AIX XL Pascal/6000

SC09-1757–00

Language Reference

Version 2.1

Note!

Before using this information, and the product it supports, be sure to read the general information under “Notices” on page vii.

Second Edition (December 1993)

This edition applies to Version 2, Release 1, of IBM AIX XL Pascal Compiler/6000 (Program 5765–245), and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers’ comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue
North York, Ontario, Canada. M3C 1H7

You can also send your comments by facsimile to (attention: RCF Coordinator), or you can send your comments electronically to IBM. Please see “Communicating Your Comments to IBM” for a description of the methods. This page immediately precedes the Readers’ Comment Form at the back of this publication.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

Trademarks and Service Marks

AIX	IBM	IBMLink
PROFS	RISC System/6000	RT
RT PC		

ANSI	American National Standards Institute

Chapter 1. Introduction

This book describes the IBM* AIX* XL Pascal Compiler/6000 language: it contains reference material to complement the *User's Guide for IBM AIX XL Pascal Compiler/6000*, SC09–1756. It describes in detail the program structure, declarations, constants, data types, variables, expressions, statements, and routines in XL Pascal.

The exceptional (XL) family of compilers provides consistency and high performance across multiple programming languages by sharing the same code-optimization technology. The XL Pascal Compiler/6000 is an optimizing compiler for the Pascal language for AIX Version 3 for the RISC System/6000* operating system. It allows invocation of routines written in other programming languages, and the creation of routines that can be invoked by programs written in other languages. It also provides detailed compile-time and runtime diagnostics, in addition to error recovery and debugging facilities.

The XL Pascal language is a full implementation of the American National Standards Institute (ANSI**) standard for Pascal ANSI/IEEE 770X3.97–1983 and incorporates IBM VS Pascal Release 1 functions, selected features of VS Pascal Release 2, and RT PC* VS Pascal. VS Pascal is an IBM licensed program.

Note: For brevity, both the IBM AIX XL Pascal language and the IBM AIX XL Pascal Compiler/6000 are referred to throughout this manual as *XL Pascal*. The compiler and the language are distinguished where necessary. The AIX Version 3 for the RISC System/6000* operating system is referred to as *AIX*.

Who Should Use This Book

You should have previous experience writing or maintaining Pascal application programs. If you have no prior Pascal knowledge or lack experience with a high-level programming language, you can obtain any of the tutorial-style Pascal books commercially available.

How to Use This Book

This manual is organized so that you can read each part independently as a reference source for a particular feature or function of XL Pascal. If you are unfamiliar with Pascal, you may first want to read Chapter 2, "The XL Pascal Program Elements," and Chapter 3, "Structure of XL Pascal Programs." After that, use the Table of Contents to find detailed information about a specific topic.

How This Book Is Organized

Each of the following chapters contains a major concept or feature of the XL Pascal language:

Chapter 1, "Introduction," is the chapter you are reading now. It introduces the major features of the language, the standards it conforms to, and shows where to get more information.

Chapter 2, "The XL Pascal Program Elements," introduces some basic elements of XL Pascal programs.

Chapter 3, "Structure of XL Pascal Programs," describes the two types of compilation units (the program unit and the segment unit) and how XL Pascal programs are structured.

Chapter 4, "Declarations," describes in alphabetical order the different types of XL Pascal declarations.

Chapter 5, “Constants,” describes XL Pascal constants.

Chapter 6, “Data Types,” provides a chart of XL Pascal data types in functional order. Descriptions of each data type follow in alphabetical order.

Chapter 7, “Variables,” describes the XL Pascal classes of variables and explains how they are referenced.

Chapter 8, “Expressions,” explains how to use XL Pascal expressions to combine data according to specific computational rules.

Chapter 9, “Statements,” provides a chart of XL Pascal statements. Explanations of each statement follow in alphabetical order.

Chapter 10, “Routines,” describes the two categories of XL Pascal routines (procedures and functions) and provides tables of the XL Pascal predefined routines in functional order. Explanations of each predefined routine follow in alphabetical order.

Chapter 11, “Compiler Directives,” describes the XL Pascal compiler directives that control several compiler options and features.

Appendix A, “Summary of XL Pascal Language Modes,” summarizes the differences between the two available language levels in XL Pascal.

Appendix B, “Predefined Identifiers in XL Pascal,” summarizes the XL Pascal predefined identifiers for constants, data types, routines, and variables.

How to Read Syntax Diagrams

The following conventions are used in syntax diagrams:

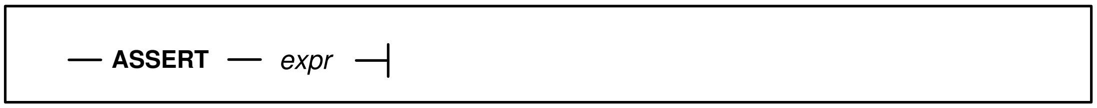
- Keywords and reserved words are in bold uppercase letters (for example, **VAR**, **BEGIN**, and **END**). They can be written in uppercase or lowercase, but they must be spelled exactly as shown.
- Variables, expressions, or identifiers that you supply are in all lowercase italics (for example, *label_dcl*).
- Enter punctuation marks, parentheses, arithmetic operators, and other nonalphabetic symbols as they are shown in the syntax.

