Въведение

В света на програмирането Pascal отдавна е извоювал своето място. Създаден с учебна цел, той до такава степен се разпространи, че даде заявката си за професионален език. И при мини и микрокомпютрите вече е такъв. Голяма част от програмното осигуряване се прави на Pascal, а в научната литература той служи като еталон. С него се сравняват новопоявилите се езици, дават се алгоритми на псевдо-Pascal и т.н. Достатъчно е да се каже само, че такива езици като Modula-2 и ADA са негови наследници и продължители. Но, въпреки големите му успехи, трябва да спомена и някои от неговите недостатъци. На първо място е липсата на възможност за разделно-модулна компилация. На второ място е липсата на средства за връзка с файловата система на дадения компютър. Друг негов недостатък е късната поява на международен стандарт, довела до това, че всяка реализация на практика решава проблемите по свой, най-често различен от другите реализации, начин. Положението не се подобри и след появата на стандарта.

При създаването на проекта на езика UniPascal бяха преследвани следните цели:

- новият език да остане близък до широко разпространената за персоналните микрокомпютри IBM PC версия на Turbo Pascal;
- в него да намерят отражение ограниченията, произтичащи от малката мощност на микрокомпютрите от фамилията "Пълдин".

В резултат се получи език, за който може да се твърди, че макар и да не осигурява пълна съвместимост на програмното осигуряване с това на Turbo Pascal, то той създава всички предпоставки за постигане на такава преносимост с неголеми разходи (поради поддръжката на същия метод на условна компилация и на основните вградени типове integer, LongInt, string и др.).

1. Основни понятия в UniPascal

1.1. Основни символи в UniPascal

При съставяне на програми на UniPascal се използва следното подмножество от символния набор ASCII:

- арабските цифри (**0**, **1**, ..., **9**);
- големите и малки букви от латиницата (**A** - **Z**, **a** - **z**);
- следните символи: интервал **" # $ & ' ( ) * + , - . / : ; < = > [ ] ^ _ { | }**

Разрешено е и използването на други, освен описаните символи, но само в коментар или символен низ. Навсякъде, освен в току що указаните места, малките и главни букви са еквивалентни.

1.2. Лексеми и разделители

Програмата на Pascal се състои от лексеми и разделители.

Разделителите са: интервал, край на ред и коментар.

Лексемите се построяват от вече описаните символи и се делят на следните групи (в скоби са дадени примери):

- специални символи (+ - := <= ..);
- ключови думи и идентификатори (begin end counter myproc);
- числови константи (1 $ff 256 3.1415926e+00 2.56e2);
- символни низове ('UniPascal' 'UniDOS').

Между две следващи една след друга ключови думи, два идентификатора или две числа трябва да има поне един разделител или специален символ. Например end else, а не endelse, a*b, а не ab.

Вътре в лексемите не може да се появяват разделители, наличието на разделител означава, че има две лексеми, а не една.

Правилата, с които се описват буквите и цифрите в езика, са следните:

```
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'.
```

Както се вижда, символът за подчертаване '_' е равностоен на буква.

1.3. Идентификатори

Идентификаторите се използват за обозначаване на константи, етикети, типове, променливи, полета на записи, модули, процедури, функции и програми. Те са последователност от букви и цифри, започващи с буква. В UniPascal съществуват следните особености:

- като идентификатори не могат да се използват ключови думи;
- малките и големи букви са еквивалентни;

- не се налага ограничение на дължината на идентификатора, но само първите осем символа са значещи, т.е. два идентификатора със съвпадащи начални осем букви са еквивалентни;
- символът за подчертаване ('_') е равностоен на буква.

Пример за правилни идентификатори:

UniPascal uni_pascal counter x2 next

Примери за неправилни идентификатори:

max-value	съдържа минус
2_pi	започва с цифра
брояч	не е разрешено използването на кирилица

В някои случаи (при използване на модули) е необходимо уточняване на идентификатор чрез друг идентификатор (името на модула). Такива идентификатори се наричат квалифицирани (qualified). Уточняването (квалификацията) става като първо се пише квалифициращият идентификатор (името на модула), след него - точка и след това - квалифицируемият идентификатор. Например, за да използвате идентификатора MyProc, описан в модула MyUnit, може да запишете следното: MyUnit.MyProc.

Синтаксис:

```
Ident=          Letter { Letter | Digit }.
QualIdent=      [Ident '.'] Ident.
IdentList=      Ident { ',' Ident }.
```

1.4. Ключови думи и специални символи

В UniPascal следните идентификатори са резервирани и са с едно или няколко фиксирани предназначения. С по-тъмен шрифт са дадени допълнителните ключови думи (само за UniPascal).

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**	

Следните символи (и двойки символи) също са с едно или няколко фиксирани предназначения:

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

1.5. Числови константи

За записване на числа в UniPascal се използва десетична бройна система, като за реалните числа се използва десетична точка (а не запетая, както е прието у нас). За записване на целочислените константи в UniPascal може да се използва и шестнадесетична бройна система, като в този случай числото трябва да бъде предшествано от знака "**$**".

Буквата E в реалните числа се чете като 'умножено по десет на степен'.

Целочислените константи трябва да бъдат в интервала от –2147483648 до 2147483647 или, ако са в шестнадесетичен вид, от $0 до $FFFFFFFF.

За разлика от стандартния Pascal в UniPascal се допуска използване на символа за подчертаване вътре в числата. Това е направено за по-лесно възприемане на дългите числа. Например числата 1000000000 и 1_000_000_000 са еквивалентни за компилатора, но човек възприема значително по-лесно второто.

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.
```

Примери:

Öåëè ÷èñëà: 0, 1, 259, $0, $f, $7fff, $EF5

Ðåàëíè ÷èñëà: 0.0, 1.0, 259.0, 0.0005, 1.0E-23, 3.141596e1.

1.6. Символни низове

Символният низ (или просто низ) е последователност от няколко (нула или повече) символа (букви) от разширения ASCII код. При записването му той се огражда с кавички. Могат да бъдат използвани единични (') или двойни кавички (").

Например:

'Пример 1' "Пример 2".

Ако е необходимо самият символ за кавичка, с който е заграден низът, да се постави в него, то той трябва да бъде повторен. Ако низът е ограден с двойни кавички, то в него могат да се използват единичните без да трябва да се повтарят, и обратното.

Например:

"В този низ има единична (') кавичка",
'А в този има двойна (") кавичка'.
"Тук двойната кавичка "" трябва да се повтори"

Един частен случай на низовете е низ с дължина един символ. В този случай той може да бъде записан и по следния начин: **#n**, където **n** е ASCII кода на символа. Например: 'Y' = #89 = #$59.

Последователност от низове се обединява и обработва от компилатора като един низ. Например: "UniPascal" = "Uni" 'Pascal' = #85'ni'#80'ascal'

Общият вид на символните константи е:

```
CharConst=         '"' ASCII_8 '"'| "'" ASCII_8 "'"|
                   '#'IntConst.
                   StringConst =  { '"' { ASCII_8 } '"' |
                    "'" { ASCII_8 } "'" |
```

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

Тук ASCII_8 означава символ от осем битовия набор ASCII, като е желателно да не се използват управляващи символи (с кодове по-малки от 32). Ако това се налага, те трябва да се записват като се използва **#n**. Например: "Този низ съдържа <cr> " #13 " и <lf> " #10

1.7. Коментари

Коментар се нарича произволна последователност от символи, заградена с фигурни скоби **{** и **}** или от двойките символи **(*** и ***)**. Коментар може да се постави на всяко място в програмата, където може да се постави разделител.

Коментарите са предназначени за записване на произволни пояснения в програмата и се игнорират от компилатора, без да оказват някакво влияние на програмата. В Pascal коментари се пишат лесно. Препоръчва се възможно по-честото им използване.

При написването на коментар трябва да се спазват следните правила:

- коментарът трябва да бъде ограден с еднакъв тип коментарни скоби, т.е. не може да започнете коментар с фигурна скоба и да го завършите с *) и обратно;
- ако коментарът е ограден с фигурни скоби, в него не може да се среща затваряща фигурна скоба;
- ако коментарът е ограден с (* и *), в него не може да се среща последователността *);
- не може да се пишат вложени коментари;
- ако коментарът започва със символа "**$**", то това е специален коментар и служи за управление на работата на компилатора. Например {$I+};
- тъй като в UniPascal на програмиста е разрешено да записва текста на програмата в няколко файла, то всеки коментар трябва да се намира изцяло в един файл.

Синтаксис:

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

Примери за правилни коментари:

```
{ Това е един правилен коментар }
(* А това е друг такъв *)
{ Символите *) могат да се напишат, като те влизат в коментара }
(* А в този коментар може да
ползваме
фигурните скоби { и }, защото коментарът е ограден с другия вид коментарни скоби *)
{ $ този коментар започва с интервал, а не с $ }
```

Примери за неправилни коментари:

{ не са разрешени { вложени } коментарии }
{ коментарните скоби трябва да са от един и същи вид *)
{$25.03 започва с $ и се счита за директива към компилатора }

2. Обща структура на програмата

Всяка програма на UniPascal се състои от заглавие (Program Heading), списък на използваните в програмата външни модули (Uses Clause) и блок (Block).

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

Блокът се състои от два основни раздела (части): раздел за описание на обработваните данни (Declarations) и раздел за задаване на действията над тези данни. В раздела за описание се описват всички използвани от програмата (процедурата или функцията) идентификатори. Те могат да бъдат етикети, константи, типове, променливи, процедури и функции. За разлика от стандартния Pascal в UniPascal последователността на различните раздели за описание не е фиксирана. Единственото ограничение е всеки идентификатор да бъде описан преди да се използва.

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

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

2.1. Заглавие на програмата

В заглавната част на програмата се дава нейното име и могат да се изброят стандартните външни файлове, с които тя ще взаимодейства.

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

По-подробно описание на списъка от идентификатори, ограден със скоби, след името на програмата (списъка от външни файлове) се дава в главата 'Стандартни файлове'.

2.2. Раздел за описание на етикети

Всеки оператор може да бъде предшестван от етикет. За разлика от някои други езици за програмиране (например FORTRAN) всеки етикет, преди да се използва в тялото на блока, трябва да бъде описан в раздела за описание на етикети. Този раздел започва с ключовата дума LABEL, последвана от списъка на използваните етикети и завършва с ;. Етикет в Pascal може да бъде само цяло число в диапазона от 0 до 9999. В UniPascal като етикет може да се използва и идентификатор.

```
Label=              Ident | IntConst.

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

Етикетът EXIT е резервиран и не може да се описва в раздела за описание на етикети. Използването му предизвиква специално действие в тялото на процедурата,

функцията, програмата или модула, където е написан. За по-подробно описание на ефекта от използването му виж глава "UniPascal в детайли".

Пример: LABEL 777, stop, quit, 123;

2.3. Раздел за описание на константи

В раздела за описание на константи се дефинират идентификатори, които по-късно се използват като имена на описваните от тях константи. В началото на раздела се поставя ключовата дума const, а разделът в общия случай има следния вид:

```
ConstDeclaration=   'const' Ident '=' Constant ';' {
                            Ident '=' Constant ';' }.
Constant=           SignedRealConst | SignedIntConst |
                    CharConst | StringConst | ConstExpression.
ConstExpression=    Expression.
```

От синтаксиса се вижда, че описанието на всяка константа се състои от идентификатор, последван от знак за равенство, след който се записва стойността на константата. По нататък в програмата при срещане на така описания идентификатор се използва зададената при дефиницията му стойност. В дясната част на дефиницията може да стои не само проста константа (цяла, реална и т.н.), но и константен израз (т.е. израз, който може да се пресметне по време на компилация), в който могат да участвуват вече описани други константи. Например:

```
const N =   100;
      N2 =  N * N;
      PI =  3.141596;
      Name = 'UniPascal';
```

Не се разрешава описването на константи, чиято стойност се определя чрез самите тях, т.е. явно или неявно рекурсивни. Например:

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

2.4. Раздел за описание на типове

В езика Pascal не само има дефинирани няколко стандартни типа данни, но програмистът сам има възможност да дефинира свои собствени типове.

Когато се определи такъв собствен тип, обикновено той се именува чрез въвеждане на идентификатор. Описанието на типовете се прави в отделна секция.

```
TypeDeclaration=    'type' Ident '=' Type ';' {
                           Ident '=' Type ';' }.
Type=               TypeIdent | SimpleType | PointerType |
                    StructuredType.
TypeIdent=          Ident.
```

Строго погледнато, последното правило е излишно, но и в този случай, както и в много други, то се използва, като се прави опит да се даде семантичен смисъл на синтактичните правила.

В описанието на всеки нов тип могат да се използват само идентификатори на вече описани типове (с изключение на указателите).

2.5. Раздел за описание на променливи

Всяка използвана в програма, процедура или функция променлива трябва да бъде описана преди това в раздела за описание на променливи. Описанието на променливата задава името ѝ и я свързва с някакъв тип данни, определяйки по този начин и операциите, които могат да се извършват над нея. Разделът за описание на променливи започва с ключовата дума var и има следния вид:

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

2.6. Раздел за описание на процедури и функции

Описанието на процедури и функции в Pascal прилича на описанието на програма. Разликата е, че вместо ключовата дума program се използва procedure или function. Описанието им се състои от заглавна част и последващ я блок. Процедурата представлява подпрограма и се активира (извиква) чрез своето име. Функцията е друг вид подпрограма. Тя явно връща като резултат от изпълнението си някаква стойност и се използва като компонента на израз.

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

2.7. Правила за достъп и област на действие на имената

Всички идентификатори и етикети са локални за блока, в който са описани. Това означава, че извън този блок те са недостъпни. Идентификаторите са достъпни навсякъде в блока, включително и във вложените в него такива. Идентификаторите, описани в главната програма, се наричат глобални. Те са достъпни в цялата програма.

Трябва да се отчитат следните особености:

- всички етикети, били те глобални или локални, са достъпни само в блока, в който са описани, но не и във вложените в него такива;
- ако някоя променлива се използва като управляваща променлива на FOR цикъл, то тя трябва да бъде локална за блока, в тялото на който се намира самият цикъл;
- при вложените блокове е разрешено преописването на идентификатор от външния блок във вътрешния. Тогава за съответния идентификатор е валидно описанието, направено във вътрешния блок, и няма никаква възможност за използване на идентификатора в смисъла на описанието, направено във външния блок;
- всеки идентификатор (или етикет) може да бъде описан един единствен път в рамките на един блок. Изключение прави случая с вложеност на блоковете (описан по-горе) и дефинирането на идентификатора като поле на структурата запис (record). Но и там един идентификатор не може да се описва на две места в един и същи запис и на едно и също ниво на вложеност;
- описанието на всеки идентификатор става валидно от мястото, на което е направено до края на блока, в който се намира. Използването на идентификатора преди неговото описание е грешка. Изключение от това правило се допуска само при описание на указатели. Разрешава се описване на указатели към още неописан тип, но неговото описание (на неописания

тип) трябва да се появи до края на същия раздел за описание на типове, в който е описан указателя.

В UniPascal е разрешено неколкократно появяване на раздел за описание от един и същ вид. Премахнато е ограничението за реда, в който разделите трябва да се появяват. В стандарта на Pascal разделите са подредени в реда: описание на етикети, типове, променливи, процедури и функции.

Всяка програма на Pascal се счита описана като локален блок на някаква обхващаща я среда. В този по-външен блок са описани всички стандартни идентификатори. В UniPascal нещата стоят малко по-различно. Вашата програма автоматично използва модула STANDARD, така че всички стандартни идентификатори се считат описани в него. Вие може да преопределите всеки от тях, но се счита за лош стил това да се прави. Например:

```
type worktype = integer;
     integer =  real;
     real =     worktype;
```

По този начин ние разменихме смисъла на типовете integer и real. На UniPascal това може да се постигне още по-лесно:

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

Ако сега опишем променливата r така: var r: real; то тя ще бъде от стандартния тип integer. Както виждате, Pascal е достатъчно гъвкав език, но всичко трябва да се използва с мярка.

3. Типове данни

Една от основните особености на езика Pascal (в сравнение с предхождащите го езици) е концепцията за типовете. По думите на самия автор на езика (Н.Уирт в "Алгоритми + Структури от данни = Програми") организацията на данните в езика Pascal се основава на теорията на структурната организация на данните на Хоар (C.A.R.Hoare), според която:

- типът определя клас от стойности (към който може да принадлежат константи или които могат да приемат променливи или изрази) и множество от операции над този клас от стойности;
- всяка стойност принадлежи на един и само на един тип;
- типът на константа, променлива или израз може да се изведе или от контекста, или от вида на самия операнд, но без да се използва стойността, изчислявана по време на работа на програмата (това е условие за статични типове данни);
- на всяка операция съответства някакъв фиксиран тип на нейните операнди и някакъв фиксиран тип на резултата й;
- за всеки тип свойствата на стойностите и елементарните операции над тях се задават с помощта на аксиоми.

При работа с език от високо ниво познаването на типа позволява на компилатора да открива в програмата безсмислени конструкции и да решава въпроса за метода, по който ще се представят данните, и за начина, по който ще се извършват преобразуванията над тях в машината.

Силната типизация (типовете отговарят на изброените по-горе условия), въведена в Pascal, е първата стъпка към абстрактните типове данни, чиято идея и реализация се появява в по-късно разработените езици (с най-известен представител ADA).

И така, двете основни характеристики на един тип са множеството от стойности, които принадлежат на този тип, и операциите, дефинирани над обектите от този тип. При описанието на всички типове данни ще се опитаме да ги представим от гледна точка на тези два основни аспекта на типа.

3.1. Прости типове данни

Към простите типове данни в Pascal се отнасят стандартните, изброимите и диапазонните типове. При определяне на синтаксиса се въвежда клас дискретни типове (ordinal types), съдържащ всички прости типове без реалния. Това се прави поради честото използване на простите типове (но без реалния) при синтактичното и семантично описание на езика.

```
SimpleType=         OrdinalType | RealType.
OrdinalType=        Enumerated | SubRange | StandardType.
StandardType=       'integer'  | 'shortint'  | 'longint' |
                    'cardinal' | 'shortcard' | 'natural' |
                    'boolean'  | 'char' |
                    'longword' | 'word' | 'byte'|.
RealType=           'real'.
```

За всички прости типове данни са дефинирани операциите (в повечето случаи няма да се спираме специално на тях, но ще предполагаме наличието им): даване на стойност (:=), отношенията за равенство (=), неравенство (<>), по-малко (<),

по-малко или равно (<=), по-голямо (>) и по-голямо или равно (>=). При отношенията типът на резултата е логически (BOOLEAN).

