода (параметр **/s**) и по обоим показателям (параметр **/b**), представляющий собою компромиссное решение.

Часто оптимизация по времени и по длине кода приводит к одинаковому эффекту, потому что Y код - компактный и длина большинства процедур не превышает 400 - 600 байтов.

10.7. Установка UniPascal-я

Здесь перечисляются подготовительные действия, которые необходимо совершить до того момента, в который станет возможно нормальное использование компилятора. Чтобы работать, компилятор (UPC) не нуждается в никаких дополнительных файлах, кроме исходного текста компилируемой программы. Но в этом случае сообщения об ошибках будут индицированы только кодами (порядковыми номерами). Текстовые сообщения об ошибках будут появляться только если в поддиректории, где находится компилятор, записан файл **UNIPAS.ERR**.

Если Ваш микрокомпьютер оснащен двумя дискетными устройствами, необходимые программы для создания, редактирования, сохранения, компилирования и выполнения Ваших программ можете записать на двух дискетах: На первом запишите компилятор (**UPC.PGM**), файл с текстами сообщений об ошибках (**UNIPAS.ERR**) и текстовой редактор (например, **UniED.CMD**). На втором будете записывать тексты Ваших программ и можете использовать его как рабочий.

Если Ваш микрокомпьютер связан в сети, лучше всего запишите все системные программы в главный компьютер (server), а используйте только локальные дискетные устройства.

Если Ваш микрокомпьютер оснащен только одним дискетным устройством, лучшее что Вы сможете сделать, это записать компилятор и исходный текст программы на одном дискете. В противном случае будете вынуждены часто менять дискеты во время работы компилятора.

На дистрибутивном дискете только компилятор, системная библиотека и текстовые сообщения об ошибках обособлены в отдельных файлах. Все системные программы (**UPL.PGM**, **YOP.PGM**, **BLpath.PGM**) находятся в файле под именем **UTILS.ARC**. Чтобы "вынуть" их из файла, необходимо применить программу **UNARC**. Лучше будет, если для этой цели будете использовать две (физических или логических) устройства. Поставьте в устройство A: пустой дискет, а в устройство B: дистрибутивный дискет. Стартирование программы UNARC следующее:

```
A:\> b:unarc b:utils
```

Програма **UNARC** распакует и запишет все системные программы, находящиеся в упакованном виде в файле **UTILS.ARC**.

Другой файл с суффиксом **.ARC**, находящийся на дистрибутивном дискете, это - **EXAMPLES.ARC**. В нем записаны некоторые примерные программы на UniPascal-е. Используя программы **UNARC**, Вы можете "вынуть" и их из этого файла-архива по уже описанному способу.

11. Кросс-продукты для IBM PC/XT/AT

В професиональной версии на дистрибутивных дискетах находятся и программы **UPC.EXE**, **UPL.EXE** и **YOP.EXE**. Они представляют собою, соответственно, компилятор, библиотечный редактор и оптимизатор, но работают на персональных компьютерах типа IBM PC/XT/AT под управлением операционной системы MS-DOS.

Способ их использования в этой операционной среде тот же самый, что и способ использования их основных вариантов для микрокомпьютера Пылдин. И так как операционные среды этих двух компьютеров под управлением, соответственно, UniDOS-а и MS-DOS-а совместимы на уровне дискового (дискетного) носителя, то все характеристики основных продуктов (для Пылдина) являются характеристиками и кросс-продуктов.

Процесс разработки программ для Пылдина значительно ускоряется, если используются кросс-версии, потому что компьютеры типа IBM PC - 16-битовые. Они оснащены памятью большего объема и обладают большим быстродействием чем Пылдин (для сравнения, кросс-компилятор работает от 6 до 10 раз быстрее на IBM PC/XT).

Генерированный кросс-компилятором код не отличается ничем от генерированного основным компилятором, и может выполняться на микрокомпьютере Пылдин. То же самое относиться и к библиотечному редактору и оптимизатору.