Read syntax diagrams from left to right, from top to bottom, following the path of the line. In the diagrams, syntax is described using the following scheme:

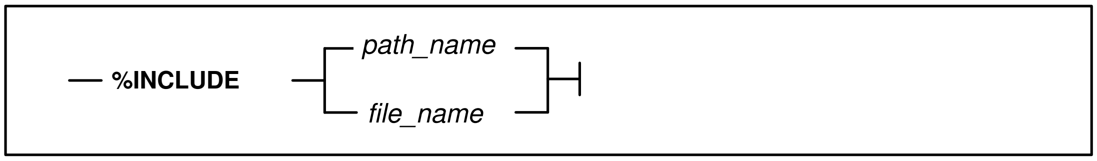
- The following symbol indicates the beginning of a diagram: ──
- The following symbol indicates that the syntax continues on the next line: ─▶
- The following symbol indicates that the syntax is continued from the previous line: ▶──
- The following symbol indicates the end of the diagram: ─┤
- Syntactical units that are not complete statements start with the following symbol: ▶──
- Syntactical units that are not complete statements end with the following symbol: ─▶

Required and Optional Items

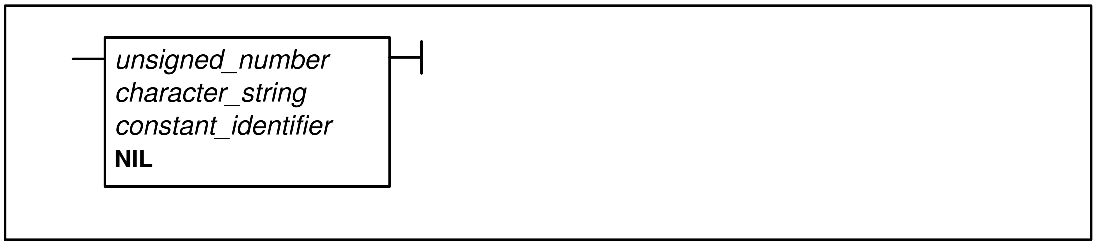
- Required items appear on the horizontal line (the main path).



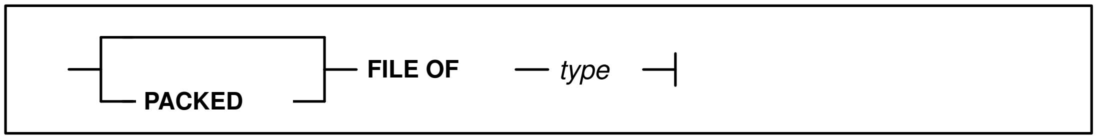
- Branching shows two paths through the syntax.



- If you must choose one of three or more items, they appear in a multiple choice box on the main path.

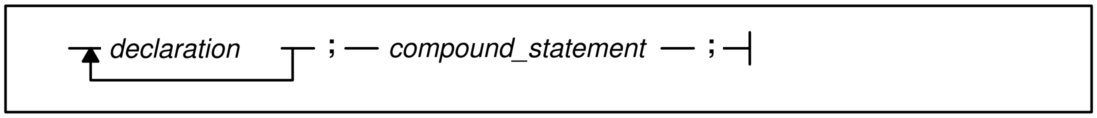


- Optional items appear on the lower line of a branched path. The upper line is empty, indicating that you need not write anything for this syntax item.

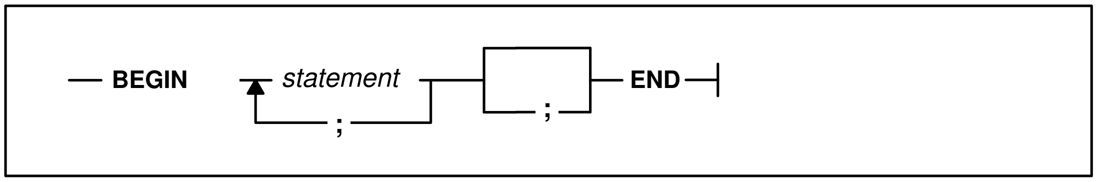


Repeatable Items

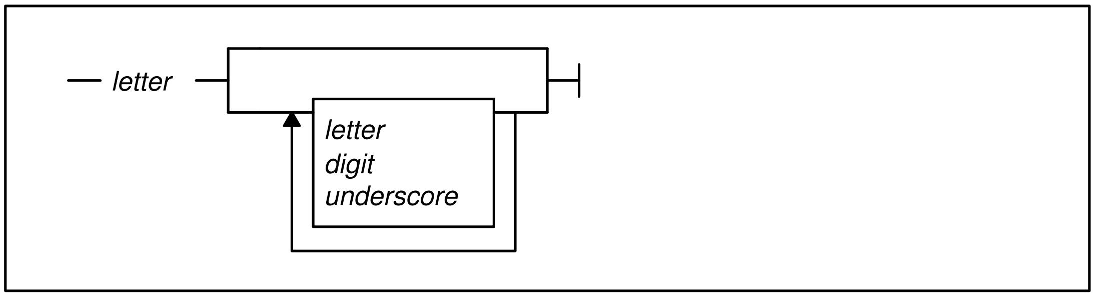
- An arrow returning to the left below a line shows that you can repeat items.



- Punctuation on a repeat arrow indicates that you must place it between the repeated items.

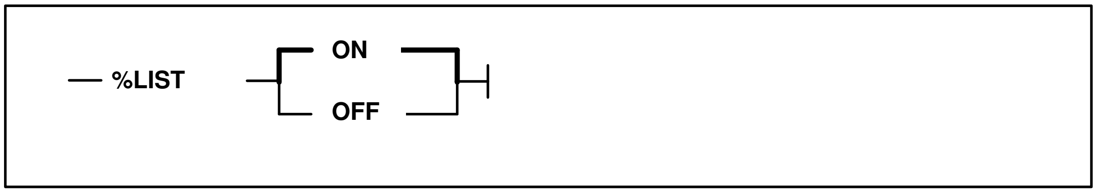


- A repeat arrow below a multiple choice box indicates that you can choose one or more items in the box, but you must choose at least one.