3.1.1. Изброим тип

Изброимият тип се дефинира чрез изброяване на всички възможни стойности, които му принадлежат.

```
Enumerated=          '(' IdentList ')'.
```

По този начин се въвеждат идентификатори за всички константи, принадлежащи на типа. За всеки изброим тип T = (W_0, W_1, ..., W_n), където W_i са идентификатори на константите на типа, е в сила:

- W_i <> W_j, ако i <> j (различимост);
- W_i < W_j, ако i < j (наредеността);
- на типа T принадлежат само константите W_0, W_1, ..., W_n;
- съществува взаимно еднозначно съответствие между множеството от стойностите на типа T и целите числа в интервала [0, n], като на W_0 съответствува 0, на W_1 - 1, ..., на W_n - n.

Преди да опишем операциите над данните от изброим тип ще дадем пример за два изброими типа (тези два типа ще считаме за валидни до края на главата, за да не ги даваме всеки път, когато е необходимо):

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

Операции над изброим тип данни:

- даване на стойност (:=). Например: today:= Monday;
- отношение. Шестте вида сравнения (=, , <, <=, >, >=) са дефинирани за всеки два обекта от изброим тип. Наредбата на константите от изброимия тип е в реда, в които те са изброени;
- получаване на пореден номер. В Pascal е дефинирана стандартната функция ORD над всеки изброим тип и даваща в резултат цяло неотрицателно число, което е и поредния номер в типа. Обърнете внимание на факта, че първата константа има пореден номер 0. Например: ORD(Black) = 0;
- получаване на стойност от изброим тип по пореден номер. В UniPascal всеки изброим тип автоматично дефинира функция над целите неотрицателни числа в изброимия тип. Името на тази функция съвпада с името на изброимия тип. Например: Color(1) = Red;
- получаване на минималната и максималната стойност на изброим тип. В UniPascal са дефинирани две стандартни функции MIN и MAX над всеки изброим тип, даващи съответно минималната и максималната стойност на типа. Например: MIN(color) = Black; Max(DayOfWeek) = Saturday;
- получаване на следващата и предишната стойност. В Pascal са дефинирани стандартните функции Succ и Pred над всеки изброим тип:

  Succ(W_i) = W_{i+1}, за всяко i = 0, 1, ..., n-1;

  Pred(Wi) = Wi-1, за всяко i = 1, 2, ..., n;

В случаите, когато полученият резултат излиза извън рамките на изброимия тип (няма съответна константа), резултатът е неопределен или се индицира грешка (в зависимост от проверката на границите, виж управление на компилацията).

3.1.2. Логически тип

Логическият тип формално може да бъде описан така:

```
type boolean = (false, true);
```

Поради това всички функции, определени за изброимия тип, са приложими и за логическия. Но освен тях над логическия тип са дефинирани и следните операции, даващи резултат отново от логически тип:

- NOT - отрицание;
- AND - конюнкция (логическо и);
- OR - дизюнкция (логическо или);
- XOR - изключващо или.

3.1.3. Целочислени типове

В UniPascal има няколко различни целочислени типа, представящи числата в различни диапазони на множеството на целите числа (за разлика от стандартния Pascal, в които е дефиниран само типа integer). Всеки два целочислени типа са съвместими помежду си в общото си сечение.

Целочислените типове са както следва:

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

Над обектите от целочислен тип са определени едноместните операции + (унарен плюс) и - (унарен минус - смяна на знака) и следните двуместни операции: събиране (+), изваждане (-), умножение (*), целочислено деление (**div**), и остатък от деление (модул) (**mod**). Ако двата операнда на операцията имат различни типове, то те първо се превръщат към минималния тип, включващ и двата диапазона, след което се извършва операцията.

Освен изброените операции, над всеки целочислен тип са дефинирани и следните стандартни функции:

- ABS(x) - абсолютна стойност на аргумента;
- SQR(x) - аргумента на квадрат (а не корен квадратен!);
- всички стандартни функции, определени над изброимия тип.

В Pascal съществува стандартната константа MAXINT. Тъй като в UniPascal има няколко целочислени типа, то е неудобно въвеждането на съответни константи за всеки тип. За целта може да се използва стандартната (в UniPascal) функция MAX. Например MAX(INTEGER), MAX(CARDINAL) и т.н.

3.1.4. Диапазонен тип

В много случай е известно, че стойностите на дадена променлива ще се менят (или по-точно трябва да се менят) само в някакви, предварително известни граници. В такъв случай е желателно това да бъде съобщено по някакъв начин на компилатора, за да може той да избере подходящо представяне и да проверява правилността на

използването на променливи от този тип. За целта в Pascal е въведен тип диапазон (подобласт) на вече дефиниран тип. Общият вид на дефиницията е:

```
SubRange=          Constant '..' Constant.
```

Всъщност всички целочислени типове могат да се разглеждат като диапазони на един единствен тип - LongInt.

Новият тип приема (наследява) всички операции, дефинирани над базовия тип (този, на който е подобласт), но принадлежащите му стойностите са в затворения интервал, определен от двете константи.

Например:

```
type WorkDay = Monday..Friday;
     Digit =   0..9;
```

3.1.5. Символен тип

Стойностите, принадлежащи на символния тип (CHAR) са символи от наличния символен набор. Символният набор може да бъде различен при различните компютри. За конкретната реализация на UniPascal това е разширеният ASCII код (8 битовия ASCII код). За 8 битовия ASCII код следните множества са наредени и заемат последователни стойности:

- арабските цифри (от 0 до 9);
- големите латински букви (от A до Z);
- малките латински букви (от a до z);
- големите букви от кирилицата (от А до Я);
- малките букви от кирилицата (от а до я).

За символния тип не са определени операции. Но за него важат всички описани за изброимия тип стандартни функции. Освен тях в Pascal е дефинирана още и стандартната функция CHR с аргумент цяло число в интервала [0, 255], даваща като резултат символ със съответния ASCII код. Между впрочем, в UniPascal необходимостта от такава функция отпада поради наличието на явно преобразуване на типа. Така че всъщност CHR(x) = CHAR(x) за всяко x (от интервала 0..255). Функцията CHR е реализирана за пълнота.

3.1.6. Реален тип

Към реалния тип (REAL) в UniPascal принадлежат (както е естествено да се очаква) реалните числа. За съжаление, поради метода на представяне на реалните числа в изчислителните машини изобщо, това не са познатите ни от математиката реални числа, а реални числа, представяни с крайна точност, в частност трансцедентните числа и безкрайните периодични дроби не могат да се представят точно в компютърната реална аритметика.

Реалните числа се представят закръглено с точност, която специално за UniPascal е 7-8 цифри за мантисата, а порядъка е в интервала [-38, 38], т.е. могат да бъдат представяни реални числа със 7-8 цифри, умножени по 10 на степен от - 38 до 10 на степен 38. Трябва винаги да се има предвид тази особеност на компютърната реална аритметика, когато се съставят програми.

Над реалния тип са дефинирани едноместните операции унарен плюс и унарен минус (смяна на знака), както и следните двуместни операции: събиране (+), изваждане (-), умножение (*), деление(/).

Дефинирани са и следните стандартни функции на реален аргумент, на които резултатът е от реален тип:

- ABS(x) - абсолютна стойност на аргумента;
- SQR(x) - 2-ра степен на аргумента.

Следните функции могат да имат аргумент както от реален, така и от цял тип, но резултат им винаги е реален:

- SQRT(x) - корен квадратен от аргумента;
- LN(x) - натурален логаритъм от аргумента;
- EXP(x) - експонента от аргумента;
- SIN(x) - синус от аргумента (даден в радиани);
- COS(x) - косинус от аргумента (даден в радиани);
- ARCTAN(x) - аркустангенс в радиани от аргумента;
- INT(x) - цялата част на аргумента;
- FRAC(x) - дробната част на аргумента.

Ако един от операндите на двуместна операция е реално а другият е цяло число, то цялото число първо се превръща в реално и след това се извършва операцията. Изобщо във всички случаи, когато се очаква реално число (константа или променлива), а е използвано цяло, то цялото се превръща в реално, преди да се извърши действието. Затова е възможно на променлива от реален тип да се дава целочислена стойност (преобразуването се прави автоматично). Обратното превръщане не се прави автоматично. За целта в Pascal има дефинирани две стандартни функции, превръщащи реално в цяло число:

- TRUNC(x) - превръща реалното число **x** в цяло чрез отрязване на дробната част;
- ROUND(X) - превръща реалното число **x** в цяло чрез закръгляване.

Ако стойността на аргумента x във функциите TRUNC и ROUND е извън целочисления интервал, определен от LongInt, то по време на изпълнение на програмата се съобщава за настъпила грешка.

3.1.7. Стандартните типове BYTE, WORD, LONGWORD

В UniPascal са въведени следните стандартни типове: BYTE, WORD, LONGWORD. Те не са целочислени типове, а са аналог на подобни типове от езика Modula-2. Тези три типа са напълно съвместими с типовете, които имат дължина съответно 1, 2 и 4 байта. Над тези типове са дефинирани следните операции:

- AND - побитово (byte($c3) and byte($66) = byte($42));
- OR - побитово (byte($a5) or byte($5a) = byte($ff));
- XOR - побитово (byte($c3) xor byte($66) = byte($a5));
- = - сравнение за равенство;
- <> - сравнение за неравенство;
- := - даване на стойност.

Други операции над тези типове не са дефинирани и при опит за използване се индицира грешка по време на компилация. Ако се използват променливи от тези типове в стандартната процедура write (за текстов файл), стойността им се отпечатва в шестнадесетична бройна система. Не е разрешено използването на променливи от тези типове в стандартната процедурата read. Ако е наложително, използвайте преопределяне на типа (Type Cast).

3.2. Съставни (структурирани) типове данни

В Pascal на основата на простите типове потребителят има възможност да конструира свои собствени типове. Такива типове се наричат структурирани. Структурираният тип се характеризира чрез метода на структурирането си и чрез типа на своите компоненти. Ако компонентите на структурирания тип са също структурирани, то тогава става въпрос за повече от едно ниво на структуриране. Видовете структурирани типове в Pascal са: масиви, записи, множества, файлове. Типът низ (StringType) е дефиниран в UniPascal, но не и в Pascal.

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

Пред описанието на всеки структуриран тип може да се постави префикса '**packed**', който указва на компилатора да икономисва памет даже за сметка на по-неефективен достъп до елементите на структурата.

Както и при простите типове, така и тук операцията даване на стойност е дефинирана за всеки структурен тип, но има едно изключение - за променливи от файлов тип не е разрешено даване на стойност.

3.2.1. Масиви

Масивът е хомогенна структура. Състои се от компоненти от един и същ тип, наречен базов. При Pascal е съществено, че броят на елементите се указва в описанието и остава непроменен през цялото време на своето съществуване. За означаване на отделна компонента на масив, името на цялата структура се разширява чрез така наречения индекс. Съществена особеност при Pascal е, че типът на индекса не е строго фиксиран (както във FORTRAN или BASIC например), а може да бъде зададен от програмиста. Единственото ограничение е типът на индекса да бъде дискретен (затова REAL не е разрешен). Разбира се, съвсем очевидно ограничение (налагано от реализацията) е структурата да се побира в паметта на компютъра. Общият вид на описанието на структурата масив е:

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

IndexType=          OrdinalType.
```

Базов тип на структурата масив може да бъде всеки тип с изключение на файл. В частност базов тип на масива може да бъде също масив. В такъв случай цялата структура се нарича многомерен масив. За краткост описанието на многомерни масиви може да се записва, като се разделят индексите чрез запетая. Но определението на масив от масиви е също възможно и, нещо повече, то е еквивалентно на краткото описание. Например, следните две определения на масиви са еквивалентни:

```
type Array1 = array [1..10] of array [boolean] of integer;
type Array2 = array [1..10, boolean] of integer;
```

Обозначаването на отделен елемент от масива (индексирането) се прави, като след името на структурата в квадратни скоби се запише номера на елемента, който може да бъде не само константа, но и променлива или израз. И при индексирането е

разрешено използването на кратката форма за многомерни масиви, но тя е напълно еквивалентна на разширената. Например: Array1[i][b] или Array1[i, b].

Освен даването на стойност, над масивите е дефинирана и операцията сравнение за равенство (=) и различие(<>). Операцията за сравнение за равенство дава като резултат true само, ако елементите на двата масива съвпадат поелементно, и false в противен случай. Операцията за сравнение за неравенство може да се определи като отрицание на равенството.

В Pascal едномерните пакетирани масиви с базов тип char са привилегировани. При тях може да се използва пълният набор от операции за сравнение, т.е. <, <=, >, >=. В тези случаи сравнението се прави по наредбата на символите в символния набор (кодовата таблица).

3.2.2. Стандартен тип STRING

В UniPascal има още една структура, много приличаща на едномерния масив. Това е структурата символен низ (string). Неговия индексен тип е диапазон на типа SHORTCARD, като долната граница е фиксирана на 1, а горната може да бъде зададена от потребителя, но не трябва да превишава 255. При структурата string дължината се изменя динамично, но в границите от 0 до зададената горна граница. Общия вид на описанието е:

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

Ако не е зададена горна граница, се подразбира максималната възможна - 255, т.е.

```
var t: string;
```

е еквивалентно на

```
var t: string[255];
```

3.2.3. Записи

Записите (Record) в Pascal са структури от данни, състоящи се от фиксирано число именувани компоненти, наречени полета. Тези полета могат да бъдат от различен тип. При определянето на запис за всяко поле се задава неговото име (идентификатор) и тип. Областта на действие на този идентификатор на полето е в рамките на записа. Най-общият вид на типа запис е:

```
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.
```

При използването на полетата на записа в програмата, те се именуват като след името на цялата структура се постави точка, последвана от името на съответното поле.

Понякога е необходимо да се опише структура, в която в зависимост от състоянието на някое поле, друго нейно поле има различен тип (смисъл). За такива случаи в Pascal е предвидена вариантна част на записа. Например, ако описваме типа ЧОВЕК, в променлива от този тип може да държим информация за неговото име, дата на раждане, пол и т.н. В случаите, когато това е мъж, е необходимо да пазим някаква информация, свързана само с мъжете - например какъв чин има (или е имал) през военната му служба, а ако е жена - информация само за жените или пък никаква друга информация.

```
type SexType = (male, female);
date = record
  day: 1..31;
  month: 1..12;
  year: cardinal;
end; { date }
Person = record
  name: string[31];                    { името на човека }
  birthday: date;                      { дата на раждане }
  case sex: SexType of                 { пол - мъж, жена }
     male: (rank: (soldier, sergeant, colonel));
  female: ();
end; { Person }
```

При описанието на записите трябва да се има в предвид следното:

- всички имена на полета трябва да бъдат различни, даже когато се срещат в различни варианти на един и същи запис;
- ако за някоя стойност не съществува вариант, той може да не бъде указан, т.е. не всички стойности от типа за признак трябва да бъдат изредени;
- всеки списък от полета може да има само една вариантна част, която трябва да следва фиксираната част;
- във всяка вариантна част може да има вложена друга вариантна част, като ограничение за нивата на влагане няма;
- ако някое от полетата има анонимен изброим тип, то областта на действие на неговите константи се разширява върху блока, в който е дефиниран записът. Анонимен се нарича тип, чието описание се намира в декларацията на променлива или поле от запис. Такъв тип няма име и затова се нарича анонимен. Например: var device: (disk, tape, printer);

Над обектите от тип запис са дефинирани операциите даване на стойност (:=) и операциите сравнение за равенство (=) и неравенство (<>).

3.2.4. Множества

Следващата след масива и записа основна структура от данни е структурата множество. Общия вид на нейната дефиниция е:

```
SetType=            'set' 'of' OrdinalType.
```

Множеството от стойности, дефинирано от структурата множество, е множество от всички подмножества (включително и празното) на множеството от елементите на базовия тип. С други думи, елемент от структурата множество е множество от елементи на базовия тип.

В UniPascal се налага следното ограничение: броят на елементите в базовия тип не трябва да бъде по-голям от 256. Ако базовият тип е числов, то минималният елемент

не може да бъде по-малък от нула, а максималният - по-голям от 255. Ако базовият тип е изброим, то той не трябва да е с повече от 256 елемента.

Над обектите от множествен тип в Pascal са дефинирани следните операции:

- даване на стойност :=
- сравнение за равенство =
- сравнение за неравенство
- включване на множества >=, <=
- обединение на множества +
- сечение на множества *
- разлика на множества -
- принадлежност към множество IN

Освен тези операции в UniPascal са дефинирани и следните две стандартни процедури (заимствани от Modula-2): Incl(s, x) и Excl(s, x), където s е променлива от тип множество, а x е елемент от базовия тип на множеството. Тези процедури съответно включват или изключват зададения като параметър елемент от множеството s и са еквивалентни съответно на s:= s + [x] и s:= s - [x]. Множеството не се променя при включване на елемент, който вече се съдържа в него, както и ако се изключва елемент, който не се съдържа в множеството.

В програмата множествени константи се създават чрез така наречения конструктор на множества. В него елементите на множеството се разделят със запетаи и се затварят в квадратни скоби. Празното множество се обозначава като [ ]. Най-общия вид на конструктора е:

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

Празното множество (отбелязвано като [ ]) принадлежи на всички множествени типове. Обозначението X..Y, където X и Y са изрази, дефинират множество, състоящо се от всички елементи от базовия тип от затворения интервал [X, Y]. Ако X > Y, тогава [X..Y] е празно множество. Когато стойността на елемент при конструирането на множество излиза извън възможностите на реализацията (поредният номер на елемента е извън интервала [0, 255]), се получава грешка и за нея се съобщава или по време на компилация (ако този елемент е константа) или по време на изпълнение (ако е променлива).

3.2.5. Файлове

Файловете са една от най-важните структури във всеки език за програмиране. Всяка програма, работеща на даден компютър, трябва да комуникира с обкръжаващата я среда (диск, екран, клавиатура и т.н.) само с помощта на файлове. Ако стартирате програма, която не използва нито един файл (т.е. нито чете някакви входни данни, нито записва някъде резултата от работата си), то ефектът от нейното изпълнение ще е нулев, т.е. все едно, че изобщо не е стартирана.

В Pascal под файл се разбира само файл с последователен достъп, т.е. във всеки момент е достъпна една компонента от него, след прочитането на която е възможен достъп само към следващата компонента. В UniPascal файловете (с изключение на текстовите) могат да бъдат използвани и с директен достъп, т.е. може да се избира коя компонента от файла да бъде прочетена.

Общият вид на типа файл е:

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

В Pascal не се налагат никакви ограничения върху типа на компонентите, но в UniPascal не се разрешава типът на компонентите да е файлов. Освен това не трябва типът на компонентата да е от тип указател или структура, съдържаща указатели. Въпреки че такава проверка не се прави, крайно опасно е да се използват такива файлове, защото при прочитане от този файл на указателя, то той вероятно ще има неправилна стойност.

В UniPascal е възможно да не се укаже типът на компонентите на файла (като се изпусне ключовата дума 'of' заедно с типа на компонентите). Такива файлове се наричат нетипизирани и служат за осъществяване на специален достъп към външен файл.

Мнозинството реализации се различават в работата с файлове, тъй като в стандарта на Pascal не са дефинирани достатъчно процедури за работа с файлове, както и неяснотата на връзката между променливата, обозначаваща файл, и реално съществуващите в операционната система файлове.

В UniPascal са дефинирани следните операции с файлове:

- RESET - отваряне на съществуващ файл за четене;
- REWRITE - създаване на нов файл (отваряне за запис);
- APPEND - отваряне на съществуващ файл (или създаване на такъв, ако не съществува) за запис в края му (добавяне към файла);
- OPEN - отваряне на файл за едновременно четене и запис (обновяване), не се разрешава използването й с текстови файлове;
- CLOSE - затваряне на файл, прекъсва връзката между файловата променлива и реално съществуващия файл;
- EOF - проверка за край на файл;
- EOLN - проверка за край на ред, разрешена е само за текстови файлове;
- READ - прочитане на една или няколко компоненти, не се разрешава използването й с нетипизирани файлове;
- WRITE - записване на една или няколко компоненти, не се разрешава използването й с нетипизирани файлове;
- READLN - прочитане на една или няколко компоненти с последващо пропускане на реда, използва се само с текстови файлове;
- WRITELN - записване на една или няколко компоненти с последващо преминаване на нов ред, използва се само с текстови файлове;
- SEEK - позициониране във файл, не се разрешава използването й с текстови файлове;
- FILESIZE - получаване на размера на файла, не се разрешава използването й с текстови файлове;
- FILEPOS - получаване на номера на текущата компонента на файла, не се разрешава използването й с текстови файлове;
- BLOCKREAD - четене от нетипизиран файл;
- BLOCKWRITE - запис в нетипизиран файл.

По-подробно описание за всяка една от тях можете да намерите в приложението, описващо стандартните процедури и функции.

Важна разлика между UniPascal и стандартния Pascal е, че в UniPascal не съществуват стандартните процедури GET и PUT, както и буферна променлива, свързана с файла.

3.2.5.1. Текстови файлове

Най-често използвани са файловете, чиито компоненти са символи (знаци). Това е всъщност и най-удобната форма за нас хората. В Pascal е дефиниран стандартен тип TEXT, който представлява такъв файл. Неговото описание е:

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

Текстовите файлове освен че са съставени от символи, се делят и на редове. За по-добро отразяване на такава структура на текстовия файл са дефинирани процедурите ReadLn и WriteLn, съответно преминаващи към четене от и запис в следващия ред, както и функцията Eoln, която проверява дали е достигнат краят на реда при четене.

Въпреки че компонентите на текстовия файл са само от тип char, те (текстовите файлове) са привилегировани в сравнение с другите файлове и в тях може да записват (или четат) променливи от всички стандартни типове (с изключение на типа BOOLEAN). Преди да се извърши записът в текстов файл, всяка променлива се превръща (неявно) от своето вътрешно представяне в символен вид и тогава се записва във файла. Аналогично, при четене първо се превръща символното представяне във вътрешно и след това се записва в променливата.

3.2.5.2. Стандартни файлове

В UniPascal са дефинирани следните пет стандартни текстови файла:

- Input - стандартен входен файл, обикновено е клавиатурата на компютъра, но може да бъде пренасочен от операционната система към външен файл;
- Output - стандартен изходен файл, обикновено е екранът на компютъра, но също може да бъде пренасочен;
- Message - входно/изходен файл, винаги работи с клавиатурата или екрана на компютъра и не може да бъде пренасочен;
- Auxiliary - входно/изходен файл (в зависимост от операционната система: най-често сериен вход и/или изход);
- Printer - изходен файл, обикновено е печатащо устройство, но понеже е свързан с хардуерен модул за паралелен вход/изход може да се използва и за входен файл.

За тези пет стандартни файла не е необходимо да се използват процедурите RESET, REWRITE или CLOSE, нещо повече - дори и да се използват, те няма да имат ефект.

В заглавната част на програмата могат да бъдат указани няколко от тези файлове (но само от тези пет) като параметри на програмата. Ако не са указани параметри, се подразбират файловете Input и Output. За файловете Input и Output е възможно при стартиране на програмата да бъдат пренасочени и тогава тя няма да чете от клавиатурата и/или да пише на екрана, а ще извършва операциите над зададените при стартирането й файлове. Понякога е необходимо някои от съобщенията да не бъдат пренасочвани, както и четенето да се извършва непременно от клавиатурата (например при съобщаване на някаква грешка на потребителя на програмата и получаването на указания какво да прави тя по-нататък). В такъв случай може явно да се укаже, че е необходимо да се използва файла Message. От друга страна може би програмата ще трябва да работи винаги с файла Message и само рядко да се използва Input и Output. Трябва ли всеки път да се указва явно използването на файла message във всеки READ и WRITE? В UniPascal е реализирана възможността да се укаже кой именно файл се подразбира като параметър на стандартните функции за работа с текстови файлове, когато бъде пропуснат.

Задаването на параметри на програмата чрез изброяване на външните файлове има съществено значение за програмата. По-точно, от особено значение е редът на изброяването на тези файлове. Когато се използва някоя от конструкциите READ, READLN, ... без да се указва файл, се подразбира използването на първия от изброените външни файлове с необходимия достъп (за четене или запис). Например, нека разгледаме следната програма:

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

Тази програма прочита един ред от стандартния входен файл Input и след това го записва в стандартния изходен файл Output. Ако заглавната част на програмата е:

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

то процедурата READLN ще използва по подразбиране файла Input, но в WRITELN ще се използва файла Message. А ако е:

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

то READLN и WRITELN ще използват файла Message, защото той удовлетворява и двата типа достъп (за четене и запис) и е зададен първи в списъка на външните файлове. Задаването на други файлове след Message няма да повлияе върху програмата, но е признак на добър стил, ако в заглавната част се изброяват всички външни файлове, които ще бъдат използвани. От тази гледна точка задаването на файла Input в заглавната част на програмата Example3, не само е ненужно, но и не е хубаво, защото този файл въобще не се използва в програмата.

И още една разлика между файловете Output и Message. Ако се записват символи, излизащи извън дясната граница на реда, то при файла Output се преминава автоматично на нов ред, а при файла Message не, т.е. символите се пишат един върху друг в последната позиция на реда на екрана. За повече подробности за работа със стандартните файлове Input и Output виж приложение "Сведения за клавиатурата и екрана на микрокомпютъра Пълдин".

3.2.6. Пакетиране в UniPascal

Както беше споменато, в Pascal описанието на всеки структуриран тип може да бъде предшествувано от префикса packed. Това означава, че при вътрешното представяне ще се използва пакетиране, за да се пести памет за данните. За съжаление, такова представяне може да доведе в някои случаи до по-големи разходи на памет поради усложнения достъп до такива пакетирани компоненти, т.е. ще се спести памет за данни, но ще се изразходва памет за код на програмата.

Затова в UniPascal е възприето компромисно решение. Пакетирането се прави на граница на байт, като по този начин са необходими значително по-прости средства за поддръжката на пакетираните компоненти.

При използването на пакетирани типове трябва да се има предвид това, че техните компоненти не могат да бъдат предавани като VAR параметри (по адрес) (виж главата за описание на процедури и функции - раздел параметри).

3.3. Динамични структури от данни

При досега разгледаните структури от данни компилаторът отделя място в паметта за тяхното представяне по време на компилация. Тези структури често се наричат статични. Разликата между тях и динамичните структури не е главно в това, че паметта за динамичните структури се отделя по време на изпълнение, а в това, че те (динамичните структури) обикновено се използват за представяне на рекурсивни структури от данни, т.е. на структура която включва себе си (явно или неявно) в своето определение. По тази причина големината на такава структура е неизвестна по време на компилация и даже нещо повече - тя обикновено се мени по време на изпълнение (рекурсивността на динамичните структури не е задължителна, но тя е най-съществения различие между двата типа структури от данни).

В Pascal динамичните структури от данни са реализирани чрез указатели. Общият вид на описанието на типа указател е:

```
PointerType=        '^' TypeIdent.
```

Идентификаторът на типа, към който сочи указателният тип (този тип се нарича базов тип за указателния), може да бъде дефиниран след дефиницията на указателния тип, но в рамките (т.е. до края) на текущия раздел за дефиниране на типове.

Множеството от стойности на всеки указателен тип се създава динамично по време на изпълнение, но за всички указатели е дефинирана стойността NIL, която се интерпретира като указател, който не сочи към никакъв елемент. В UniPascal е дефиниран стандартен тип Pointer, съвместим със всички указателни типове. Неговите стойности се интерпретират като указатели, сочещи към обекти без тип.

Над указателите са дефинирани операциите: даване на стойност (:=), сравнение за равенство (=) и различие (<>) и следните стандартни процедури: NEW - за създаване на нов обект и DISPOSE - за унищожаване на съществуващ обект. Дефинирани са и няколко процедури за работа с динамичната памет (паметта, която се използува за разполагане на динамичните променливи): GETMEMWORDS, FREEMEMWORDS, MARK и RELEASE.

Процедурите MARK и RELEASE са предвидени като заместители на процедурата DISPOSE в някои реализации на Pascal, но те се оказват удобни за работа и затова са реализирани и тук. Кой от двата метода: MARK-RELEASE или DISPOSE, ще бъде използван за освобождаване на заета памет е въпрос на конкретна необходимост и организацията на данните в програмата. Използуването и на двата метода в една и съща програма не е препоръчително и може да доведе, в най-лошия случай, до неосвобождаване на част от заетата памет (за подробности виж UniPascal в детайли).

3.4. Идентичност и съвместимост на типовете

При съставянето на програми е необходимо да се знае какви са отношенията на променливи, принадлежащи към различни типове. Досега винаги подчертавахме, че над даден тип са приложими някакви операции. Но не споменавахме дали може да имаме смесване на променливи от различни типове. За програмистите обикновено е интуитивно ясно кои типове могат да бъдат смесвани (т.е. са съвместими) и кои не. За компилатора, от друга страна, е строго дефинирана тази съвместимост.

В Pascal, както беше споменато по-рано, е в сила силната типизация, но тя има две основни свои разновидности - структурна и именна. Най-общо казано, при структурната два типа са съвместими, ако имат еквивалентна структура, а при именната няма съвместимост между два различни типа, т.е. при именна типизация всеки два различни типа са несъвместими. В по-ранните реализации на Pascal поради липса на

стандарт всяка от тях решаваше сама въпроса към кой вид силна типизация да принадлежи. В ISO стандарта на Pascal (към който се придържа и UniPascal) типизацията е нещо средно между двата вида с превес към именната.

3.4.1. Идентичност на типовете

Идентичност на типовете (type identity) се изисква само за съвместимост между фактическите и формалните параметри-променливи на процедурите и функциите. Самото название ни подсказва, че ако два типа са идентични, то те са неотличими един от друг.

Два типа T1 и T2 са идентични, ако е изпълнено едно от следните условия:

- T1 и T2 са един и същ тип (например типа integer е идентичен със себе си);
- ако T2 е определен чрез декларацията type T2 = T1; (или обратното, т.е. T1 е определен така).

Освен тези правила е в сила и:

- типът T1 е идентичен със себе си;
- ако T1 е идентичен с T2, то T2 е идентичен с T1;
- ако T1 е идентичен с T2 и T2 е идентичен с T3, то T1 е идентичен с T3 (транзитивност, T1 = T2 = T3 => T1 = T3).

При именната силна типизация е в сила единствено твърдението, че типът T е идентичен само със себе си и с никой друг тип. Дори ако типът newT е определен като newT = T, то тези два типа не са идентични при именната типизация.

Забележка: При предаване на параметри от тип низ (STRING) не е необходимо формалният и фактическият параметри да имат идентични типове, достатъчно е фактическият параметър да има не по-малка максимална дължина от формалния.

3.4.2. Съвместимост на типовете

Съвместимост на типовете (type compatibility) е необходима при смесено използване на променливи от различен тип в изрази (аритметични и сравнения).

Два типа са съвместими, ако е изпълнено поне едно от следните условия:

- двата типа са идентични;
- двата типа са целочислени;
- единият тип е диапазон на другия;
- двата типа са диапазони на един и същ (изходен) тип и имат общо сечение на множеството от стойности;
- двата типа са множества със съвместими базови типове;
- двата типа са пакетирани масиви от символи (char) с еднакъв брой компоненти;
- единият тип е низ, а другият е или низ или стандартен тип char;
- единият тип е стандартен тип pointer, а другият е какъвто и да е указателен тип;
- единият тип е стандартен тип byte (word или longword), а другият тип е какъвто и да е тип, който се реализира с 1 (2 или 4) байт.

Забележка: под низ се разбира стандартният тип string без ограничение за количеството елементи, т.е. string[7] и string[77] са съвместими типове, както и string[5] е съвместим с типа char.

3.4.3. Съвместимост за даване на стойност

Съвместимост за даване (assignment compatibility) е необходима при даване на стойност на променлива или при предаване по стойност на фактически параметър на процедура или функция.

Стойност от тип T2 е съвместима като стойност с променлива (параметър) от тип T1, ако е изпълнено едно от следните условия:

- T1 и T2 са идентични типове;
- T1 и T2 са съвместими дискретни типове и стойността от тип T2 е в диапазона от допустими стойности на типа T1;
- T1 е реален тип, а T2 е целочислен тип;
- T1 и T2 са низове (важи забележката от по-горе);
- T1 е низ а T2 е от типа char (важи същата забележка);
- T1 и T2 са съвместими пакетирани масиви с елементи от стандартния тип char;
- T1 и T2 са съвместими множествени типове, като всички елементи, принадлежащи на множеството от тип T2, са в диапазона на елементите, които могат да принадлежат на T1;
- T1 и T2 са съвместими указателни типове;
- единият тип е стандартен тип byte (word или longword), а другият тип е какъвто и да е тип, който се реализира с 1 (2 или 4) байт.

4. Променливи

Променливите притежават тип, определен от тяхното описание и могат да приемат стойности, принадлежащи на множеството стойности на този тип. За всяка статична променлива, описана в някой блок, се отделя място в паметта при активация на блока и се унищожава при завършване на работата му. За динамичните променливи се отделя място чрез стандартната процедура NEW и се унищожават - чрез DISPOSE.

Обръщението към променлива означава едно от следните неща:

- пълна променлива;
- компонента-променлива на съставен (масив или запис) тип;
- динамична променлива.

5. Изрази

Изразът, най-общо казано, е правило за пресмятане на стойност на базата на своите компоненти, наречени операнди. Действията, прилагани над тези компоненти, са операциите. Типът на получения резултат зависи от типа на операндите и от операциите, изграждащи израза. Този тип е еднозначно определен и известен още по време на компилация.

```
Expression=         (SimpleExpression [relationOp
                     SimpleExpression]) |
                    ExpTypeCast.
SimpleExpression=   ['+' | '-'] Term {AdditiveOp Term}.
Term=               Factor {MultiplicativeOp Factor}.
Factor=             Constant       |
                    VariableRef    |
                    SetConstructor |
                    FunctionCall   |
                    'not' Factor   |
                    '(' Expression ')'.
```

5.1. Операнди

Операндите могат да бъдат константи, променливи, обръщения към функции и конструктори на множества. Обръщението към функция

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

може да бъде както към стандартна функция, така и към функция, дефинирана от потребителя. Функциите със странични ефекти са нежелателни, тъй като операндите могат да бъдат пресметнати в друг, различен от написания, ред.

Използването на променливи, които не са получили стойност до началото на пресмятането на израза, е грешка. Ако тази грешка не довежда до други грешки, то тя не може да бъде установена нито по време на компилация, нито по време на изпълнение. Такава грешка най-често довежда до неопределена стойност на резултата.

При пресмятането на аритметични изрази е възможно да възникне и един друг вид грешка, която си заслужава да отбележим. Става въпрос за препълването. То възниква, когато резултатът от аритметична операция излиза извън диапазона на допустимите за него стойности. Когато резултатният тип е целочислен, такава грешка не винаги може да бъде установена по време на изпълнение.

5.2. Операции

Операциите в Pascal са три типа: мултипликативни, адитивни и операции за отношения.

```
relationOp=         '=' | '<>' | '<' | '<=' | '>' | '>=' | 'in'.
AdditiveOp=         '+' | '-' | 'or' | 'xor' | '|'.
MultiplicativeOp=   '*' | '/' | 'div' | 'mod' | 'and' | '&'.
```

Когато редът на операциите не е явно зададен чрез скоби, техният приоритет е следния:

1) логическо отрицание (not)
2) мултипликативни операции (*, /, div, mod, and, &)
3) адитивни операции (+, -, or, xor, |)
4) операции за отношения (=, <>, <, <=, >, >=, in)

Когато приоритетите съвпадат, пресмятането се извършва от ляво на дясно.

Необходимо е да се подчертае, че редът на пресмятането на операндите не е фиксиран, т.е. възможно е първо да се пресмята десният (а не левият) операнд на дадена операция. Изключение се прави само в случая за логическите операции and и or, при които първо се пресмята левият операнд и, ако от неговата стойност може предварително да се установи крайният резултат, десният операнд не се пресмята.

5.2.1. Аритметични операции

Аритметичните операции са операции над целочислени или реални числа, като получаваният резултат е целочислен или реален. Видът на получения резултат (реален или целочислен) зависи от операндите. Ако поне един от двата операнда е реален, то и другият операнд се превръща в реален преди извършване на операцията. Единствената операция, която връща реален резултат независимо от типа на операндите си е реалното деление ('/'). При него и двата операнда се превръщат в реални числа преди извършване на операцията. Във всички останали случаи (т.е. и двата операнда са целочислени) се извършват целочислени операции. Тъй като в UniPascal има няколко целочислени типа, компилаторът превръща преди изпълнение на операцията единия или и двата операнда в този целочислен тип, чиито диапазон обхваща диапазоните и на двата операнда, както и на резултата, ако неговият тип е известен.

Унарни аритметични операции

Операция	Действие	Тип на операнда	Тип на резултата
+	тъждествено	Цял или Real	Цял или Real
-	смяна на знака	Цял или Real	Цял или Real

Бинарни аритметични операции

Операция	Действие	Тип на операндите	Тип на резултата
*	умножение	Цял или Real	Цял или Real
div	деление	Цял	Цял
mod	остатък	Цял	Цял
/	деление	Цял или Real	Real
+	събиране	Цял или Real	Цял или Real
-	изваждане	Цял или Real	Цял или Real

5.2.2. Логически операции

Логическите операции се извършат само над операнди от логически тип и техният резултат е също от логически тип. В UniPascal пресмятането на логическите изрази се прекратява веднага, след като стане възможно, т.е. в някой случай няма да се пресмятат и двата операнда на операциите AND или OR. Това е възможно в следните случаи:

- операцията е AND, левият операнд е false и, следователно, резултатът е false независимо от стойността на десния операнд;
- операцията е OR, левият операнд е true и, следователно, резултатът е true независимо от стойността на десния операнд.

В UniPascal са дефинирани и алтернативни логически операции **&** и **|**, съответни на операциите AND и OR. За разлика от основните, алтернативните операции изчисляват и двата си аргумента.

Пълният списък на логическите операции в UniPascal е:

- NOT - логическо отрицание, унарна логическа операция;
- AND - логическо 'И' (конюнкция);
- & - логическо 'И' (конюнкция);
- OR - логическо 'ИЛИ' (дизюнкция);
- | - логическо 'ИЛИ' (дизюнкция);
- XOR - изключващо 'ИЛИ' (сума по модул 2).

В някои реализации (но не и в стандарта на езика) е разрешено използването на логически операции над целочислени типове. В този случай те означават побитово изпълнение на операцията над битовете на двата операнда. В UniPascal не е разрешено използването на логическите функции над целочислените типове, но е разрешено над стандартните типове BYTE, WORD, LONGWORD.

5.2.3. Операции над множества

Операциите над множествата изискват и двата операнда да бъдат от съвместим множествен тип. Получаваният резултат е от същия тип множество, от които са и двата операнда, или от обхващащ и двата операнда множествен тип.

- обединение на множества +
- разлика на множества -
- сечение на множества *
- включване на лявото множество в дясното <=
- включване на дясното множество в лявото >=
- равенство на множества =
- различие на множества
- проверка за включване на елемент в множество IN

5.2.4. Операции за сравнение

Операциите за сравнение включват широк спектър от операнди, но резултатът е винаги от логически тип (Boolean).

= сравнение за равенство, операцията е приложима над всички съвместими типове с изключение на файловите;

сравнение за различие, операцията е приложима над всички съвместими типове с изключение на файловите;

< сравнение за по-малко, операцията е приложима над всички прости типове, стандартния тип string и пакетиран масив от символи (packed array of char);

<= сравнение за по-малко или равно, операцията е приложима над всички прости типове, стандартния тип string и пакетиран масив от символи (packed array of char). Операцията може да се прилага и над множествения тип, като в този случай тя се интерпретира като включване на лявото множество в дясното;

> сравнение за по-голямо, операцията е приложима над всички прости типове, стандартния тип string и пакетиран масив от символи (packed array of char);

>= сравнение за по-голямо или равно, операцията е приложима над всички прости типове, стандартния тип string и пакетиран масив от символи (packed array of char). Операцията може да се прилага и над множествения тип, като в

този случай тя се интерпретира като включване на дясното множество в лявото;

IN включване на елемент в множество. Левият операнд е от дискретен (ordinal) тип, а десният операнд е множество със същия (или поддиапазон на същия) базов тип. Операцията дава резултат TRUE, ако левият операнд принадлежи на множеството и FALSE в противен случай.

5.3. Преопределяне на типа (Type Cast)

Преопределянето на типа е разширение на стандарта на Pascal. В UniPascal идентификаторът на всеки тип може да се използва като идентификатор на функция с единствен параметър. Тази функция връща стойността на параметъра си, но той има тип, чийто идентификатор е използван. Преопределянето е явно указание за промяна на типа и се използва, за да може да се смени типа на променлива или цял израз (неявна промяна на типа се извършва при преобразуването на целочислен в реален тип или между различните целочислени типове).