12. UniPascal в деталях

В этой главе предоставляется информация о реализации UniPascal-я для микрокомпьютера Пылдин. Она предназначена для программистов с большим опытом. Здесь рассматриваются вопросы, такие как распределение памяти, управление динамической памятью, внутреннее представление данных, вызов процедур и т.д.

UniPascal реализован методом, называемым смешанным методом компиляции и интерпретации. Компилятор UniPascal представляет собою программу на языке UniPascal. Результатом ее работы является псевдокод, называемый Y кодом. После компиляции этот код интерпретируется интерпретатором. Этот метод позволяет (при хорошо подобранном псевдокоде) получение сравнительно компактного кода. Основным недостатком этого метода является малое быстродействие в процессе выполнения программы по сравнению с выполнением машинного кода (без интерпретации).

12.1. Распределение памяти

Микропроцессор СМ 601 (MC6800) - 8-битовой и его адресное пространстово - 64 килобайта. Микрокомпьютер Пылдин 601/601A/601M оснащен оперативной памятью объемом 64 килобайта и имеет возможность для добавления постоянной памяти объемом до 64 килобайта. Это значит, что объем памяти микрокомпьютера может достичь 128 килобайтов (см. описание UniBIOS-a). Интерпретатор расположен в постоянной памяти. Область памяти с адресами от $0100 до $C000 предоставляется пользователю. Это значит, что пользовательская программа должна поместиться (вместе с данными) в область памяти объема не больше 48 килобайтов.

Распределение памяти микрокомпьютера в процессе выполнения программы выглядит так:

BIOS и текстовой экран	$F000
Системная часть оперативной памяти	$E000
8 x 8K ROM и 8K RAM (Здесь находится интерпретатор Y кода)	$C000
Часть памяти, использованная другими программами	$ZZZZ
Часть памяти для динамических переменных	$RRRR
Стек процессора СМ 601 (MC6800)	SP
↓ свободная память ↑	$PPPP
Локальные переменные процедур и глобальные переменные программы	$YYYY
Y код программы на UniPascal-е	$XXXX
Резервированное другой программой пространство (.cmd или .pgm)	$0100
Нулевая страница ($0000..$00FF)	$0000

До стартирования программы на UniPascal-е вся память с адреса $XXXX до SP (указателя стека) была свободной и адреса $RRRR и $ZZZZ совпадали (т.е. не было динамических переменных). Область памяти с адресами от $0100 до $XXXX занята некоторой программой, которая активизировала программу на UniPascal-е. Обычно, такой программы нет (т.е. программа на UniPascal-е выполняется без помощи другой программы) и поэтому $XXXX = $0100.

Область памяти с адресами от $ZZZZ до $C000 занята резидентными или системными программами. Обычно ее объем около 2-3 килобайта. Это значит, что свободная память обычно размером около 45-46 килобайтов.

После активизации программы на UniPascal-е ее код (Y код) загружается в область памяти с начальным адресом $XXXX. После того резервируется память для глобальных переменных программы. Если она использует модули, эта операция повторяется столько раз, сколько модулей у нее.

При активизации каждой процедуры резервируется место для ее локальных переменных. После ее выполнения занятая область памяти для ее локальных переменных освобождается. Параметры передаются процедуре при помощи системного стека. Если процедура оверлейная, то кроме этой области (для локальных переменных) резервируется место и для ее кода (который вводится с внешней памяти - с дискета). После окончания работы процедуры освобождается и занятая ею область.

Память для динамических переменных резервируется и освобождается через стандартные прерывания BIOS-а (int $2a и int $2b). Через BIOS легко реализуется механизм освобождения памяти по методу MARK - RELEASE. Полученные в следствии использования стандартной процедуры DISPOSE (FREEMEMWORDS) дыри обрабатываются отдельно. Они организуются в связанном списке. Попытка предоставления памяти удовлетворяется найменьшей подходящей дырой. Если такой нет, отправляется запрос к BIOS-у. При освобождении памяти через DISPOSE/FREE-MEMWORDS в динамической памяти появляется дыра. Освобождаемая дыра объединяется с соседней ей, если у нее есть сосед. Если дыра занимает верх динамической памяти, она освобождается через BIOS. Таким образом реализованные оба метода для освобождения памяти (DISPOSE/FREEMEMWORDS и MARK-RELEASE) могут быть использованы совместно. Процедуру MARK можно использовать только до наступления фрагментации динамической памяти. В таком случае применение RELEASE восстановит состояние динамической памяти в состояние, которое было до применения процедуры MARK.