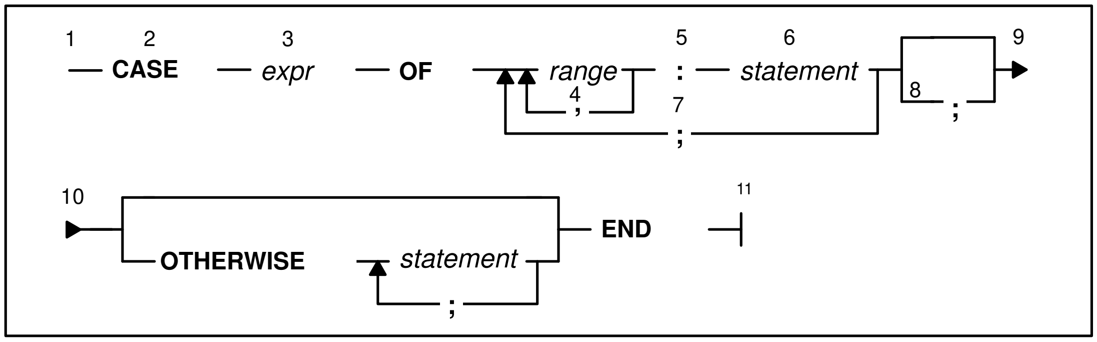


Default Items

- A heavy line is the default path.



Example



The diagram is interpreted as follows:

1. This is the start of the diagram.
2. Type the keyword **CASE**.
3. Type a valid expression followed by the word **OF**.
4. Type at least one range. For more than one range, separate each by a comma.
5. Type the colon symbol (:).
6. Type a valid statement.
7. Type the semicolon symbol (;).
8. This path is optional.
9. The diagram is continued at 10.
10. The diagram is continued from 9.
11. This is the end of the diagram.

The following **CASE** statements conform to the syntax shown in the syntax diagram:

```
CASE a_card.r OF
   ace:
      points := 11;
   two..ten:
      points := ORD( a_card.r ) + 1;
   OTHERWISE
      points := 10
END
```

```
CASE s OF
   triangle:
      area := 0.5 * side * base;
   rectangle:
      area := sidea * sideb;
   circle:
      area := 3.14159 * SQR( radius )
END
```

```
CASE s OF
   triangle:
      area := 0.5 * side * base
END
```

A Note about Examples

Examples in this book are written in a simple style. They do not attempt to conserve storage, check for errors, achieve fast run time, or demonstrate all possible uses of a language element.

Pascal Industry Standards

XL Pascal complies with the ANSI standard (commonly referred to as ANSI–83), defined in the document *American National Standard Pascal Computer Programming Language*, ANSI/IEEE 770X3.97–1983.

This standard is adopted by International Standards Organization (ISO) and Federal Information Processing Standards (FIPS). It implies conformance to the following standards:

- International Standards (ISO) 7185–1983 (Level 0), Programming Languages, Pascal
- Federal Information Processing Standard, (FIPS) PUB 109, Pascal

In this book, *standard Pascal* or *standard mode* refers to the ANSI–83 standard.

XL Pascal Extensions

The XL Pascal language comprises:

- ANSI Pascal 1983. This is the full ANSI–83 Pascal language.
- XL Pascal extensions are primarily (though not exclusively):
  - Extensions specified in the IBM VS Pascal Compiler Release 1
  - Selected functions of Release 2 of IBM VS Pascal Compiler
  - Selected features from RT PC* VS Pascal added to make the language more usable in the AIX Version 3 Operating System.

VS Mode Extensions

The VS mode of the XL Pascal Compiler comprises the extensions derived from VS Pascal.

Those elements of VS Pascal that do not translate into the AIX environment have been left out. It is possible to mix modes in a program, but each separate compilation unit must be in a single mode.

Note: Standard mode XL Pascal is a subset of VS mode. All of the features of standard mode XL Pascal described in this manual also function in VS mode.

A summary list of all the features of XL Pascal for both language modes is in Appendix A, “XL Pascal Language Modes.”

Related Publications

IBM Publications

- *User’s Guide for IBM AIX XL Pascal Compiler/6000*, SC09–1756, describes the IBM AIX XL Pascal Compiler/6000, and explains how to compile, link, and run programs written in XL Pascal. It also describes how to use input and output (I/O) facilities and storage, and how to do interlanguage calls.
- *VS Pascal Language Reference*, SC26–4320, provides definition of the VS Pascal programming language and its syntax.
- *VS Pascal Application Programming Guide*, SC26–4319, shows how to use the VS Pascal compiler and explains how to compile, link-edit, run, and debug VS Pascal programs.
- *AIX Version 3.2 Topic Index and Glossary*, GC23–2201, provides a glossary of terms used in AIX and RISC System/6000 publications. It also contains a list of some topics in the AIX and RISC System/6000 library and the books in which those topics are discussed.

Non-IBM Publications

- *American National Pascal Computer Programming Language*, ANSI/IEEE 770X3.97–1983
- *International Standards Organization Programming Language Pascal*, (ISO) 7185–1983 (Level 0)
- *Federal Information Processing Standards Publication Pascal*, (FIPS) PUB 109
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754–1985

Valid AIX XL Pascal Programs

The *Language Reference for IBM AIX XL Pascal Compiler/6000* defines the syntax, semantics, and restrictions for writing valid XL Pascal programs. The compiler finds most violations of the XL Pascal language rules, but some may not be found. Programs containing undiagnosed violations are not valid XL Pascal programs.

Typographical Conventions

Type style highlights important terms and features of the compiler. The following kinds of information are distinguished by different typographical conventions.

Keywords and Reserved Words

Predefined identifiers, which you must write exactly as presented, are in **BOLD CAPITALS**. When used generically, these words are in lowercase. For example, the reserved word **TYPE** is in capital letters, but general references to data type are in lowercase.

New Terms

When a term is used for the first time, it is in *italics*, often followed by a brief definition. All of these terms and definitions are in the *AIX Version 3.2 Topic Index and Glossary*.