```
ExpTypeCast=        TypeIdent '(' Expression ')'.
VarTypeCast=        TypeIdent '(' VariableRef ')'.
```

Преопределянето на тип е два вида:

- преопределяне на типа на променлива. То се използва за да се смени типът на променлива. Размерът на променливата и на типа, с които тя се преопределя, трябва да съвпадат или променливата трябва да бъде нетипизиран параметър. За този вид преопределяне на тип компилаторът не генерира код, а на това място (и само на това) променливата се счита, че е декларирана от типа, с който се преопределя;
- преопределяне на типа на стойност. То се използва за явно указване за преобразуване на типа. При този вид преопределяне е възможна генерация на код за превръщане от единия вид към другия.

Синтаксисът за преопределяне на типа е един и същ и за двата случая. Компилаторът решава кой от тях е в сила в зависимост от контекста.

Като пример ще разгледаме използването на побитови логически операции над операнди от целочислени типове. Тъй като тези операции са дефинирани само над типовете BYTE, WORD и LONGWORD, ще използваме необходимия ни тип за отделните случай. Нека имаме определенията:

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

Тогава следните оператори са валидни и правилни:

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

Следният оператор е синтактично и семантично верен, но неправилен (т.е. алгоритмично е неверен):

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

Тъй като в случая е в сила преопределяне на типа на израз, то l1 и l2 първо ще бъдат превърнати в WORD (чрез отрязване на старшата дума) и едва тогава ще бъде приложена операцията AND над думи, т.е. горният оператор е еквивалентен на:

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

Друг случай, когато се използва преопределянето на типа на израз, е явно да се укаже целочислените операции да се извършват над типа LongInt. Например i1 * i1 е различно от longint(i1 * i2), ако i1 = 300, i2 = 1000. Но това в повечето случаи не се налага, тъй като компилаторът сам определя как да прави пресмятанията в зависимост от очаквания тип. Под очакван тип се има предвид следното:

- ако левият операнд в израз е двойна дума (LONGINT), пресмятането на десния операнд се извършва с LONGINT аритметични операции;
- операторът за даване е аналогичен на операция; ако типа на променливата, на която се дава стойност, е LONGINT, пресмятанията се извършват с LONGINT аритметика;
- предаването по стойност на фактически параметри на процедури (функции) е аналогично на оператор за даване; ако формалният параметър е от тип LONGINT, пресмятанията се извършват с LONGINT аритметика;
- ако двата операнда са целочислени и минималният диапазон, включващ диапазоните и на двата операнда, принадлежи на типа LONGINT, то пресмятането се извършва с LONGINT аритметика;
- във всички други случаи, ако не е използвано явно преопределяне на типа (Type Cast), целочислените операции се извършват чрез INTEGER или CARDINAL аритметични операции.

Ясно е, че в някои случаи компилаторът няма да може да вземе сам правилно решение какъв тип операции трябва да бъдат използвани. Например, ако i = 10, то операторът

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

при i1 = 300 и i2 = 1000 ще даде грешен резултат, тъй като пресмятанията ще се извършват чрез INTEGER аритметика. За целта трябва явно да се укаже преобразуването на типа. Докато оператора

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

ще даде верен резултат, тъй като очакван тип е LONGINT.

6. Оператори

Операторите са градивните елементи, чрез които се описва алгоритъма за обработка на данните. Пред всеки оператор може да бъде поставен етикет, който се използва за безусловно предаване на управлението чрез оператора за преход. Операторите във всеки блок се обединяват в общ раздел, наречен раздел за оператори.

Общият вид на операторите е:

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

Label е етикет (идентификатор или цяло число), описан в секцията за етикети. Операторите в Pascal се делят на прости (Simple) и съставни или структурирани (Structured). Символът ; (точка и запетая) не е част от оператора, а е разделител между операторите. Това трябва добре да се запомни, за да не се допускат грешки.

6.1. Прости оператори

Прости се наричат тези оператори, които не включват в себе си други оператори на Pascal.

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

6.1.1. Празен оператор

```
EmptyStatement=     .
```

Празният оператор (EmptyStatement) не съдържа никакви символи и не оказва никакво влияние на работата на програмата. Поради това е възможно поставяне на ; след всеки оператор (дори няколко ;), без това да влияе на програмата. Единственият случай, в който трябва да се внимава, е - никога пред ключовата дума ELSE да не се поставя ';', защото по този начин завършваме оператора IF преди ELSE частта.

6.1.2. Оператор за даване на стойност

Операторът за даване на стойност се използва за промяна на стойност на променлива или за указване на резултата на функция.

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

Под FuncIdent се има предвид идентификатор на функция. Лявата и дясната част на оператора за даване на стойност трябва да бъдат от съвместими типове.

6.1.3. Оператор за активиране на процедура

Операторът за активиране на процедура се състои от името на самата процедура, която трябва да бъде активирана. Ако тази процедура има параметри, то името й трябва да бъде последвано от списък от фактически параметри.

```
ProcedureCall=      QualIdent [ ActualParamList ].
```

Типът и видът (променлива или израз) на всеки параметър от списъка трябва да бъде съобразен с типа и вида на съответния формален параметър (за подробности виж главата за описание на процедури и функции).

6.1.4. Оператор за преход

Общият вид на оператора за преход е

```
GotoStatement=      'goto' Label.
```

Той променя нормалния ход на програмата, като безусловно предава управлението към отбелязания с етикета Label оператор. Използването на оператора за преход нарушава структурата на програмата и от гледна точка на структурното програмиране е желателно да не бъде използван. Всъщност използването му е оправдано само в изключително редки случаи.

6.2. Сложни (структурирани) оператори

Сложните (структурирани) оператори са оператори, включващи в себе си други оператори на Pascal (прости и сложни).

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

6.2.1. Съставен оператор

Съставният оператор (CompoundStatement) се състои от последователност от оператори, изпълнявани в реда, в който са написани (ако този ред не бъде нарушен от оператор за преход). Тези оператори са оградени от така наречените операторни скоби BEGIN и END. Общият вид на този оператор е:

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

Съставният оператор обикновено се използва на място, където синтаксиса на Pascal разрешава само един оператор, а е необходимо написването на повече. Например:

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

6.2.2. Условен оператор (if)

Условният оператор изпълнява даден оператор в зависимост от стойността на зададен логически израз, в противен случай изпълнява друг оператор (в частност - нищо друго). Общият вид на оператора е:

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

Expression е израз от тип Boolean. Ако този израз има стойност TRUE, изпълнява се операторът, стоящ след ключовата дума THEN. След завършването на изпълнението му се продължава с изпълнението на следващия след оператора IF оператор. Ако

стойността на Expression е FALSE и ако има ELSE част, изпълнява се операторът, стоящ след ключовата дума ELSE.

Забележка: Синтактичната двусмисленост на оператора

```
if Expr1 then if Expr2 then Statement1 else Statement2
```

се разрешава, като се приема, че тя е еквивалентна на:

```
if Expr1 then begin
  if Expr2 then
    Statement1
  else
    Statement2
  end { if }
```

т.е. приема се, че ELSE частта на оператора IF принадлежи на най-близкия предхождащ го оператор IF, за който няма такава.

6.2.3. Селективен оператор (case)

Селективният оператор се състои от израз, наричан още селектор (selector), и списък от оператори, на всеки от които е съпоставена една или няколко константи от тип, съвместим с този на селектора. Селекторът, а следователно и константите, трябва да бъдат от дискретен (ordinal) тип.

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

Selector=           Expression.

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

При изпълнение на този оператор се пресмята изразът (селекторът) и се изпълнява този оператор, на който е съпоставена константата, равна на изчислената стойност на селектора. Ако такъв оператор няма, изпълнява се операторът, пред който стои ключовата дума ELSE, а ако и такъв оператор няма (т.е. не е зададена ключова дума ELSE), не се изпълнява нищо.

Забележки:

1. Възможността за задаване на оператор, който се изпълнява, ако изчислената стойност на селектора е различна от всички зададени константи (ELSE частта на оператора CASE), е особеност на UniPascal и не съществува в стандарта на Pascal.

2. Една и съща константа не може да се използва два пъти в един и същ оператор CASE.

6.2.4. Оператори за цикъл

Операторите за цикъл указват, че изпълнението на даден оператор (или група оператори) трябва да се повтаря. Броят на повторенията на изпълнението на операторите може да бъде известно предварително или да се определя чрез пред- или постусловие. Операторът (или групата оператори), който се повтаря се нарича тяло на цикъла.

```
RepetativeStat=     ForStatement   |
                    WhileStatement |
                    RepeatStatement.
```

6.2.4.1. Цикъл с предусловие (while)

Общият вид на оператора за цикъл с предусловие е следния:

```
WhileStatement=     'while' Expression 'do' Statement.
```

Намиращият се след ключовата дума DO оператор се повтаря до тогава, докато изразът Expression има стойност TRUE. Expression е израз от логически тип, който се пресмята преди всяко изпълнение на тялото на цикъла. Това означава, че тялото на цикъла може да не се изпълни нито веднъж (ако Expression = FALSE).

6.2.4.2. Цикъл с постусловие (repeat)

Общият вид на оператора за цикъл с постусловие е следния:

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

Намиращите се между ключовите думи REPEAT и UNTIL оператори се повтарят до тогава, докато изразът Expression получи стойност TRUE (Expression е израз от логически тип, който се пресмята след всяко изпълнение на тялото на цикъла). Това означава, че тялото на цикъла се изпълнява поне веднъж.

6.2.4.3. Цикъл с параметър (for)

При цикъла с параметър един оператор се повтаря определен брой пъти, като едновременно с това параметърът на цикъла се увеличава/намалява при всяка итерация. Общият вид на оператора е:

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

Намиращата се след ключовата дума FOR променлива е параметър на цикъла и трябва да бъде от дискретен (ordinal) (с изключение на LongInt) тип. Нейната стойност се увеличава или намалява с 1 (в зависимост от това дали е използвана ключовата дума TO или DOWNTO) на всяка итерация на цикъла (по-точно на всяка итерация променливата получава стойност чрез стандартните функции SUCC или PRED, тъй като увеличаването с 1 е възможно само за числови променливи). Двата израза, ограждащи ключовата дума TO (DOWNTO), определят началната и крайната стойност на параметъра на цикъла. Тези два израза се пресмятат само веднъж, в началото на изпълнението на оператора FOR, и ако още тогава началната стойност е по-голяма (по-малка, ако е използвано DOWNTO) от крайната, то цикълът не се изпълнява нито веднъж.

Необходимо е спазването на следните ограничения:

- зададена като параметър променливата не трябва да изменя стойността си в тялото на цикъла (въпреки че UniPascal не прави проверки за нарушение на тази забрана);
- параметърът на FOR цикъла трябва да бъде локална променлива за блока, в който се използва цикъла;
- след нормално завършване на цикъла FOR, стойността на параметъра на цикъла е неопределена.

6.2.5. Оператор за присъединяване (with)

Операторът за присъединяване се използва за по-кратък запис на обръщенията към полетата от един или няколко различни записа. Вътре в оператора всички полета на специфицирания запис са достъпни и чрез техните имена като обикновени променливи. Общият вид на оператора е:

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

В тялото на оператора за присъединяване (по-точно в оператора след ключовата дума DO) за всеки идентификатор се проверява дали може да бъде интерпретиран като поле от зададения запис и, ако е възможно, той винаги се интерпретира така. Това означава, че всички константи, типове, променливи, процедури и функции с идентификатори, които съвпадат с имената на полетата на записа, стават недостъпни в тялото на оператора with.

Операторът

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

е еквивалентен на

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

Ако при задаването на записа се използва указател или индексация на масив, то тези действия се извършват само веднъж в началото на оператора WITH. Например, следните два фрагмента от програма не само не са еквивалентни, но най-вероятно вторият ще доведе до непредсказуеми резултати, ако стойността на променливата I не е определена в началото на оператора WITH, а ако е определена, то той представлява десетократно даване на стойност на един и същ елемент от масива, като разбира се в крайна сметка остава в сила последното даване. Докато първият фрагмент е даване на стойност на десетте елемента от масива. Считаме, че е валидна следната декларация:

```
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. Процедури и функции

Процедура или функция се нарича именована част от програмата, която се извиква (изпълнява) с помощта на оператора за извикване на процедура или чрез използуване на името на функцията в израз. Програмистът може да описва нови функции и процедури. Имената на процедурите и функциите са идентификатори, които се подчиняват на правилата за видимост на идентификаторите. Описанията на процедурите и функциите се обединяват в общ раздел на описание на процедури и функции.

7.1. Описание на процедура

Описанието на процедура е предназначено за дефиниране на нейното име, свързания с това име блок и указване на списък от формални параметри. Името на процедурата и параметрите се задават чрез заглавната част от описанието на процедурата.

```
ProcDeclaration=    ProcHeading ';' (Block | Directive) ';'.
ProcHeading=        ['segment'] 'procedure' Ident [FormalPList].
Directive=          'forward' | 'external' |
                    ('code' IntConst {',' IntConst}).
```

Обикновено след заглавната част от описанието на процедурата следва нейното тяло (Block). Това е най-разпространения вид описание на процедура.

Друг вид описание е изпреварващото описание. При него се дават две описания на процедурата. При първото - след заглавната част от описанието следва директивата FORWARD. По този начин процедурата може да се използва преди да е описано нейното тяло. Второто описание на процедурата трябва да бъде дадено до края на раздела за описание на блока, в които е дадено изпреварващото описание, и трябва да съдържа тялото на процедурата (не се разрешава второто описание да съдържа директива). В стандарта на Pascal второто описание не трябва да съдържа списък с формалните параметри. В UniPascal е разрешено да се даде повторно списъка с формални параметри, но в такъв случай той трябва да съвпада със списъка от изпреварващото описание.

Директивата EXTERNAL е предвидена за свързване към програмата на външни процедури (написани на асемблер).

Директивата CODE е за специален вид процедури, чиито код (Y код) се дава непосредствено след директивата.

Използването на префикса SEGMENT пред описанието на процедурата указва на компилатора, че процедурата е сегментирана. В UniPascal по този начин се реализират овърлейните подпрограми. Това означава, че кодът, генериран за тялото на процедурата (заедно с вложените в нея процедури), ще се намира в паметта на компютъра само докато тя (или някои от вложените в нея процедури) е активна. Ако имате поне една сегментна процедура в тялото на програмата (модула), то файлът, в който е генериран кодът на програмата (модула), не се затваря по време на работа на програмата, за да може да се зарежда в паметта при необходимост.

Ако в тялото на описваната процедура има обръщение към самата нея, то тя е рекурсивна. Изпреварващо описание се използва, когато е налице неявна рекурсия (процедурата A използва B, която на свой ред използва A и не е вложена в нея).

7.2. Описание на функция

Описанието на функция е аналогично на описанието на процедура. Единствената разлика между процедура и функция, е че функцията връща явен резултат от работата си. Затова описанията на функция и процедура се различават съвсем малко. Вместо ключовата дума PROCEDURE в описанието на функцията стои FUNCTION, и се задава типът на резултата, връщан от функцията. В Pascal функциите могат да връщат резултат само от прост или указателен тип.

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

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

В тялото на процедурата трябва поне на едно място да се среща оператор, даващ резултата от работата на функцията (оператор, даващ стойност на името на функцията или стандартната процедура RETURN). Използването на името на функцията на друго място, освен в лявата част на оператор за даване на стойност, означава рекурсивно обръщение към функцията.

7.3. Формални параметри

Параметрите позволяват при всяко извикване на процедурата да се работи с обекти (стойности или променливи), задавани в точката на извикване. Списъкът от формални параметри определя имената и типа, които се използват за работа с тези обекти в тялото на процедурата или функцията.

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

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

Формалните параметри могат да бъдат следните видове:

- параметри-значения - пред описанието им няма ключова дума VAR или CONST;
- параметри-променливи - пред описанието им стои VAR и техният тип е зададен;
- параметри-константи - пред описанието им стои CONST и техният тип е зададен;
- нетипизирани параметри - пред описанието им стои VAR или CONST (и в зависимост от това са нетипизирани константи или променливи параметри) и не е указан техния тип.