Область свободной памяти находиться между указателем стека (SP) и концом локальных переменных ($PPPP). При предоставлении памяти для динамических переменных указатель стека идет сверху внизь, а при активизации процедуры адрес $PPPP увеличивается. Если они встретятся, т.е. если SP <= $PPPP, возникает ошибка - переполнение памяти.

Использование нулевой страницы

Нулевая страница микрокомпьютера Пылдин 601/601A/601M представляет собою та часть оперативной памяти, старший байт которой содержить $00, т.е. от $0000 до $00FF. Область этой страницы от $0000 до $007F резервирована для UniDOS-а и UniBIOS-а. Область от $0080 до $00DF используеться интерпретатором. Область с $00E0 до $00FF не используется интерпретатором. Пространство от $00B0..$00DF используется интерпретатором в качестве рабочей области. Ассемблерские подпрограммы тоже могут использовать эту область памяти как рабочая. Но нужно иметь ввиду, что интерпретатор тоже пользуется ею.

Таблица распределения нулевой страницы

$0000..$007f резервирована для систем BIOS, UniBIOS и UniDOS;

$0080..$00af резервирована для интерпретатора Y кода. Не должна использоваться ассемблерскими подпрограммами во время работы программы на Uni-Pascal-е;

$00b0..$00df рабочая область интерпретатора. Ассемблерские подпрограммы могуть использовать ее;

$00e0..$00ff не используется интерпретатором.

12.2. Внутреннее представление данных

Во всех случаях простые переменные (которые не являются элементами массива или записи) располагаются на границе слова (2 байта). Размер их представления кратен 16 битам, независимо от того сколько битов необходимо. Это относится и к элементам массива или записи. Исключение составляет только случай непосредственного указывания на упаковку (переменные типа STRING - упакованные по умолчанию).

12.2.1. Неупакованные переменные

При представлении переменных простых типов младший байт слова предшествует старшему (вопреки тому, что у микропроцессора СМ601 это наоборот), за исключением переменных ссылочного типа (их представление старший-младший).

Для переменных, тип которых позволяет представить их в небольше чем 8 битах, используются 16 битов. Это значит, что простая переменная типа SHORTINT, BYTE ..., всущности занимает два байта и интерпретируется как переменная типа INTEGER, WORD, ... Если включена проверка границ {$R+}, она совершается как нужно. Использованные 16 битов вместо 8 прозрачно для програмиста. Но если будут использованы некоторые из специальных процедур, например, MOVE, BLOCKREAD, FILLCHAR, ..., нужно обеспечить чтобы старший байт имел содержимое 0 (или $FF при отрицательном значении). Например:

```
function GetByte(var f: file): byte;
   var b: byte;
begin
   b:= 0;                                   { нулирование старшего байта }
   BlockRead(f, b, 1);
   GetByte:= b;
end { GetByte };
```

или

```
function GetByte(var f: file): byte;
 var b: byte;
begin
   BlockRead(f, b, 1);
   GetByte:= b and $ff;                     { только младший байт возвращается}
end { GetByte };
```

Если исключена проверка границ, при присваивании тоже возможно получение значения, не принадлежащего типу (множеству допустимых значений). Если проверка включена, в том случае получается ошибка в процессе выполнения. Использование

стандартных процедур READ и READLN для ввода с текстового файла не создает таких проблем.

Переменные перечисляемого типа

Переменные перечисляемого типа представляются как целые переменные с границами 0..N-1, где N - число констант перечисляемого типа.

Логические переменные

Как уже было сказано, логический тип может формально рассматриваться как перечисляемый тип, определенный следующим образом:

BOOLEAN = (FALSE, TRUE);

Это означает, что 0 представляет FALSE, а 1 - TRUE.