Multibyte Characters

VS mode XL Pascal permits the use of multibyte character set (MBCS) characters:

- A boldface capital **D** represents one multibyte character.
- A boldface capital **B** represents one MBCS blank.

Chapter 2. The XL Pascal Program Elements

Pascal is a high-level general purpose programming language that relies on block structures. A structured program is a hierarchy of routines, with each routine having a single entry point and a single exit point.

XL Pascal provides the following basic elements in most Pascal programs. This chapter explains the conventions governing the use of these elements:

- Characters
- Identifiers
- Basic symbols
- Comments
- Literals

The XL Pascal Character Set

The following characters have an order known as a *collating sequence*, which determines the comparison status of the items in a character set for a system. XL Pascal uses ASCII (American National Standard Code for Information Interchange) to determine the ordinal sequence of characters. The *special characters* are the nonalphanumeric characters used in Pascal code.

Letters				**Digits**	**Special Characters**	
A	N	a	n	0		Blank
B	O	b	o	1	$	Dollar sign
C	P	c	p	2	%	Percent sign
D	Q	d	q	3	'	Single quotation mark
E	R	e	r	4	()	Parentheses
F	S	f	s	5	*	Asterisk
G	T	g	t	6	+	Plus sign
H	U	h	u	7	,	Comma
I	V	i	v	8	−	Minus sign
J	W	j	w	9	.	Decimal point/period
K	X	k	x		/	Slash
L	Y	l	y		:	Colon
M	Z	m	z		;	Semicolon
					<	Less than
					=	Equal sign
					>	Greater than
					_	Underscore
					@	At
					^	Caret
					|	Vertical bar
					&	Ampersand
					~	Tilde
					[]	Brackets
					{}	Braces
					#	Number sign

Related Information

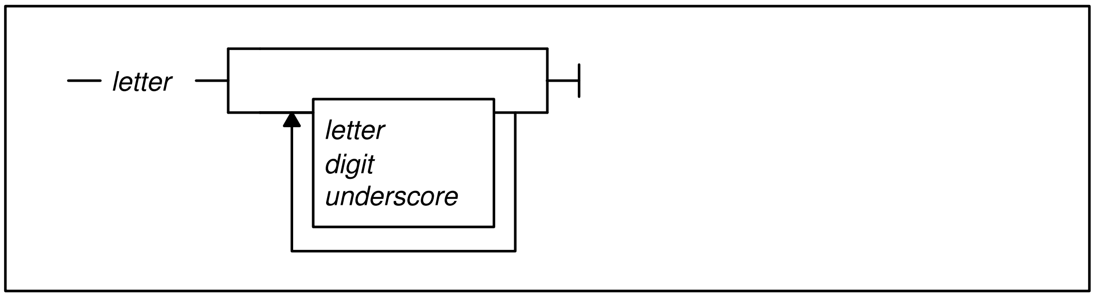
A table showing the full ASCII character set is in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

Identifiers

The *identifiers* are names for the following language constructs of Pascal. Identifiers can be internal or external to a program.

- Constants
- Data types
- Files
- Functions
- Procedures
- Variables
- Program names
- Segment names
- Label names
- Enumerated values

Syntax



Parameters

letter	is A through Z, a through z, and in VS mode, $
digit	is 0 through 9.
underscore	is _ (VS mode only).

Description

XL Pascal permits identifiers of up to 256 characters, which is the maximum length of the source line. The first character of an identifier must be a letter. Valid characters for identifiers are the letters a through z and A through Z, and the digits 0 through 9.

The space character is not valid within an identifier.

XL Pascal makes no distinction between lowercase and uppercase letters within an identifier name, because the compiler folds all identifiers in a source program to lowercase unless they are in a quoted string. Specifying the **–U** compiler option prevents case folding.

You must declare every identifier before using it.

Note: Declaring a name that is already a predefined identifier overrides the predefined use of the identifier. You cannot declare reserved words as identifiers.

Identifiers in VS Mode

In VS mode, the dollar sign ($) and the underscore (_) are also valid characters, but because most external names in the XL Pascal runtime environment begin with a dollar sign, you should avoid using it as the first character in an identifier. Defining an identifier with the same name as a runtime library routine may make the runtime library name inaccessible.

Examples

```
I
K9
New_York
AMOUNT$
```

The following identifiers are incorrect:

```
WEATHER_#1/2 (* contains special characters   *)
5K           (* starts with a number          *)
NEW JERSEY   (* has a blank between the words *)
```

Related Information

Compiler options are described in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

Basic Symbols

XL Pascal has a set of basic symbols that the compiler uses for specific purposes in the language:

- Reserved words
- Keywords
- Special symbols
- Operators

Reserved Words

Identifiers that define the syntax of the XL Pascal language are *reserved words*. You cannot declare these words for other uses. You must separate a reserved word from other reserved words and identifiers by either a comment, at least one blank, or one of the special symbols.

XL Pascal makes no distinction between uppercase and lowercase for reserved words. The **–U** compiler option has no effect on reserved words.

The identifiers in the following tables are the reserved words of XL Pascal.

Standard Mode XL Pascal Reserved Words

AND	END	NIL	SET
ARRAY	FILE	NOT	THEN
BEGIN	FOR	OF	TO
CASE	FUNCTION	OR	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	PROCEDURE	VAR
DO	IN	PROGRAM	WHILE
DOWNTO	LABEL	RECORD	WITH
ELSE	MOD	REPEAT	

VS Mode XL Pascal Reserved Words

ASSERT	LEAVE	REF	STATIC
CONTINUE	OTHERWISE	RETURN	VALUE
DEF	RANGE	SPACE	XOR

Keywords

Like a reserved word, a *keyword* is a predefined identifier, but it represents a language construct that you can redefine in a declaration.