7.3.1. Параметри-значения

Идентификаторът на параметър-значение е еквивалентен на идентификатор на локална променлива, на която преди изпълнението на процедурата (в точката на извикване) е дадена стойността на фактическия параметър. Изменението на параметъра-значение в процедурата не се отразява върху фактическия параметър (той дори може да бъде израз). От извикващата процедура се предава само стойността на фактическия параметър, затова този вид параметри се наричат още и параметри, предавани по стойност. Типът на фактическия параметър трябва да бъде съвместим (assignment compatible) с този на формалния параметър.

7.3.2. Параметри-променливи

Идентификаторът на параметър-променлива се третира аналогично на параметъра-значение със следните разлики:

- всяко изменение на формалния параметър е изменение и на зададения фактически параметър;
- като фактически параметри могат да бъдат задавани само променливи (не и изрази), типът на фактическия параметър трябва да бъде идентичен (type identity) с този на формалния параметър;
- разрешено е предаването на променливи от файлов тип.

Този вид параметри се наричат още и параметри, предавани по адрес, тъй като обикновено се реализират чрез предаване на адреса на формалния параметър.

7.3.3. Параметри-константи

Параметрите-константи в UniPascal са разширение на стандарта на Pascal. Компилаторът не разрешава промяната на тяхната стойност в тялото на процедурата (т.е. не може да им се дава стойност или да се предават като параметри-променливи в друга процедура). Когато са от прост тип, параметрите-константи се предават по стойност, а ако са от съставен тип, те се предават по адрес. Това позволява пестене на памет и време.

Като пример нека разгледаме следната функция, пресмятаща детерминантата на матрица 3x3, чийто тип е дефиниран съответно като 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 };
```

Ако пред формалния параметър не е поставена ключовата дума CONST, то фактическият параметър ще се предава по стойност. А стойността на един масив се състои от стойностите на всички негови елементи, т.е. на функцията ще бъдат предадени девет реални числа. За тази дейност се губи както време, така и памет (за самата локална променлива 'a'). Обикновено в такива случаи програмистите предпочитат да използуват параметър-променлива, но той не отразява реалното използуване на параметъра, защото VAR означава, че параметърът може да бъде променен. Ако типът на параметъра е низ, то неприятностите, свързани с използуването на ключовата дума VAR, вече са реални. Вижте следната подпрограма, отпечатваща дадения й като параметър низ. При това низът се отпечатва ограден с единични кавички и вместо всяка съдържаща се в него кавичка се отпечатват две кавички (както низът се задава по синтаксиса на Pascal):

```
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 };
```

Тази процедура може да се извиква с фактически параметър, представляващ променлива от тип низ или константа-низ. Ако вместо CONST, е написано VAR, то фактическият параметър може да бъде само променлива от тип низ. А ако се премахне ключовата дума CONST, нейният параметър може да бъде и константа, но затова пък ще се губи време и памет за предаването на цялата стойност на низа.

7.3.4. Нетипизирани параметри

Нетипизираните параметри (променливи или константи) в UniPascal също са разширение на Pascal. Те се предават винаги по адрес. Фактическият параметър може да бъде променлива от произволен тип. В тялото на процедурата не е възможно да се използват директно, а само чрез преопределяне на типа (Type Cast). Ето пример на функция, която изчислява контролна сума (CheckSum) на променлива от произволен тип:

```
function CheckSum(const mem; ByteSize: natural): LongWord;
  type mem_array = packed array [1..max(natural)] of byte;
  var i: natural;
      sum: LongInt;
begin { контролната сума се пресмята, като се съберат }
  sum:= 0; { стойностите на всички байтове }
  for i:= 1 to ByteSize do begin
    sum:= sum + mem_array(mem)[i];
  end { for };
  ChåckSum:= sum; { или return(sum) }
end { ChåckSum };
```

7.3.5. Параметри от тип низ (STRING)

Стандартният (в UniPascal) тип STRING е привилегирован по отношение на другите типове. При предаване на параметри от тип низ (STRING) не е необходимо формалният и фактическият параметри да имат идентични типове, достатъчно е фактическият параметър да има не по-малка максимална дължина от формалния. При задаване на формален параметър от тип низ е разрешено да се задава и максималната му граница. Например:

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

На така описаната процедура ExpandTabs може да се предават променливи от тип string[x], където x е не по-малко от 79.

Ако се зададе формален параметър-променлива от тип низ без спецификатор за дължина, типът му не се възприема като низ с максимална големина (string[255]), както това е при променливите. Например, нека горната процедура е дефинирана като:

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

В този случай спецификаторът за дължина на фактическия параметър също се предава на процедурата неявно. Поради това максималната дължина на този низ е

динамична, т.е. тя е неизвестна по време на компилация и е различна за различните фактически параметри. Например:

```
procedure ExpandTabs(var s: string);
  var i: shortcard;
begin                                   { Тази процедура замества всички символи <TAB> в }
    i:= 0;                                { даден низ с необходимото количество интервали}
    repeat i:= i +1;
        if s[i] = #9 then begin                    { всяка табулация се заменя с толкова}
            s[i]:= ' ';                          { интервали, че следващият символ да е на}
            while i mod 8 <> 0 do begin                 { следващата +1 кратна на 8}
                i:= i +1;                      { т.е. <TAB> в 1-ва позиция вмъква 8 интервала}
                insert(' ', s, i);                  {ако е във 2-ра позиция - 7 интервала}
            end { while };                            { в 3-та - 6, ..., в 8-ма - само един}
        end { if };
    until i = length(s);
end { ExpandTabs };
```

Тази процедура ще даде грешка по време на изпълнение, ако се подаде такъв параметър, че след като се разширят табулациите до необходимия брой интервали се получи низ по-дълъг от максималния (даден като фактически параметър). Ако подобна вероятност има и е нежелателно предизвикването на грешка, можем да добавим допълнително условие в началото на цикъла WHILE. Това условие ще проверява дали има място за вмъкване, т.е. дали реалната дължина на фактическия параметър (SizeOf(s)), е по-малка от текущата (максималната дължина на низ е SizeOf(s) -1, тъй като при представянето на низ в оперативната памет се заема един байт в повече, представящ текущата дължина, за подробности виж глава "UniPascal в детайли").

7.4. Задаване на фактически параметри

При извикване на процедура се задава списък от фактическите параметри. Списъкът е ограден със скоби, а отделните параметри са разделени със запетая. Всеки фактически параметър може да бъде променлива или израз. На списъка от фактически параметри се съпоставя списъкът от формални параметри, като на първия фактически параметър се съпоставя първият формален и т.н. При съпоставянето е необходимо да има съвместимост (такава, каквато се изисква за съответния вид параметър) между типовете на формалния и фактическия параметри. Тази съвместимост трябва да бъде съвместимост за даване на стойност, ако параметърът се предава по стойност, и идентичност, ако параметърът се предава по адрес. От това правило изключения са само следните случаи:

- ако формалният параметър е нетипизиран, то фактическият параметър може да бъде променлива от всякакъв тип;
- ако формалният параметър е от стандартния тип BYTE, WORD или LONG-WORD, то фактическият параметър може да бъде от всеки тип с размер съответно 1, 2 или 4 байта;
- компоненти на пакетирана структура не могат да бъдат предавани параметри-променливи;
- ако формалният параметър-променлива е от тип STRING със спецификатор за дължина, то достатъчно е фактическият параметър да има не по-малка максимална дължина от формалния;

- ако формалният параметър-променлива е от тип STRING без спецификатор за дължина, то на максималната дължина на фактическия параметър не се налагат ограничения;
- ако формалният параметър е от стандартния тип POINTER, то фактически параметър може да бъде всяка променлива от указателен тип.

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

Забележка: По-голямата част от стандартните процедури и функции в Pascal не се подчиняват на правилата за използване на параметрите. Затова е по-правилно те да се наричат конструкции на езика (каквито всъщност са), а не процедури и функции. Но в настоящото ръководство се придържаме към общоприетото им название - процедури или функции.

8. Модули

В тази глава ще бъде описана възможността на UniPascal за разделно-модулна компилация. Такава възможност в стандарта на Pascal не съществува.

Модулът е множество от константи, типове, променливи, процедури и функции, компилирани отделно. Модулите се използват за две основни цели:

- като библиотека от изброените обекти. По този начин се намалява времето за компилация (те вече са компилирани) и се избягва многократното написване на едни и същи, често използвани части от програма;
- реализиране на модулния принцип при съставяне на програми.

При изпълнение на програма след нейното зареждане в паметта на компютъра се зареждат и инициализират модулите, които тя използва явно или неявно. Неявно използвани са тези модули, които програмата не използва, но се използват от използвани в нея модули (по премълчаване модулът STANDARD се използва от всяка програма). По този начин програмата на UniPascal представлява цяла йерархия от модули, в която на най-ниско ниво стоят модулите, които не използват други модули.

При компилация на програма (или модул), която използва модули, компилаторът трябва да има достъп до вече компилираното описание на експортираните (изнасяните навън) обекти. Този механизъм е качествената разлика между разделно-модулната компилация и независимата компилация (каквато има например във FORTRAN-IV).

Модулите се състоят от две основни части: част за описание (interface part) и част за реализация (implementation part). В UniPascal е възможно тези две части да бъдат компилирани поотделно или заедно. Има и четири вида модули:

- модул-описание, съдържащ само частта за описание;
- модул-реализация, съдържащ само реализацията;
- обикновен модул, съдържащ и двете части;
- модул-само-описание (interface only unit), не съдържащ реализация и ненуждаещ се от такава част.

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

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

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

От синтаксиса се вижда, че след идентификатора на модул (в обикновен модул или в описателната част на модул) може да има целочислена константа, поставена в скоби. Тази константа указва номера на версията на модула (за подробности виж глава "Модули и техните версии").

8.1. Раздел за описание (interface part)

```
InterfacePart=     [UsesClause] {
                    ConstDeclaration |
```

```
                TypeDeclaration |
                VarDeclaration |
                PFDeclaration }.
```

В описанието на даден модул, независимо дали е раздел от обикновен модул, модул-описание или е модул-само-описание, се декларират (описват) обектите, които се експортират навън от него. Описанието на константи, променливи и типове не се отличава по нищо от описанието, което би било дадено в обикновена процедура. Описание на етикети не е разрешено, а описанието на процедури и функции е сведено само до задаване на заглавната част от описанието (не трябва да се задава тяло на процедурата или директивата FORWARD).

Описаните в тази част обекти стават видими за импортиращия (използващия) ги модул или програма. Освен това, за импортиращия модул (програма) те са и единствените достъпни за използване обекти от този модул (обектите, дефинирани в реализационната част, са невидими).

Чрез тази схема се реализира разделно-модулната компилация, като се запазва строгата типизация. Компилаторът продължава да контролира съвместимостта на типовете и броя на параметрите на процедурите и функциите. Самата реализация остава скрита за импортиращия модул. Нещо повече, в UniPascal е възможна промяна на реализацията (изменение и прекомпилация), без да е необходима прекомпилация на използващите я модули.

Следва пример на модул, реализиращ стек от цели числа. За работата на стека са необходими следните четири операции:

- поставяне на елемент в стека;
- изваждане на елемент от стека;
- проверка за празен стек;
- проверка за пълен стек.

Проверката за пълен или празен стек ще реализираме чрез логически функции.

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

При така зададеното описание на модула STACK, той може да се използва от програмата, като съставителят й не знае предварително колко е голям стекът и как е реализиран.

8.2. Раздел за реализация (implementation part)

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

Разделът за реализация практически не се отличава от обикновена програма. Разликите се състоят в следното:

- не е необходимо (и не трябва) да се декларират отново обектите, описани в частта за описание. По правилата за видимост те се виждат като глобални идентификатори;

- на всички описани в частта за описание процедури и функции трябва да бъде дадена реализацията. Описанията в interface частта се разглеждат от компилатора като изпреварващи (forward) описания;
- всички глобални променливи на модула, описани както в interface, така и в implementation частите, съществуват през цялото време на работа на програмата или на модула, използващ този модул;
- тялото на модула (операторите между BEGIN и END в implementation частта на модула), се изпълнява преди изпълнението на първия оператор от използващия го модул и играе роля на инициализираща част;
- ако бъде използван етикетът EXIT в тялото на модула, то по време на инициализацията на модула се изпълняват всички оператори преди този етикет, но не и тези след него. При завършване на работата на модула или програмата, използващи този модул, управлението се предава на етикетирания с EXIT оператор. По този начин може да се предприемат необходимите действия по завършване работата на този модул (затваряне на файлове и други такива).

```
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 на модула STACK }
  writeln('Начало на инициализация на модула STACK');
  index:= 0; IsFull:= false; IsEmpty:= true;
  writeln('Край на инициализацията на модула STACK');
EXIT:
  writeln('Край на модула STACK');
end { Stack }.
```

8.3. Модул-само-описание (interface only unit)

В някой случаи не е необходимо модулът да има реализационна част. Типичен пример е модул, в който има описани само типове. В такъв случай описването на празна реализационна част е едно възможно решение. Единственият проблем (от който можем да се абстрахираме) е, че тази реализационна част, независимо че е празна, все пак използва памет (в оперативната памет и върху диска), трябва да се търси и свързва нейната реализационна част и т.н.

Друго възможно решение е да опишете модула като модул-само-описание (без реализационна част). В този случай модулът-само-описание няма реализационна част и, следователно, няма да се използват допълнително области в паметта, нито пък ще съществува файл с компилирана реализационна част на модула.

Синтаксисът на модула-само-описание е като този на обикновен модул, но цялата реализационна част (от IMPLEMENTATION до END) липсва. Ако създавате програма с няколко модула, удобно е типовете, които ще използвате навсякъде, да бъдат описани чрез модул-само-описание (такъв е модулът UniLEX, намиращ се на дистрибутивната дискета).

В модул-само-описание може да има декларирани: типове, прости константи (без низове), процедури или функции, описани с директивата CODE. Не е разрешено декларирането на променливи или низови константи.

Следва пример на модул-само-описание:

```
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. Използване на модулите

Използването на модулите от програми или други модули става чрез клаузата USES. След ключовата дума USES следва списък от използваните модули.

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

Всички експортирани от модулите идентификатори автоматично стават видими като глобални. Освен по обикновения начин, експортираните идентификатори могат да

бъдат използвани чрез квалифицирано указване. Това става като първо се изписва името на модула, а след него името на необходимия идентификатор, разделени с точка (същото както при записи, с тази разлика, че вместо името на записа се използва името на модула).

Ако два модула експортират един и същи идентификатор, то този идентификатор е недостъпен за директно използване, поради неоднозначността си. В този случай е разрешено само квалифицираното му използване.

Всички стандартни идентификатори, достъпни в UniPascal, са декларирани в модула STANDARD. Затова е възможно квалифицираното им използване от този модул.

Пример на програма, използваща модула STACK:

```
program UseStack(message);
  uses Stack;
  var i: integer;
begin
  writeln('Начало на програмата.');
  writeln('Поставяне на цели числа в стека...');
  i:= 0;
  while not full do begin
    writeln('Поставяне на ', i, ' в стека.');
    push(i); inc(i);
  end { while };
  writeln('Изваждане на числата от стека...');
  while not empty do begin
    writeln('Извадихме ', i, ' от стека.');
  end { while };
  writeln('Край на програмата.');
end { UseStack }.
```

На настоящия пример не е необходим голям стек. Но ако напишете програма, използваща по-голям стек, достатъчно е да смените само константата в реализационната част на модула. При това не е необходимо да се прекомпилира изобщо програмата, използваща стека. Нещо повече - ако стекът е твърде голям, за да се побере в паметта на компютъра (особено ако стекът се състои не от цели числа, а от записи или масиви), може да промените изцяло реализацията, като използвате външна памет (дисков файл).

9. Компилация и управление на компилацията

```
Compilation=        Program | Unit | InterfaceUnit | ImplmntUnit.
```

Компилаторът от UniPascal може да компилира следните обекти (Compilation), които са в съответствие със синтаксиса на UniPascal:

- програма (Program);
- модул (Unit);
- модул-описание (InterfaceUnit);
- модул-реализация (ImplmntUnit).

Някои от възможностите на UniPascal се управляват чрез така наречените директиви на компилатора. Директивите на компилатора се записват като коментар със специален синтаксис:

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

Както се вижда от синтактичното правило, това е коментар, започващ със знака '$', последван от директивата (една директива, а не няколко). Всяка директива трябва да бъде записана изцяло на един ред.

Въпреки че всички реализации на Pascal използват този синтаксис за управление на работата на компилатора, всички те се различават в по-голямата си част по директивите си. Някои от тях (например UCSD) просто игнорират директива, която не 'познават', други (като Turbo Pascal) съобщават за грешка в този случай, а трети (в това число UniPascal) само предупреждават потребителя за това.

Повечето директиви се състоят от една буква, последвана непосредствено (без интервали) от знака + или -, като по този начин се включва или изключва съответната възможност на компилатора. Такива директиви ще бъдат наричани превключваеми. За тях е направено малко изключение по отношение на синтаксиса на записване, като той е малко по-свободен. В случая се има в предвид това, че ако трябва да се превключат няколко възможности едновременно, не е нужно да се изписват толкова управляващи коментара, колкото са те. Достатъчно е да се напишат в един коментар, но трябва да бъдат разделени със запетаи. Всички символи след последния превключвател се приемат за коментар. Ако в някоя от директивите е допусната грешка (или директивата не съществува в UniPascal), всички останали директиви след нея се игнорират.

Другият вид директиви са тези, които след еднобуквения спецификатор на директивата очакват параметър (отделен с поне един интервал).

Последният вид директиви са тези, свързани с условната компилация. При тях след знака '$' следва една от специалните думи, евентуално последвана от параметър (виж глава "Условна компилация").

9.1. Превключваеми директиви

В дадения по-долу списък на превключваемите директиви (подреден по азбучен ред) след името (буквата) на директивата е поставен знак + или - в зависимост от стойността на ключа по премълчаване. Например (*$I+*) означава, че директивата I е включена по премълчаване.

9.1.1. Проверка на резултата на вход/изхода (*$I+*)

Този ключ разрешава (+) или забранява (-) генерацията на код за проверка на резултата от входно/изходните операции. Ако е изключен, то програмата продължава изпълнението си при входно/изходна грешка. Предполага се, че програмистът е взел необходимите мерки в програмата си.

При (*$I+*) използването на стандартната функция IOresult е безсмислено, защото програмата ще преустанови изпълнението си (по входно/изходна грешка), преди да се изпълни обръщението към функцията IOresult.