Целые переменные

Представление переменных типа **LONGINT** занимет 4 байта. Первый байт содержит младшую часть числа, следующие два байта - следующие по старшинству части двойного слова и четвертый байт - старшая часть двойного слова.

Остальные целые типы **INTEGER**, **CARDINAL** и их поддиапазоны представляются в двух байтах - сначала младшая, потом старшая часть слова. Для представления чисел со знаком используется дополнительный код.

Переменные типа CHAR

Переменные типа CHAR представляются значениями от 0 до 255, представляющими собою коды раширенного ASCII набора.

Вещественные переменные

Представления переменных вещественного типа занимают четыре байта согласно стандарту IEEE. Байты пронумерованы в обратном порядке. Знак располагается в 7-ом бите старшего байта, экспонента - в битах с 6-ого до 0-ого старшего байта и в 7-ом бите следующего байта. Мантисса занимает остальное пространство этого байта и остальные два байта.

Пусть через **e** обозначим порядок (exponent), через **m** - мантиссу (mantissa) , а через **s** - знак (sign) числа, тогда значение(value) определяется следующим образом:

(1) if $0 < e < 255$ then value = $(-1)^s * 2^{(e-127)} * (1.m)$;
(2) if (e = 0) and (m = 0) then value = $(-1)^s * 0 = 0$;
(3) if (e = 0) and (m <> 0) then value = $(-1)^s * 2^{-126} * (0.m)$;
(4) if (e = 255) and (m = 0) then value = $(-1)^s$*infinite;
(5) if (e = 255) and (m <> 0) then value is a NaN (Not A Number);

UniPascal не поддерживает:

- денормализованные значения (3). Получается 0.0;
- представление бесконечности (4). Получается переполнение;
- невалидные значения (5). Непредсказуемый результат.

12.2.2. Упакованные переменные

В UniPascal-е упаковка оказывает существенное влияние на представление переменных, массивов и структур. Упаковка предпринимается, только если она явным образом указана и если возможно ее выполнение.

Тип T называется упакованным, если указана его упаковка и среди его элементов есть по крайной мере один упаковаемый.

Тип T называется упаковаемым, если справедливо одно из следующих условий:

- Тип T ординальный (целый, логический, символьный, перечисляемый или диапазонный) и его минимальное и максимальное (и оба одновременно) значения принадлежать одному из следующих двух диапазонов [0..255] или [-128..127];
- Тип T - составной тип и все его элементы (если это массивовый тип) или все его поля (если это записный тип) упакованы и упаковаемые. При том для типа указана упаковка.

Из этих двух определений следует, что тип может быть упакованным, но неупаковаемым. Понятие неупаковаемости вводится для облегчения объяснения, что некоторые массивы могут, а другие не могут быть упакованными, а так же и для облегчения способа упаковки записей. Если элементы массива неупаковаемыми, то указание на упаковку (зарезервированное слово PACKED) игнорируется.

Примеры:

packed array [0..5] of char;

упакованный и упаковаемый массив;

packed array [0..5] of integer;

неупакованный массив;

packed record x, y: shortcard; end;

упаковаемая (и упакованная) запись, т.е. ее можно использовать как элемент упакованного массива (или записи);

packed record x, y: shortcard; i: integer; end;

упакованная, но неупаковаемая запись, т.е. если она будеть использована как элемент массива, то указание на упаковку (PACKED) будет игнорировано для этого массива.

Кроме того, упаковаемость оказывает существенное влияние на способ упаковки записей. Упаковаемые поля записи могут начинаться с границы байта, пока неупаковаемые поля должны начинаться с начала слова (выравнивание на границе слова).

Рассмотрим упаковаемость отдельных стандартных типов.

Перечисляемый тип

Значения констант перечисляемого типа представляются как целые числа диапазона 0..N-1, где N - число констант перечисляемого типа. Обычно представление занимает один байт и тогда тип упаковаемый, так как число констант редко превышает 256.

Логический тип

Для представления значений 0 и 1 достаточен один бит, но используется байт. Тип упаковаемый.

Целые типы

Если минимальное и максимальное значение находятся одновременно в границах:

- 0..255, представление занимает один байт (число без знака). Тип упаковаемый;
- -128..127, представление занимает один байт (число со знаком) в дополнительном коде. Тип упаковаемый;
- во всех остальных случаях тип неупаковаемый.