You can write keywords in uppercase, lowercase, or mixed case, but the compiler folds everything outside single quotation marks to lowercase by default. If you disable case folding with the **–U** compiler option, you must enter a keyword in lowercase to keep its predefined meaning.

The keywords of XL Pascal are listed in the following tables.

Standard Mode XL Pascal Keywords

ABS	FALSE	PACK	SIN
ARCTAN	GET	PAGE	SQR
BOOLEAN	INPUT	PRED	SQRT
CHAR	INTEGER	PUT	SUCC
CHR	LN	READ	TEXT
COS	MAXINT	READLN	TRUE
DISPOSE	NEW	REAL	TRUNC
EOF	ODD	RESET	UNPACK
EOLN	ORD	REWRITE	WRITE
EXP	OUTPUT	ROUND	WRITELN

VS Mode XL Pascal Keywords

ADDR	GTOSTR	MIN	RPAD
ALFA	HALT	MINDEX	SEEK
ALFALEN	HBOUND	MININT	SHORTREAL
ALPHA	HIGHEST	MINREAL	SIZEOF
ALPHALEN	INDEX	MINSREAL	STDERR
CLOCK	LBOUND	MLENGTH	STOGSTR
CLOSE	LENGTH	MLTRIM	STR
COLS	LOWEST	MRINDEX	STRING
COMPRESS	LPAD	MSUBSTR	STRINGPTR
DATETIME	LTOKEN	MTRIM	SUBSTR
DELETE	LTRIM	NEWHEAP	TERMIN
DISPOSEHEAP	MARK	PARMS	TERMOUT
EPSREAL	MAX	POINTER	TOKEN
EPSSREAL	MAXCHAR	QUERYHEAP	TRACE
FLOAT	MAXLENGTH	RANDOM	TRIM
GCHAR	MAXREAL	READSTR	UPDATE
GSTR	MAXSREAL	RELEASE	USEHEAP
GSTRING	MCOMPRESS	RETCODE	WRITESTR
GSTRINGPTR	MDELETE	RINDEX	XL__TRAP

Special Symbols

The *special symbols* are nonalphabetic characters or groups of characters that represent such syntax elements as operators and variable quantifiers. The **–U** compiler option has no effect on special symbols.

Multiple-character special symbols such as <= cannot contain embedded spaces.

The following are the special symbols used by XL Pascal.

Standard Mode

`+`	Addition and set union operator
`-`	Subtraction and set difference operator
`*`	Multiplication and set intersection operator
`/`	Division operator, real result only
`=`	Equal operator, constant definition, and type definition
`<`	Less than operator
`<=`	Less than or equal operator, a set subset operator
`>=`	Greater than or equal operator, a set superset operator
`>`	Greater than operator
`<>`	Not equal operator
`:=`	Assignment symbol
`.`	Period to end a unit or a record field separator
`,`	Comma, list separator
`:`	Colon, specifies definition
`;`	Semicolon, statement separator
`..`	Subrange notation
`'`	Single quotation mark, begins and ends string literals
`@` or `^`	Pointer reference symbol
`(`	Left parenthesis
`)`	Right parenthesis
`[` or `(.`	Left square bracket
`]` or `.)`	Right square bracket
`{` or `(*`	Comment start delimiter
`}` or `*)`	Comment end delimiter

VS Mode

`–`	Unary negation
`/*`	Comment start delimiter
`*/`	Comment end delimiter
`~=`	Not equal operator
`->`	Pointer reference symbol

`~`	Boolean **NOT**, **INTEGER** one's complement, or set complement
`|`	Boolean **OR**, logical **OR** on **INTEGER**
`&`	Boolean **AND**, logical **AND** on **INTEGER**
`&&`	Boolean **XOR** operator, logical **XOR** on **INTEGER**
`>>`	Right logical shift on **INTEGER**
`<<`	Left logical shift on **INTEGER**
`||`	String concatenation operator

You can also write the following VS mode symbols as reserved words:

Symbol	Reserved Word
~	**NOT**
`|`	**OR**
&	**AND**
&&	**XOR**

Operators

The *operators* represent the logical or algebraic processes, such as addition or multiplication, that can be performed on a value or pair of values. The *operands* are the values manipulated by the operators. The *results* are produced from this manipulation.

The operators used in XL Pascal are in four categories:

- The **NOT** operator
- Multiplication operators
- Addition operators
- Relational operators

The NOT Operator

Standard Mode:

Operator	Operation
NOT	Boolean **NOT**

VS Mode:

Operator	Operation
~	Boolean **NOT**; logical one's complement; set complement
NOT	Boolean **NOT**; logical one's complement; set complement

Multiplication Operators

Standard Mode:

Operator	Operation
*	Multiplication; Set intersection
/	Real division
DIV	Integer division
MOD	Modulus
AND	Boolean **AND**

VS Mode:

Operator	Operation
AND	Logical **AND**
&	Boolean **AND**; logical **AND**
||	String concatenation
<<	Left logical shift
>>	Right logical shift

Addition Operators

Standard Mode:

Operator	Operation
+	Addition; set union
−	Subtraction; set difference
OR	Boolean **OR**

VS Mode:

Operator	Operation
+	String concatenation
OR	Logical **OR**
|	Boolean **OR**; logical **OR**
&& or **XOR**	Logical **XOR**; set exclusive union; Boolean **XOR**

Relational Operators

Standard Mode:

Operator	Operation
=	Compare equal
<>	Compare not equal
<	Compare less than
>	Compare greater than
<=	Compare less than or equal to; subset (inclusion of left operand in right)
>=	Compare greater than or equal; superset (inclusion of right operand in left)
IN	Set membership (left operand is a member of right operand)

VS Mode:

Operator	Operation
~=	Compare not equal

Comments

The *comment* is an annotation in your source code that explains some aspect of the program. It does not affect the operation of the program. You can place a comment anywhere in a unit where a blank would be acceptable. Use comments frequently to improve the understanding of your programs.