9.1.2. Modula-2 в UniPascal (*$M-*)

Ключът M превключва два възможни синтаксиса на операторите в UniPascal. Когато е изключен, за компилатора са валидни операторите, описани в предходните глави на този документ. Когато е включен, компилаторът третира операторите, като такива от език, подобен на Modula-2. Тук ще дадем само синтактичните правила на операторите на UniPascal при използване на директивата (*$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.
```

Както се вижда, отпада необходимостта от съставния оператор (BEGIN ... END) и той вече не съществува. Затова пък всеки сложен оператор завършва с END. Операторът REPEAT ... UNTIL не се променя, а синтаксисът му е даден само за пълнота. Освен смяната на синтаксиса, използването на (*$M+*) довежда и до появата на нова ключова дума - ELSIF. И последната промяна: след ключовата дума END в края на тялото на процедура, функция, модул или програма трябва да стои идентификаторът (името)

на процедурата, функцията, модула или програмата, т.е. променя се синтаксиса на блок.

9.1.3. Проверка на името или разширение на Modula-2 (*$N-*)

Нормално този ключ е изключен. Но ако е включен, то тогава в зависимост от състоянието на ключа M този ключ превключва две различни възможности.

- ако ключът M е изключен, тогава е в сила стандартният синтаксис на Pascal с едно малко разширение: изисква се след края на процедурата или функцията да стои идентификатора й, т.е. името да се повтаря след ключовата дума END;
- ако ключът M е включен, изисква се след всяка ключова дума END да стои ключовата дума, с която е започнал съответният сложен оператор (ако това е край на сложен оператор).

Например:

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

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

9.1.4. Автоматично пакетиране (*$P-*)

Ако този ключ е включен, компилаторът разглежда като пакетирани всички масиви или записи. Ако ключът е изключен, пакетиране се прави само при явно указване на ключовата дума PACKED.

9.1.5. Тиха компилация (*$Q-*)

При изключено състояние на този ключ, компилаторът издава звуков сигнал при всяка грешка, а при включено състояние - не.

9.1.6. Проверка на границите (*$R-*)

Този ключ разрешава (+) или забранява (-) генерацията на код за проверка на границите на диапазон по време на изпълнение. При изключено състояние компилаторът проверява само константите и константните изрази, като съобщава за грешка при излизане извън необходимия диапазон. При включено - освен проверка на константите, по време на компилация се генерира код за проверка на индексите на масивите, даването на стойност на променливи от тип диапазон и предаването по стойност на параметри, ако формалният параметър е от диапазонен тип. Генерираният при включено състояние код индицира грешки, появяващи се по време на работа на програмата.

9.1.7. Предупредителни съобщения (*$W-*)

Този ключ разрешава (+) или забранява (-) издаването на предупредително съобщение, издадено за оператора, пред който е поставена директивата. Това означава, че този ключ е непрекъснато във включено състояние, освен в рамките на даден оператор, пред който е бил изключен. След този оператор той автоматично се включва отново. Ключът е поставен, за да не получавате при всяка компилация

стряскащо предупредително съобщение, ако наистина искате да направите това, което правите. От друга страна е нежелателно да се остави на програмиста включването му, защото той или ще забрави (или ще го домързи), а няма да са малко и тези, които ще започват всяка програма с {$W-}, без изобщо да го включват, като по този начин няма да бъдат предупредени за евентуална грешка. И един съвет: ако трябва да се изключи предупредителното съобщение за няколко последователни оператора можете да ги обедините в един (съставен) оператор като ги заградите с ключовите думи 'begin' и 'end' и напишете само едно {$W-} пред този съставен оператор.

Този ключ обикновено се използува, когато трябва да запълните масив от даден елемент нататък с дадена стойност с помощта на стандартните процедури FILLCHAR/FILLWORD, или когато премествате част от масив в друг масив с помощта на MOVE/MOVEWORDS. Например, ако s е низ (var s: string) и искаме да запишем 5 интервала от третия символ нататък може да използваме цикъл:

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

Но обикновено се използува стандартната процедура FILLCHAR, защото е по-бърза и по-кратка:

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

При това използване компилаторът ще даде предупредително съобщение, защото големината на един елемент е 1 байт, а е зададено запълване на 5 байта. В такъв случай трябва да поставим {$W+} пред FILLCHAR.

9.1.8. Специален вид условна компилация (*$Y+*)

Този ключ разрешава (+) или забранява (-) възприемането на последователността '(*)' като цялостен коментар. Съществуването на тази наглед безсмислена възможност дава един елементарен вид условна компилация, който ни измъква от някои иначе непреодолими препятствия при преносимостта между UniPascal и някоя друга реализация на Pascal. Ще разгледаме действието й чрез примери. Нека имаме:

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

Тогава, ако ключът Y е изключен, написаното е коментар със следния текст ') writeln; (', но ако е включен, това са два коментара и между тях е writeln. По този начин получаваме условна компилация от вида {$ifOpt Y+} ... {$endif} (виж "Условна компилация"). А ако имаме

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

то при изключен ключ Y това ще представлява коментара - ') writeln; {', последван от readln;, последван от коментар - звездичка (звездичката в коментара няма функционално предназначение и може да се постави произволен коментар). При включен ключ разглежданият пример представлява специалния вид коментар (*), последван от writeln;, последван от коментар, който съдържа '*) readln; {*'. Така получихме аналог на условната компилация {$ifOpt Y+} ... {$else} ... {$endif}.

При използването на този вид условна компилация трябва да се внимава с използването на коментар, заграден от (*), {*) и {*}. Основно правило е, че в първата част от условната компилация (тази заградена в (*) и {*) или в (*) и (*)) може да се използват коментари само във фигурни скоби '{ }'. А във втората част (заградена от {*) и {*}), коментарът може да бъде ограден само с коментарните скоби (* и *). Все пак по-добре е да не се използва коментари (не изобщо, а само в условната компилация от този вид), ако има макар и малка неяснота в принципа на работа на условната компилация.

9.2. Параметрични директиви

В настоящата реализация на UniPascal този вид директиви са две - за свързване с външни процедури (LINK) и за включване на файл (INCLUDE). И двете директиви имат един и същ вид - след буквата I или L в началото на директивата се очаква име на файл.

9.2.1. Включване на файл в текста на програмата (INCLUDE)

Директивата за включване на файл е (*$I filename *), като за компилатора файлът с име FILENAME ще изглежда така, сякаш е поставен на мястото на директивата. Името на файла, който ще се вмъква, се указва изцяло, т.е. компилаторът не поставя автоматично суфикс на файловете, които се вмъкват. При използването на директивата се налагат следните ограничения:

- максимално ниво на влагане на такива директиви е четири. Под влагане се разбира появяването на тази директива във файл, който също се вмъква;
- тялото на всяка процедура или функция трябва да бъде изцяло в един файл;
- всеки коментар трябва да се съдържа изцяло в един файл.

9.2.2. Указване на файл за свързване (LINK)

При свързване на външни процедури компилаторът трябва да знае от кой файл да вземе генерирания от асемблера код. Това се указва чрез директивата (*$L filename *). Както и при директивата за вмъкване, така и тук името на файла се указва изцяло, т.е компилаторът не добавя никакъв суфикс по премълчаване. При използването на директивата има следните ограничения:

1) Директивата не трябва да се среща на максималното ниво на влагане на INCLUDE файлове.

2) Тъй като всяка външна процедура се търси във файла, указан чрез тази директива, компилаторът отваря за четене указания файл и го затваря, ако:

- срещне друга директива за свързване;
- завърши компилация;
- започне компилация на нов сегмент;
- отвори файл за вмъкване;
- срещне директивата за свързване без име на файл, т.е. (*$L*). Това е направено, за да може да се накара компилаторът да затвори файла с външните процедури.

3) Ако компилаторът е затворил файла с външните процедури, при поява на описание на такава ще бъде съобщена грешка.

Препоръчва се след указването на файл за свързване да се опишат всички външни процедури, които се използват в него, и след това този файл да бъде затворен. Ако в програмата на две места се свързва един и същ файл с външни процедури, то техният код ще бъде копиран два пъти в програмата, което ще доведе до загуба на памет, а в някой случай и до грешки.

9.3. Условна компилация

В някои случаи е желателно да можете да указвате на компилатора, че дадени части от програмата не трябва да се компилират. Най-прост пример за това е тестовият печат. При него съществува и друга възможност - просто го изтривате, но това означава да обходите цялата си програма и не е ясно дали няма да забравите нещо. А ако ви се наложи отново да го включите, трябва да го пишете отново.

В такива случаи се използва условна компилация. За програмистите, работили на асемблер, това не е нищо ново и те трябва само да научат синтаксиса. При някои реализации на Pascal тя също е реализирана, но в ISO стандарта условна компилация не съществува. Условната компилация в UniPascal е не само силно повлияна от тази в разпространената версия на Turbo Pascal 4.0+ за 16 битовите компютри IBM PC, но нещо повече - тя е същата. Така че програмисти, работили на Turbo Pascal, могат да прескочат това описание.

9.3.1. Директиви DEFINE и UNDEF

Дефинирането на идентификатори за условна компилация става чрез директивата {$define xxx}, където xxx е идентификатор за условната компилация. Този идентификатор няма нищо общо с нито една част от вашата програма, нищо не ви пречи да имате променлива, процедура, тип и т.н. със същото име. Двата идентификатора няма да имат нищо общо помежду си. При идентификаторите за условна компилация важното е само дали те са дефинирани или не. Единственото ограничение е: общата дължина на едновременно дефинираните идентификатори за условна компилация може да бъде най-много 70 букви.

Може да премахнете дефиницията на даден символ за условна компилация чрез аналогичната директива {$UnDef xxx}. Тази директива премахва дефинирането на идентификатор за условна компилация, т.е. по този начин се забравя за съществуването на такъв идентификатор за условна компилация.

В UniPascal е дефиниран по премълчаване идентификатора UniPas.

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

Основната идея на условната компилация е да се укаже на компилатора, че ако е изпълнено някакво условие, той трябва да компилира дадена част от текста на програмата, в противен случай - друга част (или, в частност, нищо).

Общият вид на условната компилация е:

```
{$IFxxx condition}
   { текст, който да се компилира, ако условието се удовлетворява }
{$ENDIF}
```

или

```
{$IFxxx condition}
   { текст, който се компилира, ако условието се удовлетворява }
{$ELSE}
   { текст, който се компилира, ако условието не се удовлетворява }
{$ENDIF}
```

където IFxxx е IFDEF, IFNDEF или IFOPT, а condition е проверяваното условие.

При реализираната в UniPascal условна компилация проверяваното условие зависи от използваната директива IFxxx по следния начин:

- при използване на директивата IFDEF условието се удовлетворява, ако даденият параметър е идентификатор за условна компилация, който **е** дефиниран;
- при използване на директивата IFNDEF условието се удовлетворява, ако даденият параметър е идентификатор за условна компилация, който **не е** дефиниран;
- при използването на директивата IFOPT условието трябва да бъде от вида X+ или X-, където X е буква на някоя от превключваемите директиви. В

този случай условието се удовлетворява, ако ключът е в указаното състояние.

Най-прост пример за използването на условна компилация е тестовият печат. Нека допуснем, че в началото на всяка процедура е написано следното:

```
{$ifdef debug}
  writeln('Влизане в процедура XXX. Натисни RETURN'); readln;
{$endif}
```

където XXX е името на процедурата, в която е написан даденият текст. Сега е достатъчно да поставите в самото начало на програмата директивата {$define debug}, и контролният печат ще бъде включен в програмата ви.

10. Използване на UniPascal на микрокомпютъра Пълдин

За да изпълните програма на UniPascal на микрокомпютъра Пълдин е необходимо първо да въведете текста на програмата в компютъра и да запазите този текст като обикновен текстов файл. За това Вие може да използвате някакъв текстов редактор (например UniED).

След като имате текста на програмата, записан в текстов файл, трябва да стартирате компилатора на UniPascal, като му укажете името на текстовия файл, в който се намира Вашата програма. Компилаторът има за задача да създаде изпълнима програма от Вашия текст, т.е. той ще създаде файл със суфикс .pgm, който представлява програма, която може да се изпълни на микрокомпютъра Пълдин 601/601A/601M. Ако се налага, може да пренасочите стандартния изход към някой файл (ако имате много грешки по време на компилация).

Например за да 'пуснете' програмата:

```
program Hello(output);
begin
  writeln('Здравей!');
end.
```

е необходимо да извършите следните стъпки:

- въведете дадения по-горе текст с текстовия редактор и запишете резултата от работата си във файла hello.pas;
- на командната линия на UniDOS напишете upc hello. По този начин ще компилирате програмата си и в резултат ще получите файла hello.pgm;
- на командната линия на UniDOS напишете hello. Така Вие ще изпълните програмата.

Ето какво ще бъде написано (приблизително) на екрана на компютъра, при компилация на програмата HAN (задача за преместване на ханойските кули) намираща се на дистрибутивната дискета:

```
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
```

Текстът на програмата се състои от 212 реда. Генерираният за нея код е с дължина 1293 байта (без да се включва кода за модула UniCRT). Компилаторът дава на всеки сегмент уникален номер, а на всяка процедура (функция) друг уникален в рамките на сегмента номер. На главната програма е даден сегмент № 0 (s0). Модулът UniCRT се използва като сегмент № 1 (u1). Процедурата PREPDISK е компилирана под № 2 (p2), в текстът на програмата започва от ред 61. Процедурата MOVEDISK има № 3. В текста на програмата тя започва от ред 73. В нея има вложена процедура SHOWMOVE с № 4, започваща от ред 75. Тялото на процедурата MOVEDISK започва от ред 128 и т.н. Тялото на главната програма започва от ред 172. На последния ред е написано името

на файл, в който е записан резултатът от компилацията (генерираният от компилатора код на програмата) - HAN.PGM.

За да изпълните коя да е от примерните програми (дадени на дистрибутивната дискета) трябва само да я компилирате, тъй като текстът й вече е въведен.

10.1. Стандартни суфикси на файловете

При работа с UniPascal се използват следните стандартни суфикси (разширения) на имената на файловете:

- **.PAS** Това е суфикс, който компилаторът автоматично добавя към името на текстовия файл, който сте дали за компилация (ако не сте дали никакъв суфикс). Затова обикновено текстовете на програмите в UniPascal завършват на .pas;
- **.PGM** Когато компилирате програма, файлът, в които се записва генерирания за нея код, получава името на програмата (дадено след ключовата дума 'program') със суфикс .pgm;
- **.SYM** Такъв суфикс получават всички interface части на модулите, които компилирате;
- **.BDY** Това е суфикс, който се поставя на implementation частите на модулите, които компилирате.

10.2. Задавани от командната линия параметри

При стартиране на компилатора от командната линия на UniDOS трябва да зададете поне един параметър и това е името на файла, който ще компилирате. Освен него може да дадете и други параметри. Всеки параметър трябва да започва с наклонена черта (/) или минус (-). Компилаторът възприема следните параметри:

/Dxxx дефиниране на символи, управляващи условната компилация. В този параметър **xxx** е име на символа за условна компилация, който искате да дефинирате. Задаването на всеки такъв символ е еквивалентно на написването на един ред преди текста на програмата, съдържащ {$Define **xxx**};

/Uxxx тук **xxx** е списък от пътища (paths), по които компилаторът ще търси компилираните interface части (*.SYM - файловете) на използваните от програмата модули;

/Ixxx с **xxx** е означен списък от пътища, по които компилаторът ще търси всички файлове, зададени в директивата за включване на файл - {$I fname};

/Oxxx този параметър е аналогичен на **/Ixxx**, но тук се задават пътища за търсене на файловете за свързване - {$L fname};

/Lxxx тук **xxx** е името на библиотеката, която ще се използва. Името се задава изцяло (включително и пътя). По премълчаване е \SYSTEM.UPL;

/Txxx тук **xxx** е пътя (path), където компилатора трябва да създаде работен файл (необходим по време на компилация);

/Sxxx път, където да бъде създаден .SYM файлът, който се получава в резултат от компилацията на interface частта на модул;

/Bxxx път, където да бъде създаден .BDY файлът, който се получава в резултат от компилацията на implementation частта на модул;

/Pxxx път, където да бъде създаден .PGM файлът, който се получава в резултат от компилацията на програма;

/Cxxx този параметър задава едновременно и трите параметъра **/Sxxx**, **/Bxxx** и **/Pxxx**;

/W-|+ по премълчаване се подразбира **/W-**. Ако е зададено **/W+**, компилаторът очаква натискане на ENTER или ESC след всяка грешка. Ако изходния файл е пренасочен, обяснителният текст за грешката не се появява на екрана (тя се отпечатва в изходния файл);

/$x-|+ задаване началното състояние на превключваема директива. Тук '**x**' е буква от една от превключваемите директиви. Началното състояние на указаната директива се установява в зависимост от знака: при '+' е включена, а при '-' е изключена;

***xxx** този параметър не започва с / или -, а с * и задава името **xxx** на конфигуриращия файл, който трябва да се използва от компилатора. Този параметър не може да бъде поставен в конфигуриращ файл.

Във всички случаи, ако не е зададен път или списък от пътища, се подразбира текущата директория. Ако се задава списък от пътища, компилаторът търси необходимите му файлове в последователността, в която е даден списъка. Отделните елементи от списъка се разделят с ';'. В текущата директория се търси само, ако необходимият файл не е намерен в зададените от списъка пътища. Ако искате да бъде търсено първо в нея, необходимо е да я включите в списъка. Например /I;c:\work;d:\inc... или /I.;c:\work;d:\inc... (тъй като празно име или '.' обозначава текущата директория).

10.3. Конфигуриращ файл

При стартирането си компилаторът проверява за наличието на конфигуриращия файл **UPC.CFG** в текущата директория, а ако там го няма, и в тази, в която се намира самия компилатор. Ако бъде намерен (файлът UPC.CFG), компилаторът прочита от него параметрите (тези, които се задават от командната линия). Конфигуриращият файлът трябва да бъде обикновен текстов файл, като всеки параметър трябва да бъде записан на отделен ред. Едва след това се проверяват параметрите от командната линия. Параметрите от командната линия имат приоритет пред параметрите, прочетени от конфигуриращия файл. Това означава, че ако към компилатора е зададен някакъв параметър и в конфигуриращия файл, и от командната линия, то е валиден зададеният от командната линия.

10.4. Свързване на модули. Библиотека и нейното използване

В UniPascal свързването на програмата с използваните от нея модули става по време на изпълнение, независимо къде се намира техният (на модулите) компилиран код. Физически кодът на всеки модул може да се намира или в отделен файл с името на модула и суфикс **.BDY** или в библиотеката (SYSTEM.UPL) или в един файл заедно с програмата.

Добавянето на модули към програма или към библиотеката се извършва с една и съща програма **UPL**. Това е така, защото в UniPascal програмата е всъщност

библиотека, в която първият модул е изпълним. Поради това към една програма може да се свържат модули, които тя не използва явно или изобщо не използва.

Библиотеката (и програмата) е файл, в който са записани множество от модули. В нея могат да бъдат записани както описателните части на модулите, така и реализационните им части.

Записването (добавянето) и изключването на модул от библиотеката (програмата) се извършва с помощта на програмата **UPL**.

Програмата **UPL** има следните параметри: първият от тях е име на библиотеката (или програмата), а следващите параметри могат да бъдат следните:

+xxx Където **xxx** е име на файла, в който е записан модулът, добавящ се към библиотеката. При указан суфикс **.SYM** се добавя само описателната част. При суфикс **.BDY** - само реализационната част . Ако не е указан суфикс, се добавят и двете части или само тази, която е необходима. С други думи, ако в библиотеката вече се намира описателната част на модула, добавя се само реализационната, и обратно;

-xxx Където **xxx** е името на модула, който се изтрива от библиотеката;

***xxx** Където **xxx** е името на модула, който се изважда от библиотеката. Под изваждане се разбира създаване на файл с необходимия суфикс, записване на съдържанието на модула в този файл и изтриване на модула от библиотеката;

/m Дава списък на наличните в библиотеката модули;

/s Същото като **/m**, но освен това дава информация за отделните сегменти (овърлеи);

/p Същото като **/s**, но дава информация и за всяка процедура поотделно.

Ако на **UPL** се зададе само име на библиотека (или програма), дава се списък на наличните модули, сегменти и процедури в библиотеката или програмата.

Ако не й се зададе нито един параметър, програмата **UPL** изписва помощно съобщение за начина на използването й.

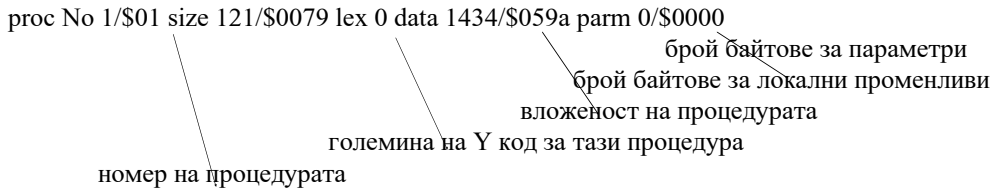
Пример: разпечатка от използването на програмата **UPL** върху себе си, т.е. изпълнение на командата:

```
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.
```

брой на сегментите в модула
дата и време на компилиране
версия на модула
име на модула

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

брой процедури
големина на таблицата за преместване
големина на константите
големина на Y код
обща големина на сегмента
номер на сегмента
име на сегмента



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

10.5. Задаване на път за .BDY файловете и за библиотеката

При изпълнение на вече компилирана програма използваните външни (тези които не са добавени към файла, съдържащ програмата) модули се свързват с нея по време на изпълнение. За това е необходимо да се укаже къде да бъдат търсени техните компилирани реализационни части (**.BDY** файловете). Файловете с компилирани interface части на модулите по време на изпълнение не са необходими и не е нужно да се указва къде се намират.

По премълчаване файловете със суфикс **.BDY** се търсят в текущата директория, а името на библиотеката е \SYSTEM.UPL. Чрез програмата BLpath, намираща се на дистрибутивната дискета, може да се укаже списък от поддиректории за **.BDY** файловете, както и друго име и/или местонахождение на библиотеката.

Параметрите на програмата BLpath са три:

BDY=xxx С този параметър се задава списък от пътищата, където трябва да се търсят файловете със суфикс **.BDY**, като отделните елементи в списъка се разделят с ';'

LIB=xxx Задава името на библиотеката и пълния й път;

? Дава помощно съобщение за използването на програмата.

Ако изпълните програмата BLpath без параметри, тя ще изпише зададения преди това (или установения по премълчаване) списък от пътища и името на библиотеката.

Реализационната част на всеки модул се търси в следната последователност:

1) В текущия файл, възможно е да е различен от програмата, ако се свързва файл със суфикс **.BDY**.

2) Във файла с програмата (**.PGM** файла);

3) В библиотеката.

4) Търси се файл със суфикс **.BDY** по списъка от пътища в зададения от него ред.

10.6. Оптимизация на програмите на UniPascal

Тъй като за компилатора на UniPascal не достига нито памет, нито време, той не генерира оптимален код. За да се оптимизира генерирания код се използва програмата **YOP**. Тя оптимизира само генерирания от компилатора код, като оставя подпрограмите на асемблер непроменени.

Първият параметър на програмата е името на програмата, която подлежи на оптимизиране. Ако не се зададе втори параметър, оптимизираният код се записва във файл под същото име, а ако се зададе, въведеното име се използва за име на файл, в който се записва оптимизираният код. Другите параметри се отнасят до метода за оптимизация. Тя може да се прави по два основни показателя: по време (параметър **/t**) и по големина (параметър **/s**). Може да се направи и оптимизация по време и големина (параметър **/b**), но тя е компромисен вариант между двете оптимизации.

В повечето случаи ефектът от двата вида оптимизации е почти еднакъв, защото Y код е компактен и повечето процедури не надвишават големина от 400 - 600 байта.

10.7. Инсталиране на UniPascal

Под инсталиране в случая се разбира подготовка за работа с UniPascal. За да работи компилаторът (UPC) не са му необходими никакви допълнителни файлове, освен разбира се, текстът на програмата. Но в такъв случай при евентуална синтактична грешка той (компилаторът) ще я съобщи само по номер. Ако искате да получите и текста на съобщението за грешка, трябва в поддиректорията, в която се намира компилаторът, да запишете файла с текстовете на съобщенията за грешките, който се намира на дистрибутивната дискета и има име **UNIPAS.ERR**.

Датата и времето са важни за системата UniPascal. За съжаление микрокомпютърът Пълдин няма часовник, който да 'помни' времето, когато го изключите, а установяването му 'на ръка' е неприятна работа. За по-голямо удобство можете да използвате програмата **DATETIME**. Тя записва върху диска текущата дата и време (променяйки собствената си дата и време), като предоставя удобен начин за установяване на текущата дата и време. Параметрите й са както следва: D записва датата върху диска, T - записва времето върху диска, * - ако текущото време е валидно, не чака въвеждането му. Включвайки реда

```
DATETIME D T *
```

във Вашия **AUTOEXEC.JOB** файл, ще можете лесно да задавате вярната дата и време.

Ако имате две флопидискови устройства, добре е на една от дискетите да запишете компилатора (**UPC.PGM**), файла с текстовете на съобщенията за грешки (**UNIPAS.ERR**) и текстов редактор (например **UniED.CMD**). А на друга дискета да записвате текстовете на програмите си и да я използвате за работна.

Ако работите в условията на мрежа, добре е да поставите всички системни програми на главния мрежов компютър (server), а да работите на локалните флопидискови устройства.

И накрая, ако имате само едно флопидисково устройство, просто нямате избор. Трябва да поставите компилатора и текста на програмата на една и съща дискета. Компилаторът е програма с много овърлейни процедури и ако се опитвате да използвате различни логически устройства върху едно физическо (т.е. A: и B: при едно устройство) ще трябва много често да сменяте дискетите.

На дистрибутивната дискета на UniPascal само компилаторът, системната библиотека и файлът със съобщенията за грешки са дадени като отделни файлове. Всички системни програми (**UPL.PGM**, **YOP.PGM**, **BLpath.PGM**) се намират във файла под името **UTILS.ARC**. За да ги 'извадите' трябва да използвате програмата **UNARC**. Най-добре е да използвате две (логически или физически) устройства. Поставете в устройство A: празна дискета, а в устройство B: дистрибутивната дискета и напишете:

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

Програмата **UNARC** ще разпакетира и запише на празната дискета всички системни програми, пакетирани и намиращи се в **UTILS.ARC**.

Друг файл със суфикс **.ARC**, намиращ се на дистрибутивната дискета, е **EXAMPLES.ARC**. В него са архивирани някои примерни програми на UniPascal. Използвайки **UNARC** можете да 'извадите' и тях от архива по вече описания начин.

11. Крос-продукти за използване на IBM PC/XT/AT

В професионалната версия на дистрибутивните дискети се намират и програмите **UPC.EXE**, **UPL.EXE** и **YOP.EXE**. Това са компилаторът (**UPC**), библиотекарят (**UPL**) и оптимизаторът (**YOP**) на UniPascal, но те могат да се изпълняват на IBM PC/XT/AT.

Начинът на използване е абсолютно същия, както и при техните основни варианти за микрокомпютъра "Пълдин". И тъй като операционните системи на двата компютъра са съвместими на ниво дисков носител, то всичко, отнасящо се до версиите на програмите на "Пълдин", се отнася и за крос-версиите на продуктите.

Разработката на програмното осигуряване за "Пълдин" се ускорява значително, ако се използват крос-версиите, тъй като компютрите IBM са 16 битови и имат значително повече памет и по-голямо бързодействие от "Пълдин" (крос-компилаторът работи от 6 до 10 пъти по-бързо на IBM PC/XT).

Генерираният от крос-компилатора код не се отличава по нищо от този, генериран от основния компилатор, и може да се изпълнява на микрокомпютъра "Пълдин". Същото се отнася и до библиотекаря и оптимизатора.

12. UniPascal в детайли

В тази глава се предоставя информация за реализацията на UniPascal на микрокомпютъра "Пълдин 601/601A/601M". Тя е предназначена за напреднали програмисти. Тук ще бъдат разгледани въпроси като разпределение на паметта, манипулиране с динамичната памет, вътрешното представяне на данните, извикване на процедури и др.

UniPascal е реализиран чрез така наречения смесен метод на компилация и интерпретация. Компилаторът от UniPascal е програма, написана на UniPascal и компилираща от текст на UniPascal до псевдокод (наречен Y код). След компилация този Y код се интерпретира от интерпретатор. Този метод позволява (при добре подбран псевдокод) да се генерира сравнително компактен код, за разлика от машинния код. Основният и най-важен недостатък е по-малката скорост на изпълнение в сравнение с машинния код, но този недостатък се компенсира от компактността на псевдо-кода, което е важно за компютри с малък обем оперативна памет.

12.1. Разпределение на паметта

Микропроцесорът СМ 601 (MC6800) е 8-битов с 64 килобайтово адресно пространство. Компютърът Пълдин 601/601A/601M има 64 килобайта оперативна памет и възможност за добавяне до 64K ROM (постоянна памет, памет само за четене). Това означава, че общо паметта на компютъра може да достигне 128 килобайта (за подробности виж описанието на UniBIOS). Интерпретаторът на Y код се намира в постоянната памет. Областта с адреси от $0100 до $C000 е на разположение на потребителя. Това означава, че максималният обем памет, който една програма може да използва, е 48 килобайта.

Най-общо паметта на компютъра по време на работа на програма на UniPascal изглежда така:

BIOS и текстов екран	$F000
Оперативна памет (системна)	$E000
8 банки от по 8K ROM всяка и 8K RAM (използвани за системни нужди)	$C000
Памет, използвана от други програми	$ZZZZ
Памет за динамичните променливи	$RRRR
Стек на процесора СМ 601 (MC 6800)	SP
↓ свободна памет ↑	$PPPP
Локални променливи на процедурите и глобални променливи на програмата	$YYYY
Y код на програмата на UniPascal	$XXXX
Резервирано от друга програма пространство (.cmd или .pgm)	$0100
Нулева страница ($0000..$00FF)	$0000

Преди стартирането на програма на UniPascal цялата памет от $XXXX до SP (указателя на стека) е била свободна и, освен това, адресите $RRRR и $ZZZZ са съвпадали (т.е. не е имало отделени динамични променливи за програмата). Областта между адресите $0100 и $XXXX е заета от някоя .pgm или .cmd програма, от която е била активирана програма на UniPascal. Обикновено такава програма няма (програмата се стартира, без да е активна друга програма) и затова $XXXX = $0100.

Паметта между адреси $ZZZZ и $C000 е памет, заета от резидентни програми или от системните програми. Обикновено това е област от паметта с големина от порядъка на 2-3 килобайта. Това означава, че всъщност в повечето случаи свободната памет е около 45-46 килобайта.

След активиране на програмата на UniPascal нейният код (Y код) се зарежда от първия свободен адрес - $XXXX. След това се отделя място за всички глобални променливи на програмата. Ако тя използва модули, тази операция се повтаря за всеки използван модул.

При активиране на всяка процедура, от първия свободен адрес - $YYYY се отделя място за нейните локални променливи. След приключване на нейното изпълнение заетата памет за локалните променливи се освобождава. Параметрите на процедурите се предават през системния стек. Ако процедурата е овърлейна, то освен за променливите се отделя място и за нейния код (който се прочита от външния носител - дискета). След завършване на изпълнението й се освобождава и областта, заемана от кода на процедурата.

Памет за динамичните променливи се заема и освобождава чрез стандартните прекъсвания на BIOS (int $2a и int $2b). Чрез BIOS лесно се реализира механизмът на освобождаване на памет, базиран на метода MARK - RELEASE. Получилите се в следствие на използване на стандартната процедура DISPOSE (FREEMEMWORDS) дупки се обработват отделно. Те се организират на свързан списък, като при заявка за памет се дава най-малката подходяща за целта дупка и едва ако няма такава се прави заявка към BIOS. При освобождаване на памет чрез DISPOSE/FREEMEMWORDS се появява свободна дупка в динамичната памет. Свободната дупка се обединява със съседната й, ако има такава, или се включва в списъка на свободните дупки. Ако дупката е на върха на динамичната памет, то тя се освобождава чрез основния BIOS. Така реализирани двата метода за освобождаване на памет (DISPOSE/FREEMEMWORDS и MARK-RELEASE) могат да бъдат използвани съвместно. Единствената неприятност, която може да се получи, е при използването на RELEASE да не бъде освободена паметта, заета в някоя от дупките. За да няма такъв проблем, процедурата MARK трябва да се използва само, ако не е настъпила фрагментация на динамичната памет. В такъв случай използуването на RELEASE ще възстанови състоянието на динамичната памет такова, каквото е било при използването на MARK.

Свободната памет се намира между указателя на стека (SP) и края на локалните променливи ($PPPP). При заемане на памет за динамични променливи стекът се премества надолу (намалява), а при активиране на процедура адресът $PPPP нараства. Когато те се срещнат настъпва грешка - препълване на паметта.

Използване на нулевата страница

Нулевата страница на микрокомпютъра "Пълдин 601/601A/601M" е тази част от паметта, чиито старши байт от адреса е $00, т.е. от $0000 до $00FF. От нея областта от $0000 до $007F е резервирана за операционната система (UniDOS и BIOS). От останалата част интерпретаторът на Y код използва областта от $0080 до $00DF. Останалата част ($00E0..$00FF) не се използва от интерпретатора. Областта от $00B0 до $00DF се използва от интерпретатора като работна област и Вие също можете да я

използвате като такава в подпрограмите си на асемблер. Но след връщане в интерпретатора съдържанието и може да се промени.

Таблица на разпределението на нулевата страница

- $0000..$007f - резервирана за BIOS, UniBIOS и UniDOS;
- $0080..$00af - резервирана за интерпретатора на Y код. Не трябва да бъде променяна по време на работа на програма на UniPascal;
- $00b0..$00df - работна памет за интерпретатора на Y код. Може да бъде използвана за работна и в подпрограми на асемблер;
- $00e0..$00ff - не се използва от интерпретатора на Y код.

12.2. Вътрешно представяне на данните

Във всички случаи простите променливи (тези които не са елементи на масив или запис) се разполагат на граница на дума (2 байта) и са с големина, кратна на 16 бита, независимо от това колко бита са необходими за представянето й. Това правило важи и за елементите на масиви или записи, като изключение се прави само при непосредствено указване на пакетиране (низовете са пакетирани по премълчаване).

12.2.1. Непакетирани променливи

При представянето на променливи от прост тип младшият байт в думата предхожда старшия, въпреки че на този процесор (СМ 601) това е обратно. Изключение правят променливите от тип указател, които се представят в нормалния за процесора вид (старши-младши).

За променливите, за които са достатъчни не повече от 8 бита, се използват целите 16 бита. Това означава, че проста променлива, обявена от тип shortint, byte ..., всъщност заема два байта и се интерпретира като променлива от тип integer, word, ... Ако е включена проверката за граници {$R+}, тя се извърша както трябва и това, че се използват 16 бита е 'прозрачно' за програмиста. Но ако се използват някои от специалните процедури, като например MOVE, BLOCKREAD, FILLCHAR, ... трябва да се внимава старшият байт да е винаги 0 (или $FF при отрицателна стойност). Например:

```
function GetByte(var f: file): byte;
  var b: byte;
begin
  b:= 0; { нулиране на старшия байт }
  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;{ връщаме само младшия байт  }
end { GetByte };
```

Ако е изключена проверката на границите, при даването на стойност също може да се получи по-голяма стойност, отколкото разрешава типа (ако е включена, ще получите грешка по време на изпълнение). Използването на стандартните процедури

READ и READLN от текстов файл не поставят такива проблеми, т.е. старшият байт ще бъде такъв, какъвто е необходим.

Променливи от изброим тип

Променливите от изброим тип се представят като целочислени променливи с граници 0..N-1, където N е броят на константите от изброимия тип.

Логически променливи

Както вече беше казано, логическият тип може формално да се разглежда като изброим тип, дефиниран така:

```
BOOLEAN = (FALSE, TRUE);
```

а това означава, че 0 представя FALSE, а 1 - TRUE.

Целочислени променливи

Променливите от тип LongInt се представят чрез четири байта. Първият байт е младши, следващият байт е следващ по старшинство и последният (четвърти) байт е старши.

Останалите целочислени типове **INTEGER**, **CARDINAL** и техните поддиапазони се представят чрез два байта - първо младшият след него старшият. Числата със знак при целочислените типове се представят в допълнителен код.

Променливи от тип CHAR

Променливите от тип CHAR се представят със стойности от 0 до 255, като стойността им съответствува на разширения ASCII код.

Реални променливи

Данните от тип REAL заемат четири байта и представянето им съответства на IEEE стандарта. Тъй като е възприета обратна наредба на байтовете, то и тук представянето е с обратен порядък на байтовете. Знакът се разполага в бит 7 на старшия байт (3), експонентата се разполага в битове 6 - 0 на байт 3 и в бит 7 на байт 2, а мантисата заема останалото пространство на байт 2 и байтове 1 и 0.

Нека с **e** означим порядъка (exponent), с **m** - мантисата (mantissa) и със **s** - знака (sign). Тогава стойността (value) е:

(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 поддържа:

- денормализираните стойности (3). Получава се 0.0;
- представянето на безкрайност (4). Получава се препълване;
- невалидните стойности (5). Непредсказуем резултат.

12.2.2. Пакетирани променливи

В UniPascal пакетирането оказва съществено влияние на представянето на променливите, масивите и структурите. Пакетиране се предприема, само ако то е явно указано и ако е възможно.

Типът T ще наричаме пакетиран, ако е указано пакетирането му и освен това, ако в него има поне един пакетируем елемент.

Типа T ще наричаме пакетируем, ако е изпълнено едно от следните условия:

- Типът T е дискретен тип (целочислен, логически, символен, изброим или диапазонен) и освен това минималната и максималната му стойност лежат (и двете едновременно) в един от двата интервала [0..255] или [-128..127];
- Типът T е структуриран, като всички негови елементи (ако е масив) или всички полета (ако е запис) са пакетирани и пакетируеми. Освен това е указано пакетиране за този тип, т.е. той е и пакетиран.

От тези две определения следва, че един тип може да бъде пакетиран, но непакетируем. Непакетируемостта се въвежда, за да се обясни кои масиви могат да бъдат пакетирани и кои не и за да се поясни начина на пакетиране при записите. Ако елементите на масив са непакетируеми, то указанието за пакетиране (ключовата дума PACKED) се игнорира. Указанието за пакетиране се игнорира също и ако всички полета на даден запис са непакетируеми. За непакетируемите елементи се отделя място на граница на дума. Затова, ако те са елементи на масив, то той не е пакетиран. А ако е поле от запис, то адресът му е изравнен на граница на дума, останалите полета са пакетирани, ако са пакетируеми.

Примери:

packed array [0..5] of char;

е пакетиран и пакетируем масив;

packed array [0..5] of integer;

е непакетиран масив;

packed record x, y: shortcard; end;

е пакетируем (и пакетиран) запис, т.е. може да се използва като елемент на пакетиран масив (или запис);

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

е пакетиран но непакетируем запис, т.е. ако се използва като елемент на масив, то указването на PACKED ще бъде игнорирано за масива.

Освен това пакетируемостта оказва съществено влияние на начина, по който се пакетират записи. Пакетируемите полета на запис могат да започват на граница на байт, докато непакетируемите полета са изравнени на граница на дума.

По-долу се дава представянето на стандартните типове при пакетиране.

Изброим тип

Стойностите на константите от изброим тип се представят като цели числа в диапазона 0..N-1, където N е броят на константите от изброимия тип. Обикновено представянето заема един байт и в такъв случай е пакетируемо, тъй като броят на константите рядко е по-голям от 256.

Логически тип

Тъй като е необходимо представяне на стойностите 0 и 1, то е достатъчен само един бит, но се използва един байт. Типът е пакетируем.

Целочислен тип

Ако минималната и максималната стойност са в границите:

- 0..255 се представят с един байт (беззнаково). Типът е пакетируем;
- -128..127 се представят също с един байт (със знак). Стойности се представят в допълнителен код. Типът е пакетируем;
- във всички останали случаи са непакетируеми.

Типовете CHAR и BYTE

Тези типове са пакетируеми и се използва само един байт.

Реален тип

Реалният тип е непакетируем.

Стандартен тип STRING

Този тип е винаги пакетиран, но непакетируем. За него се отделя област от памет с размер N + 1 байта, където N е константата, указана в декларацията. Тъй като типът е непакетируем, то винаги се предприема изравняване на граница на дума и следователно променливи от тип string[7] и string[6] ще заемат еднакво количество памет.

Първият байт (съответстващ на нулевия елемент) е дължината на текущата стойност на низа. Ако дължината, е по-малка от максималната дължина на низа, използват се само първите K+1 байта от представянето (K - текущата дължина). Например 'Test' има представяне: $04, $54, $65, $73, $74.

12.3. Връзка с подпрограми на асемблер

Връзка между програма на UniPascal и подпрограма на асемблер (машинен код) се реализира само чрез параметрите. Те се предават през стека, като се намират в обратен ред на обявяването си, т.е. на върха на стека се намира последният параметър, под него е предпоследният и т.н. В момента на активизиране на асемблерската подпрограма на върха на стека (над параметрите) се намира и адресът на връщане (return address).

За да може интерпретаторът на Y код да различава асемблерските подпрограми от тези на Y код, необходимо е всяка подпрограма на машинен код да започва с два байта съдържащи нула.

Подпрограмата е длъжна да 'извади' от стека всички параметри, които й се подават. Връщането към интерпретатора на Y код може да бъде извършено по два начина: като се използва адреса за връщане, намиращ се в стека; като се направи преход към адрес $BEFE (абсолютен адрес). Ако се използва преход, необходимо е да бъде изваден от стека адресът на връщане.

Ако подпрограмата е функция, тя трябва да върне на върха на стека резултата си. Резултатът на функцията трябва да е съобразен с представянето на данните, което

използва UniPascal, т.е. трябва да бъде с наредба на байтовете младши-старши (на върха на стека е младшият, а под него старшият байт), освен ако се връща указател. Освен това размерът на резултата трябва да бъде кратен на дума, т.е. ако се връща байт трябва да бъде върната дума със старши байт 0 (или $FF, ако е отрицателна стойност).

Всеки параметър също се предава съобразно представянето на данните в UniPascal. Най-младшият байт е на върха на стека, а по-старшите байтове са под него. Независимо от типа на параметъра той се предава с дължина, кратна на дума. Това означава, че ако се очаква параметър байт, то неговата стойност е на върха на стека, а намиращият се под него старши байт на думата може да бъде игнориран. Освен това, всеки параметър в зависимост от типа и вида си се предава в съответствие със следните правила:

Ако параметърът се предава по стойност (или ако е параметър-константа), то:

- всички параметри с големина до две думи (4 байта) се предават директно в стека;
- за всички останали се предава адрес. Модифицирането на стойността е нежелателно.

Ако параметърът се предава по адрес (VAR параметри), то на върха на стека се предава адресът на параметъра.

Параметрите от тип STRING се предават по специален начин:

- VAR параметри от тип STRING без описател за дължина се предават по следния начин: в стека се предава адресът на променливата и след него (т.е. отгоре - на върха на стека) се намира спецификаторът за дължина на фактическия параметър. Пример: procedure MyASM(var s: string); се активира чрез оператора MyASM(sss); където var sss: string[77]; тогава в стека над адреса на променливата се намира дума със съдържание 77;
- VAR параметри от тип STRING с описател за дължина, се предават по нормалния (както за другите типове) начин;
- CONST параметри от тип STRING (със или без описател за дължина). За тези параметри се предава адресът на параметъра. Ако фактическият параметър е от тип CHAR, компилаторът е генерирал код за даване на стойността на вътрешна променлива от тип STRING[1], така че да бъде предаден адрес на променлива от тип STRING, а не от тип CHAR;
- Параметри-значения от STRING (със или без описател за дължина) се предават, както CONST параметрите, т.е. предава се адреса им. Ако фактическият параметър е от тип CHAR се предава съдържанието му като адрес (т.е. първо старшия после младшия), чийто младши байт е нейният ASCII код, а старшият байт е 0. Адресите, чийто първи байт е нула, са адреси на клетки от нулевата страница, а тя не се използва за разполагане на променливи в нея. По този начин, ако вашата процедура получи нормален адрес, тя може спокойно да го обработи, а ако получи адрес, чийто старши байт е 0, то това означава, че е бил предаден фактически параметър от тип CHAR и стойността му се намира в младшия байт. Вие винаги трябва да предполагате, че параметърът може да бъде от тип CHAR. Ако ви е неудобно да обработвате такива параметри, опишете ги като CONST, тогава подпрограмата ви на асемблер ще се опрости за сметка на увеличаване на кода, генериран за UniPascal програмата (с 3 байта на всеки предаден фактически параметър от тип CHAR).

12.4. Използване на етикета EXIT

Специфичният за UniPascal етикет EXIT се появява в синтактичното описание на блок, както следва:

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

Изпълнението на стандартната процедура RETURN или EXIT (процедурата EXIT, а не етикета) в тяло на блок, в който няма етикет EXIT, предизвиква принудително завършване на работата на блока, т.е. управлението се предава на края на блока.

Наличието на етикет EXIT в тялото на блока изменя семантиката на стандартните процедури EXIT и RETURN. Тогава те предават управлението на етикетирания с EXIT оператор от този блок. Процедурите RETURN и EXIT, появяващи се след етикета EXIT, имат обикновената си семантика. Операторите, намиращи се след етикета EXIT до края на блока, изпълняват функция на завършваща част на този блок. Тези оператори се изпълняват винаги преди излизане от съответната процедура.

В случаите, когато етикетът EXIT се намира в инициализиращата част на модул, той има малко по-различно, но близко до това значение (виж описанието на реализационната част на модул). Пример за използването му в модул е даден в модула STACK, намиращ се на дистрибутивната дискета.

12.5. Овърлейни процедури

Овърлей на процедурите се реализира без никакви допълнителни (освен указването на ключовата дума SEGMENT) действия от страна на програмиста. Използването на овърлейни процедури се налага, ако паметта на компютъра не достига да побере едновременно кода и данните на програмата.

Овърлейните процедури в UniPascal се зареждат в паметта на микрокомпютъра, когато се активират. Кодът на процедурата остава в паметта, докато процедурата е активна или докато е активна активирана от нея процедура. След завършване на работата паметта, заета от тях, се освобождава (паметта заета за код на процедурата се освобождава). При ново активиране процедурата се зарежда отново.

За да може във всеки момент да бъде заредена овърлейна процедура, необходимо е файлът с кода на програмата (или модула) да остане отворен, докато програмата е активна. Оставянето на отворен файл дава следните отражения върху работата на операционната система:

- броят на файловете, които могат да бъдат отворени едновременно от програмата, се намалява с 1 за всеки файл съдържащ код на овърлейни процедури;
- дискетата, върху която е отворен файла, не трябва да се изважда от флопидисковото устройство, освен ако имате само едно и използвате възможността на UniDOS да работи с две логически устройства върху едно физическо. В такъв случай не трябва да сменяте дискетата, която поставяте за устройство A: или B: (т.е. трябва да ползвате една и съща дискета за съответните устройства). Замяна на дискета, върху която има отворени файлове, може да доведе до унищожаване на информацията върху другата дискета и/или и върху двете.

При свързването на подпрограми на асемблер към овърлейни процедури трябва да се внимава да не се свързват подпрограми, които записват стойности, трябващи им при повторно извикване. Подпрограмата може да бъде изхвърлена и по-късно заредена отново в паметта, тогава съхранените стойности ще бъдат загубени, т.е. те ще бъдат в началното си състояние, записано от асемблера. Например, нека сме написали подпрограма на асемблер, която попълва някакъв буфер, за да го отпечата на екрана/принтера или за да го запише във файл. Тя изпраща съдържанието на буфера по реалното му предназначение, едва след като той се напълни или ако бъде предизвикано явно записване на буфера.

```
; procedure SendChar(ch: char);
SendChar ent                    ; подпрограмите на асемблер, които ще бъдат
         dw      0                ; свързвани с UniPascal, започват с 0.
         ins
         ins                    ; изхвърляне адреса за връщане в интерпретатора
         pula                   ; изваждане стойността на параметъра
         ins                    ; и игнориране на старшия му байт
         ldx     buff_ptr       ; записване на символа в буфера
         staa    x, 0
         inx
         stx   buff_ptr
         cmpa #13               ; символа <cr> ли е?
         beq     SendIt         ; ако да, форсиране изпращането на буфера
         cpx     #buff_end ; пълен ли е буфера?
         beq     SendIt         ; да, записване / изпращане / обработка
         jmp     $befe          ; връщане в интерпретатора на Y код (UniPascal)

; procedure SendBuff;

SendBuff ent                    ; тук започва подпрограмата за
         dw      0                ; изпращане на буфера
         ins
         ins
SendIt
         ldx     #buffer        ; буфера може да е празен
         bra     test_x         ; затова първо проверяваме дали не е такъв
loop
         ; ... записване / изпращане / обработване на един символ
         inx
test_x  cpx      buff_ptr      ; края на буфера?
         bne     loop           ; цикъл докато свършим
         ldx     #buffer        ; задаваме празен буфер
         stx   buff_ptr ; (започване от начало)
         jmp     $befe          ; връщане в интерпретатора на Y код (UniPascal)

buffer  ds     1024          ; 1 килобайт буфер
buff_end
```

Забележка: В синтаксиса на UniCross не съществува директива ENT (тя е директива за UniASM), в неговия синтаксис тази директива се заменя от директивата public, написана на отделен ред.

Тези асемблерски подпрограми могат да се свържат с нормална подпрограма. Ако подпрограмата е овърлейна, това не бива да се прави (тъй като съдържанието на буфера ще бъде загубено), освен ако сте абсолютно сигурни, че подпрограмата на UniPascal ще завърши изпълнението си, само ако буферът е изпратен. Например може да напишете:

```
segment procedure some_proc;
begin
  repeat
    { ... обработка на някаква информация }
    SendChar(ch);
  until OK;
exit:
  SendBuff;
end { some_proc };
```

В конкретния случай е възможно да поставите асемблерски подпрограми в овърлейна процедура. В общия случай това е възможно, само ако след завършване на работата на подпрограмата на UniPascal асемблерските подпрограми не се нуждаят от никаква междинна информация, която би трябвало да се съхрани до следващото им активиране. При следващото си активиране асемблерските подпрограми ще бъдат прочетени (заедно с подпрограмата на UniPascal) от диска и следователно ще бъдат в начално състояние (все едно, че не са били активирани нито веднъж).

12.6. Модули и техните версии

При използването на модули тяхната реализационна част е или поне би трябвало да бъде изцяло скрита от потребителите модули или програми. Ако това е така (използващите даден модул програми или модули не предполагат конкретна реализация и зависят само от описателната част), то е възможна коренно различна реализация на описаните в interface частта обекти. Прекомпилация на реализационната (implementation) част на един модул не налага прекомпилация на модулите, които го използват (и поради това нейното изменение не създава проблеми).

Прекомпилацията на описателната (interface) част на един модул винаги изисква прекомпилиране на всички програми и модули, които го използват. Това изискване се налага, тъй като след като се сменя описанието на даден модул, то той вече не е същия и тези модули, които го използват, трябва да 'научат' за новите му свойства. Например, ако се добави нов параметър към някоя процедура или ако цялата процедура се премахне, то използващият я модул не би могъл да продължи да я използва по същия начин или въобще не би могъл да използва тази процедура.

Понякога новото описание на модула съвпада със старото и различията се изразяват само в добавени нови възможности (константи, типове, променливи, процедури или функции). В такъв случай модулите, които са го използвали в стария му вид, могат да продължат да го използват в новия му вид (новата версия), тъй като описанието на старата част не се е изменило и следователно тя е съвместима с него. Пример: нека модул за работа с магнитни ленти е описан така (от реализационната част не се интересуваме):

```
interface unit Tape;
  procedure rewind(No: shortcard);
    {- пренавиване на лентата }
  procedure write_TM(tNo: shortcard);
```

```
    {- записване на лентов маркер }
  procedure write_buff(No: shortcard; const buff; size: natiral);
    { записване на буфер (buff) с дължина size }
  procedure read_buff(No: shortcard; var buff; var size: natural);
    {- прочитане от лентата на блок в buff, дължината е в size }
end { Tape };
```

След като този модул е заработил и се използва от няколко програми, чиито изходен текст може и да е неизвестен, ако са писани от други програмисти, може да се наложи да се правят поправки и изменения в реализационната част на модула, за да се поддържат друг вид лентови устройства. Докато изменението се прави само в реализационната част на модула, програмите които го използват, ще работят и с прекомпилираната реализация. Но може да се наложи добавянето на нова процедура, например за четене на блок в обратна посока:

```
procedure read_back(tape_no: shortcard; var buff; var size: natural);
  {- прочитане от лентата на блок в обратна посока }
```

За такава промяна в модула е необходима промяна в неговата описателна част. Но ако се смени описанието на модула, с добавянето на тази процедура, ще трябва да се прекомпилират всички програми, които го използват (освен ако специално заради тези от тях, чиито изходен текст е неизвестен, се пази стария вариант и ако стария вариант може да работи с новия тип лентови устройства).

В UniPascal е възможно да се добавят нови възможности (декларации на обекти) към описателната част на модула (както в дадения пример), без да се прекомпилират програмите, които вече използват този модул. Това се прави, като се укаже номер на версията на модула, оградена в скоби след идентификатора му в описателната част (реализационната част може да се прекомпилира без това да налага прекомпилация на други модули). Например:

```
interface unit Tape (2);
  procedure rewind(tape_no: shortcard);
    {- пренавиване на лентата }
  procedure write_TM(tape_no: shortcard);
    {- записване на лентов маркер }
  procedure write_buff(tape_no: shortcard; const buff; size: natiral);
    { записване на буфер (buff) с дължина size }
  procedure read_buff(tape_no: shortcard; var buff; var size: natural);
    {- прочитане от лентата на блок в buff, дължината е в size }
  {=== добавено във версия 2: ===}
  procedure read_back(tape_no: shortcard; var buff; var size: natural);
    {- прочитане от лентата на блок в обратна посока }
end { Tape };
```

Всички версии на даден модул са съвместими в строго възходящ ред, т.е. версия 2 е съвместима с версия 3, версия 3 - с 4 и т.н., но версия 7 не е съвместима с 6 или с версия със същия номер 7, но компилирана отново. Не е необходимо номерата на версиите на модула да бъдат последователни, т.е. след версия 1 може да следва версия 10, след нея версия 50 и т.н. Компилаторът не проверява дали наистина новата версия е реално съвместима със старата. Ако в горния пример процедурата READ_BACK бъде описана първа, модулът ще бъде реално (на практика) несъвместим със предната си версия.

Ако новата версия на един модул е компилирана като съвместима със старата, но реално е несъвместима, то програмите и модулите, компилирани със старата версия ще имат непредсказуемо поведение и ще дават непредсказуеми резултати при работа с новата версия на модула.

Две версии на един модул са реално съвместими, ако **всички** нови обекти (константи, типове, променливи, процедури и функции) са добавени (физически в изходния текст) **в края** на описателната част на модула. Ако **поне един** от обектите е поставен **пред** последния от дефинираните в старата версия обекти, то новата версия е на практика несъвместима със старата. Отново повтаряме, че компилаторът **не проверява** за реалната съвместимост на новата версия на модула със старата и ако новата несъвместима версия се използва от програми, компилирани със старата, то тяхното поведение е **непредсказуемо**.

И още нещо за съвместимостта. Ако новата версия на един модул е реално съвместима със старата и е компилирана с нов по-голям номер на версия, то програми, използващи старата версия, ще работят правилно и с новата версия. Програма, използваща модул, се свързва реално с него едва при стартирането. При това в програмата е записано името и версията на модула, както и датата и часа, в който този модул е бил компилиран. Свързването на програма с даден модул, който тя използва, може да стане, ако има модул под това име, под което програмата го използва и е изпълнено едно от следните две условия (ако не е изпълнено нито едно от двете условия се счита, че използваният от програмата модул не е намерен):

- версията, датата и часа на модула съвпадат с тези записани в програмата. Така се осигурява идентичност на модула, т.е. че използваният и наличният модули са един и същ модул;
- версията на модула е по-голяма от тази, която е записана в програмата, и моментът (датата и часа, взети заедно) на компилиране на модула е по-късен, от този, който е записан в програмата. Така се осигурява идентичност на модула в различните му версии, т.е. че наличният модул е нова версия на използвания от програмата модул.

Тъй като времето е важно при компилация на модули, необходимо е в операционната система (от която компилаторът получава това време) да бъдат установени вярната дата и час, особено ако предстои компилиране не описателна част на модул.