Типы CHAR и BYTE

Эти типы - упаковаемые и представление занимает один байт.

Вещественный тип

Вещественный тип неупаковаемый.

Стандартный тип STRING

Этот тип всегда упакованный, но неупаковаемый. Представление занимает N + 1 байта, где N - константа, указанная в декларации. Так как тип неупаковаемый, всегда занимаемое место начинается с границы слова и, следовательно, переменные типа string[7] и string[6] будут занимать одинаковый объем памяти.

Первый байт (соответствующий нулевому элементу) содержит длину текущей строки. Если длина **K** строки меньше максимальной длины, будут использованы только **K**+1 байта. Например, 'Test' имеет представление: $04, $54, $65, $73, $74.

12.3. Связь с ассемблерскими подпрограммами

Связь между программой на UniPascal-е и ассемблерской подпрограммой (машинный код) реализуется параметрами, передаваемыми через стек. В стеке они находятся в обратном порядке. Последний параметр вталкивается последним в стек и поэтому должен быть вытолкнутым первым. В момент активизации ассемблерской подпрограммы на самой вершине стека (над последним параметром) находится адрес возврата с подпрограммы (return address).

Чтобы интерпретотор мог различать ассемблерские подпрограммы от подпрограмм на Y коде, ассемблерские подпрограммы должны начинаться двумя байтами, заполненными нулями.

Подпрограмма должна вытолкнуть из стека все параметры, которые ей подала вызвавшая ее программа. Возвращение к интерпретатору можно осуществить двумя способами:

- использованием адреса возврата, находящегося в стеке;
- переходом по абсолютному адресу $BEFE;

В обоих случаях ассемблерская подпрограмма должна вытолкнуть все переданные ей данные.

Если подпрограммой реализуется функция, она должна расположить результат на вершину стека. Представление результата должно подчиняться принятым в UniPascal-е правилам, т.е. порядок байтов идет с младшего к старшему (на вершине стека должен быть младший, а под ним - старший байт) за исключением случая, когда результатом

является указатель. Кроме того представление результата должно занимать четное число байтов (т.е. если реультат занимает только один байт, его нужно расширить добавлением старшего байта, содержащего 0 при положительном числе и $FF при отрицательном.

Каждый параметр передается согласно принятым правилам представления данных в UniPascal-е, т.е. в стеке младший байт находится над старшим. Независимо от типа передаваемого параметра, его длина кратна слову, т.е. параметр занимает четное число байтов. Это значит, что при ожидаемых однобайтовых параметрах ассемблерская подпрограмма должна использовать только верхний байт, игнорируя находящийся под ним старший.

Передача параметров подчиняется еще следующим правилам:

При параметрах, передаваемых по значению, и при параметрах-константах:

- значение параметров, представление которых занимает одно или две слова (до 4 байтов), передаются через стек;
- адрес области памяти, занятой неудовлетворяющими верхнему условию параметрами, передается через стек. Модификация значения **нежелательно**.

При параметрах, передаваемых по адресу (var параметры), на вершине стека находится адрес области, в которой записан параметр.

Параметры типа STRING передаются специальным способом:

- VAR параметры типа STRING без описателя длины передаются так: в стек вталкивается адрес переменной и после него (т.е на вершину стека) - описатель длины фактического параметра. Пример: procedure MyASM(var s: string); активизируется оператором MyASM(sss); где var sss: string[77]; тогда в стек над адресом переменной находится слово, содержащее 77;
- VAR параметры типа STRING с описателем длины передаются обычным способом;
- CONST параметры типа STRING (с описателем длины или без описателя). Для таких параметров передается адрес параметра. Если фактический параметр - типа CHAR, компилятор сгенерировал код для совершения присваивания внутренней переменной типа STRING[1] для того, чтобы был передан адрес переменной типа STRING, а не типа CHAR;
- Параметры-значения STRING (с описателем длины или без описателя) передаются как CONST параметры, т.е передается их адрес. Если фактический параметр - типа CHAR, однако, передается его содержимое как адрес (сперва старший, потом младший байт), младший байт которого содержит ASCII код параметра, а старший - 0. Адреса, чей старший байт содержит 0, являются адресами байтов нулевой страницы, а она не используется для записи переменных. Этим способом ассемблерская процедура сможет различить что ей подали: адрес или параметр типа CHAR. Если Вам кажется неудобно обрабатывать такие параметры, опишите их как CONST. Тогда ассемблерская подпрограмма станет проще, но за счет этого, увеличится код (Y код) программы на UniPascal-е (каждому фактическому параметру типа CHAR соответствует код тремя байтами длиннее обычного).