A comment can contain any printable character except the characters that act as the delimiters. Note that a comment delimiter within a string literal is part of the string literal; it is *not* interpreted as a comment delimiter. Comments can span multiple lines to form block comments.

To add comments to your XL Pascal code, use one of the following pairs of opening and closing comment delimiters:

```
{      }

(*     *)
```

The compiler uses the symbols (* and *) and { and } interchangeably. For the symbol { or (*, it bypasses all characters that follow it until it encounters the } or *) symbol. Comments can begin with one symbol and end with another.

Note: The left brace { and right brace } are X'7B' and X'7D' in ASCII. On some keyboards, these characters may not map to left brace and right brace. In this case, use only (* and *) to delimit comments.

Comments in VS Mode

In VS mode, you can use these as comment delimiters:

```
/*  ...  */
```

Using /* ... */ to delimit comments is different from using { ... } or (* ... *). The compiler bypasses all characters following / * until it encounters the */ symbol. A comment beginning with /* must end with */ to be considered one comment.

Different comment delimiters can indicate different types of comments. For example, you might want to use the delimiters (*...*) and {...} to indicate ordinary comments, and / *...*/ to block out a piece of temporary code or a debugging statement:

```
/*
IF A = 10 THEN (* this statement is
                  for program debugging *)
  WRITE('A IS EQUAL TO TEN');
*/
```

Multibyte characters can be included in comments. For example:

(* abcd **D DD** efg   **D D** hijklm*)

Note: **D** represents one MBCS character.

Examples

The following are examples of XL Pascal comments:

```
{ comment enclosed in braces }

(* other comment delimiters *)

{  mixed comment delimiters *)

/* To close this comment, } does not work.
The comment continues until it finds */
```

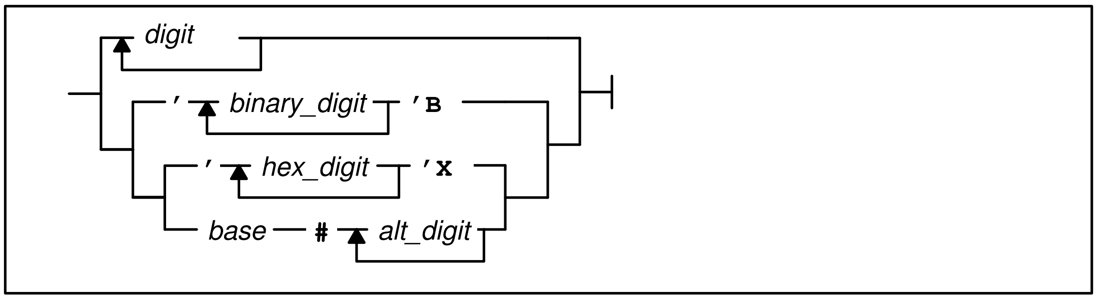
Literals

A *literal* is a quantity or other language object that takes only one specific value. Literals are not assigned to represent other values.

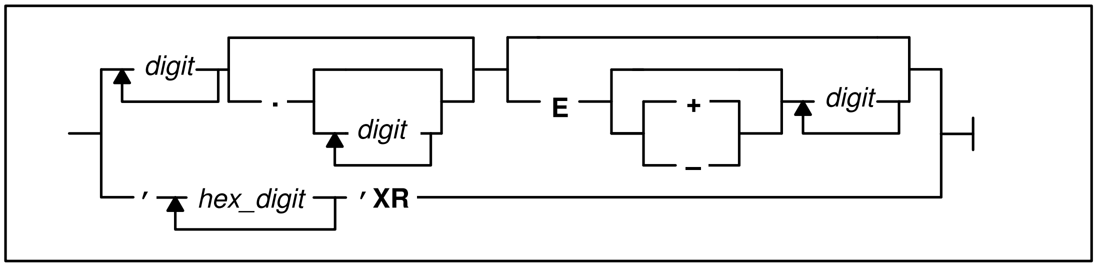
Syntax

In these diagrams, the uppercase letters can be replaced by lowercase letters with no change in meaning.

Unsigned Integer

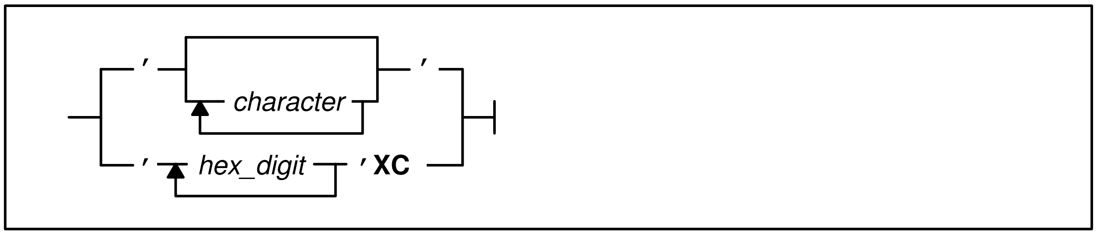


Unsigned Real Number

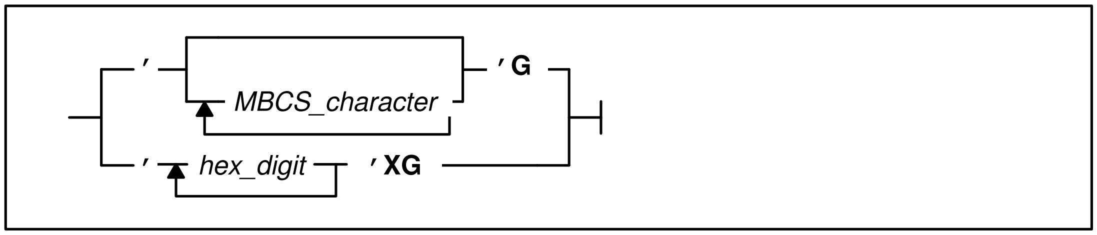


Note: The first digit is optional in VS mode only.

String



GSTRING



Parameters

Standard Mode:

digit	is 0 through 9.
character	is any ASCII character.

VS Mode:

binary_digit	is 0 or 1.
hex_digit	is 0 through 9, A through F, and a through f.
base	is a decimal integer literal whose value is one of 2 through 16.
alt_digit	is one or more digits that are members of the specified base number system.
MBCS_character	is any character that requires an MBCS representation.

Description

The symbol *E* or *e* indicates values in scientific or exponential form. The e precedes the exponent. When used in a real number, it expresses *times ten to the power of*.

Sequences of characters enclosed in single quotation marks are *strings*, which conform to the type **STRING** or **PACKED ARRAY** [1..*n*] **OF CHAR**. A string of one character conforms to the type **CHAR**.

For a single quotation mark to be recognized in a string, you must write the character twice. For example, you would have to type the string `SQUARE'S_SIDES` as:

```
'SQUARE''S_SIDES'
```

XL Pascal is case-sensitive for the characters in string literals: uppercase and lowercase letters are different. String literals written in XL Pascal cannot extend past the end of a line in the code.

String Concatenation

In VS mode, if your string literal cannot fit on a line, you must concatenate shorter strings, such as:

```
VAR
   s : STRING ;
BEGIN
   s := 'Since literals cannot be continued beyond the end of '||
        'a line, use concatenation to get the full string.';
```

Examples

Standard Mode

Literal	Standard Type
`0`	**INTEGER**
`1.0`	**REAL**
`314159E-5`	**REAL**
`0E0`	**REAL**
`1.0E10`	**REAL**
`TRUE`	**BOOLEAN**
`'A'`	**CHAR**
`'ABC'`	**Fixed String**
`'abc'`	**Fixed String**
`''''`	**CHAR**
`' '`	**CHAR**
`'  '`	**Fixed String**
`'That''s all '`	**Fixed String**

VS Mode

Literal	Standard Type
`'FF'X`	**INTEGER**
`'C1C2C2'XC`	**STRING**
`'4E800000FFFFFFFF'XR`	**REAL**
`''`	**STRING**

Hexadecimal and Binary Literals

VS mode XL Pascal permits the use of hexadecimal and binary literals of various types.

Integer Hexadecimal

These literals are enclosed in single quotation marks and suffixed with an *X* or *x*. For example, `'FF'X` is a valid integer hexadecimal literal. You can use one in any context where an integer literal is appropriate. If you do not specify 8 hexadecimal digits (4 bytes), the digits not supplied are zeros on the left. For example, `'F'X` is the same as `'0000000F'X`.

Integer Binary

These literals are enclosed in single quotation marks and suffixed with a *B* or *b*. A binary digit is either a 0 or a 1. For example,`'00000110'b` is a valid binary literal. If you do not specify 32 binary digits (4 bytes), the necessary number of zeros are added on the left.

Floating-Point Hexadecimal

These literals are enclosed in single quotation marks and suffixed with an *XR* or *xr*. You can use one in any context where a real literal is appropriate. If you do not specify 16 hexadecimal digits (8 bytes), the digits not supplied are zeros on the right. For example, `'4110'xr` is the same as `'4110000000000000'xr`.

String Hexadecimal

These literals are enclosed in single quotation marks and suffixed with an *XC* or *xc*. You can use one in any context where a string literal is appropriate. A string hexadecimal literal must contain an even number of digits, and you must fully specify each character in the string. For example, `'C1C2C2'XC` is a valid string hexadecimal literal.

Multibyte Character Set Hexadecimal

You can use multibyte character set (MBCS) literals in character strings. An MBCS hexadecimal literal consists of hexadecimal digits to specify an even number of bytes, enclosed in single quotation marks and suffixed with an *XG* or *xg*. Data is specified in the file code format. Each character in any particular string is represented by any number of bytes, from 1 to 4.

An MBCS hexadecimal literal is compatible with type **PACKED ARRAY** [1..*n*] **OF GCHAR** where *n* is the number of multibyte characters. For example, `'A0A1B2B3C4C5'xg` is an MBCS hexadecimal literal constant compatible with type `PACKED ARRAY [1..3] OF GCHAR`. All hexadecimal digits in an MBCS literal must be specified.

MBCS Literals

These are enclosed in single quotation marks and suffixed with *G* or *g*. Only MBCS characters are permitted between the quotation marks. MBCS literals are permitted only in VS mode.

Chapter 3. Structure of XL Pascal Programs

All Pascal programs are composed of separate compilation units link-edited to form a complete program. Each unit must be in a single language mode. There are two types of units in XL Pascal: *program units* and *segment units*. Compiling a unit independently from the rest of the program allows you to organize your code logically.

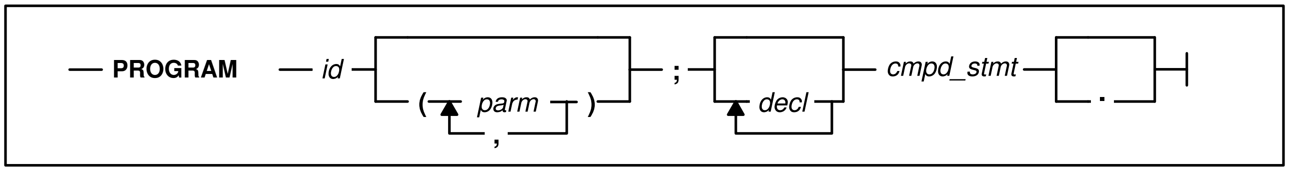
Each unit can consist of a series of declarations and statements. Declarations define program objects, and statements determine the actions the program performs on those objects. Together, they describe a computer program in Pascal.