12.4. Использование метки EXIT

Специфическая в UniPascal-е метка EXIT появляется в синтаксическом описании блока, как следует:

```
Block=          [ Declarations ]
                'begin'
                  Statement { ';' Statement } [
                'exit' ':'
                  Statement { ';' Statement } ]
                'end'.
```

Выполнение стандартной процедуры RETURN или EXIT в теле блока, в котором отсутствует метка EXIT, вызывает принудительное завершение работы блока, т.е. управление передается концу блока.

Наличие метки EXIT в теле блока изменяет семантику стандартных процедур EXIT и RETURN, если их использование находится после метки EXIT. Тогда они передают управление не концу блока, а помеченному этой меткой оператору. Обращения к процедурам RETURN и EXIT, находящимся за меткой EXIT в теле блока сохраняют их обычный семантический смысл и передают управление концу блока. Фрагмент блока с помеченного меткой EXIT оператора до конца блока выполняет завершительные работы по окончанию выполнения блока.

Случай, когда метка EXIT находится в инициализационной части модуля, очень близок к здесь описанному (см. п. 8.2). Пример применения метки EXIT в модулях дается в модуле STACK, находящемся на дистрибутивном дискете.

12.5. Оверлейные процедуры

Оверлей (перекрытие) процедуры реализуется с точки зрения программиста на языке UniPascal только прибавлением зарезервированного слова SEGMENT. Использование оверлейных процедур наложительно, если память компьютера не хватает на одновременное расположение в памяти всего кода программы и ее данных.

Оверлейные процедуры в UniPascal-е загружаются в память компьютера в момент их активизации. Код процедуры остается в памяти, пока процедура не перейдет в неактивное состояние. После окончания работы процедуры занятая ею память освобождается (код процедуры в памяти уничтожается). При новой активизации, она снова загружается в память. Чтобы загрузка оверлейной процедуры была возможна, файл с кодом программой (или модуля) должен оставаться открытым.

Простаивание файла в открытом состоянии не обходится даром операционной системе:

- допустимое число N открытых одновременно программой файлов, становится N-k, где k число файлов, содержащих код оверлейных процедур;
- дискет, на котором находится открытый файл, нельзя вынимать из дискетного устройства. (Если микрокомпьютер оснащен только одним дискетным устройством, используйте возможность UniDOS-а обеспечивать доступ к двум логическим устройствам на одном физическом.) Замена дискета, содержащего открытые файлы, может привести к уничтожению информации другого и/или информации обоих дискетов.

Необходимо особое внимание при связывании ассемблерских подпрограмм к оверлейным процедурам. Опасность возникает при связывании подпрограмм, которые записывают значения, необходимые им при повторном вызове. Код подпрограммы можеть быть уничтожен в памяти (вместе с оверлейной процедурой на UniPascal-е) и позже снова загружен в ней. Тогда сохраненные значения будут потеряны, т.е. они будут в своем начальном состоянии, записанном ассемблером. Например, пусть имеем ассемблерскую подпрограмму, которая заполняет некоторый буфер с целью отпечатать его либо на экране, либо на печатающем устройстве или обработать его

другим способом. Она посылает содержимое буфера лишь после его заполнения или если будет вызвана явная запись буфера.

```
; procedure SendChar(ch: char);
SendChar   ent                ;□ассемблерские подпрограммы, связываемые
           dw     0           ; с UniPascal-ем, начинаются с 0.
           ins               ; выталкивание адреса возврата
           ins               ; к интерпретатору
           pula              ; выталкивание значение параметра
           ins               ; и игнорирование его старшего байта
           ldx    buff_ptr; запись символа в буфер
           staa  x, 0
           inx
           stx    buff_ptr
           cmpa#13           ; символа <cr>?
           beq    SendIt   ; если да, посылка буфера
           cpx    #buff_end ; буфер полный?
           beq    SendIt   ; да, посылка буфера
           jmp    $befe       ; возвращение в интерпретатор (UniPascal)
; procedure SendBuff;
SendBuff ent                  ; начало подпрограммы
           dw     0           ; посылки буфера
           ins
           ins
SendIt
           ldx    #buffer ; буфер может быть пустым. Поэтому
           bra    test_x   ; сначала проверка на пустой буфер
loop
           ; ... запись / посылка / обработка одного символа
           inx
test_x   cpx     buff_ptr; конец буфера?
           bne    loop     ; цикл обработки всех символов
           ldx    #buffer ; задаем пустой буфер, т.е. подготовка к
           stx    buff_ptr; работе
           jmp    $befe       ; возврат к интерпретатору (UniPascal)
buffer    ds   1024    ; 1 килобайт буфер
buff_end
```

Примечание: В синтаксисе UniCross не существует директива ENT (она директива UniASM), в его синтаксис эта директива заменяется директивой public, записанной в отдельной строке.

Эти ассемблерские подпрограммы можно связать с обычной (не оверлейной) подпрограммой. Если подпрограмма оверлейная, этого нельзя сделать (так как содержимое буфера потеряется), если только Вы не абсолютно уверены, что подпрограмма на UniPasal-е окончит свое выполнение после того как послала буфер. Например, можете написать:

```
segment procedure some_proc;
begin
  repeat
    { ... обработка }
```

```
      SendChar(ch);
    until OK;
exit:
    SendBuff;
end { some_proc };
```

В конкретном случае возможно включение ассемблерской подпрограммы в оверлейной процедуре. В общем случае это возможно, только если после окончания работы подпрограммы на UniPascal-е ассемблерские подпрограммы не нуждаются в никакой промежуточной информации, которую нужно было сохранять до их следующей активизации. При следующей своей активизации ассемблерские подпрограммы будут загружены (вместе с UniPascal-ской) с диска и, следовательно, они будут в начальном состоянии (как будто они вообще не были активизированными).

12.6. Модули и их версии

При использвании модулей их реализационная часть скрыта (или по крайней мере должна быть) для пользовательских модулей или программы. Если это так (программы или модули, использующие данный модуль, не предполагают конкретную реализацию и зависят только от описательной части), реализацию можно заменить полностью (например алгоритмическим изменениям). Эти изменения ведут к перекомпиляции реализационной части модуля, но программы и модули, использующие этот модуль, не должны быть перекомпилированы. Пользователи не должны знать об изменениях того рода. Если, однако, наступят изменения в описательной части модуля, его пользователи должны узнать об этом, чтобы сделать необходимые изменения своих программ. Например, если у некоторой процедуры появился новый параметр или ее уничтожили, использующий ее модуль не смог бы продолжить работать без изменения.

Иногда новое описание модуля почти совпадает с старым и различия состоят только в добавлении новых возможностей (константы, типы, переменные, процедуры или функции). В таком случае модули, которые использовали его в его старом виде, могут продолжить использовать его в его новом виде (новой версии), так как описание старой части не изменилось и, следовательно, она совместима с старой. Например, пусть модуль для работы с магнитной лентой описан следующим образом:

```
• nterface unit Tape;
  procedure rewind(tape_no: shortcard);
     {- перемотка ленты }
  procedure write_TM(tape_no: shortcard);
     {- запись лентового маркера }
  procedure write_buff(tape_no: shortcard; const buff; size: natiral);
     { запись буфера (buff) длиной size }
  procedure read_buff(tape_no: shortcard; var buff; var size: natural);
     {- чтение блока с ленты в buff, длина получается в size }
end { Tape };
```

После того как модуль вошел в употреблении и его уже используют несколько программ, исходный код (текст) которых не известен составителю модуля Tape, пришлось дополнить и/или изменить модуль возможностью управлять лентопротяжным устройством другого вида. Пока изменение не выйдет за пределами реализационной части, другие программы без изменения будут работать с новой конфигурацией. Но, пусть например, будет дабавлена процедура для чтения блока в обратном направлении:

```
procedure read_back(tape_no: shortcard; var buff; var size: natural);
   {- чтение блока в обратном направлении }
```

Такое изменение модуля не оставит без изменения его описательную часть. Но, если описание изменится добавлением этой новой процедуры, необходимо перекомпилировать все программы, которые используют его (если только не сохранят специально для них старый вариант и если он сможет работать с новыми лентопротяжными устройствами).

В UniPascal-е возможно добавление новых возможностей (декларации объектов) к описательной части модуля (как в примере) без перекомпилирования программ, которые использовали этого модуля еще до изменения. Это делается указанием порядкого номера версии модуля, заключенного в скобках после его идентификатора в описательной части (реализационная часть может перекомпилироваться). Например:

```
interface unit Tape (2);
   procedure rewind(tape_no: shortcard);
      {- перемотка ленты }
   procedure write_TM(tape_no: shortcard);
      {- запись лентового маркера }
   procedure write_buff(tape_no: shortcard; const buff; size: natiral);
      { запись буфера (buff) длиной size }
   procedure read_buff(tape_no: shortcard; var buff; var size: natural);
      {- чтение блока с ленты в buff, длина получается в size }
   ===  добавлено к второй версии ===}
   procedure read_back(tape_no: shortcard; var buff; var size: natural);
      {- чтение блока в обратном направлении }
end { Tape };
```

Все версии данного модуля совместимы в строго восходящем порядке, т.е. версия 2 совместима с версией 3, версия 3 - с 4 и т.д., но версия 7 несовместима с 6 или с версией того же самого порядкого номера 7, но перекомпилированной. Порядковые номера версии модуля не обязательно быть последовательными, т.е. после версии 1 может следовать версия 10, потом 50 и т.д. Компилятор не проверяет действительно ли новая версия совместима с старой. Если в верхнем примере процедура READ_BACK будет описана первой, модуль станет несовместимым со своей старой версией.

Если новая версия модуля компилирована как совместимая со старой, но в действительности несовместима, то программы и модули, компилированные с старой версией будут вести себя непредсказуемо.

Две версии одного модуля являются действительно совместимыми, если все новые объекты (константы, типы, переменные, процедуры или функции) добавлены (физически в исходном тексте) в конце описательной части модуля. Если хотя бы один из объектов поставлен перед последним из дефинированных в старом версии объектов, то новая версия является действительно несовместимой с старой. Снова повтаряем, что компилятор не проверяет на действительную совместимость новой версии модуля с старой и, если новая несовместимая версия будет использована программами, компилированными с старой, их поведение будет непредсказуемым.

Еще о совместимости. Если новая версия модуля действительно совместима с старой и компилирована с другим, большим порядковым номером версии, использующие старую версию программы будут работать правильно и с новой версией. Программа, использующая модуль, связывается с ним в процессе ее выполнения (при ее стартировании). При том в программе записаны имя и версия модуля, дата и время (час), в котором этот модуль был компилирован. Связывание программы с

используемым ею модулем может состояться, если наличен модуль под тем-же самым именем, под которым указан в программе, для которого справедливо одно из следующих двух условий:

- версия, дата и время (час) создания модуля совпадают с записанными в программе. Так обеспечивается идентичность модулей, т.е. что использованный и наличный модуль представляют собою один и тот же модуль;
- версия модуля больше той, которая записана в программе, и момент (дата и время дня) компилирования модуля является более поздним по отношению требуемой программой. Таким образом обеспечивается идентичность модуля в его различных версиях, т.е. что наличный модуль - это новая версия использованного модуля.

Так как фиксирование времени создания и перекомпиляции модулей очень важно, необходимо устанавливать дату и время дня в операционную систему (компилятор берет эти данные от нее), особенно при компилировании описательной части модуля.