Program Unit

Purpose

Gains initial control when you call a compiled program. It consists of all the statements between a **PROGRAM** statement and an **END** statement. The **PROGRAM** statement identifies the main program to the XL Pascal compiler.

Syntax



Parameters

PROGRAM	is an XL Pascal reserved word.
id	must be a unique external name.
parm	is an optional list of program parameters that specify links to external names.
decl	can be **LABEL**, **CONST**, **TYPE**, **VAR**, or routine declaration. Programs in VS mode can also include **DEF**, **REF**, **STATIC**, or **VALUE** declarations.
cmpd_stmt	is a compound statement that constitutes the program body.

Note: The period at the end of the program unit is optional only in VS mode.

Description

In standard mode, the program unit is the only compilation unit, and its sections must appear in the following order:

1. Label declarations
2. Constant definitions
3. Type definitions
4. Variable declarations

5. Procedure and function declarations
6. Compound statement (main program block)

Structure of an XL Pascal Program Unit

PROGRAM HEADER	Program Header
LABEL DECLARATIONS	Label Declarations
CONSTANT DEFINITIONS TYPE DEFINITIONS VARIABLE DECLARATIONS	Data Descriptions
PROCEDURE DECLARATIONS FUNCTION DECLARATIONS	Routine Declarations

BEGIN
    STATEMENTS;
      .
      .
      .
    STATEMENTS;
END.
	Main Program Block

The only required items (for both standard and VS mode) are the program header followed by the main program block.

Program Unit in VS Mode

Value declarations can be included in the data descriptions. The various declarations and definitions in XL Pascal are optional in VS mode and can appear in any order. Pointer target types are the only forward references permitted in a declaration.

You can have multiple declaration and definition sections in a single program unit.

Example

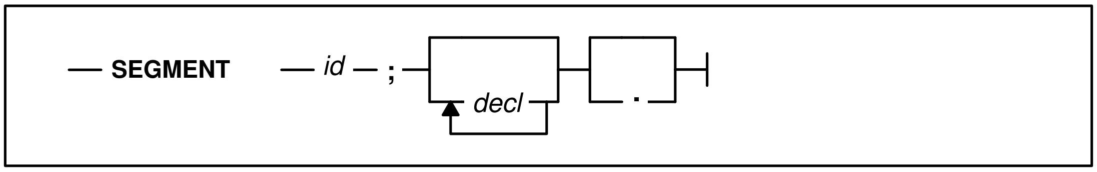
```
PROGRAM example;

VAR
   i : INTEGER;

BEGIN
   FOR i:=0 TO 1000 DO
      IF i MOD 7 = 0 THEN
         WRITELN( i : 5, ' IS DIVISIBLE BY SEVEN' )
END.
```

Segment Unit

Purpose

Consists of routines linked with the program unit at compilation. You can compile a segment unit independently of a program unit. Segment units are permitted in VS mode only.

Syntax



Parameters

SEGMENT is an XL Pascal reserved word.

id can be the same name as one of the **EXTERNAL** routines in the segment or it can be a unique name. A function called **SIN** could be in a segment called `SIN`. An external name is an identifier for a program, segment, **DEF** or **REF** variable, or **EXTERNAL** routine.

decl can be **CONST**, **TYPE**, **VAR**, **DEF**, **REF**, **STATIC**, **VALUE**, or a routine declaration.

Note: The declarations and the period at the end of the segment are optional.

Description

Segments are useful for sharing common code among different programs, or as a means of breaking up large programs into smaller units. Using smaller units, you can see the effect of small programming changes without having to compile the entire program with each modification.

Data is passed to routines through parameters and external variables. A segment unit has access to the global automatic variables of the program unit.

The various kinds of declarations in the segment unit are optional and can be in any order. The only required item is the segment header.

Structure of an XL Pascal Segment Unit

SEGMENT HEADER	Segment Header
CONSTANT DEFINITIONS TYPE DEFINITIONS VARIABLE DECLARATIONS VALUE DECLARATIONS	Data Descriptions
PROCEDURE DECLARATIONS FUNCTION DECLARATIONS	Routine Declarations

Example

```
SEGMENT cosine;
FUNCTION cosine ( x : REAL ) : REAL ; EXTERNAL;
FUNCTION cosine ;
   VAR
      s : REAL ;
   BEGIN
      s := SIN( x ) ;
      cosine := SQRT( 1.0 - s * s )
   END;
```

Related Information

Global automatic variables are described in “VAR” on page 31.

Program Parameters

Specify external bindings with XL Pascal variables. They contain one or more identifiers separated by commas. Program parameters are optional.

In Standard Mode

- To use the predefined files **INPUT** and **OUTPUT** in a program, you must specify them in the program parameter list. The default for **INPUT** and **OUTPUT** is terminal I/O.
- If you specify **INPUT** as a program parameter the file for input (**RESET**) is opened.
- If you specify **OUTPUT** as a program parameter the file for output (**REWRITE**) is opened.
- If you specify **INPUT** and **OUTPUT** as program parameters, you cannot redefine them as global variables.
- You cannot specify duplicate identifiers in the program parameter list. For example, the following is incorrect because parameter `f` is specified twice:

```
PROGRAM USER( OUTPUT, f, f );
```

- You must declare any identifier (other than **INPUT** and **OUTPUT**) that appears in the program parameter list as a variable identifier in the program block. For example, the following is incorrect because parameter `g` is declared as a constant rather than a variable, and parameter `f` is not declared at all:

```
PROGRAM USER( OUTPUT, f, g );
CONST
  g = 3;
```

In VS Mode

- The files **INPUT** and **OUTPUT** are predefined, so you need not specify them in the program parameter list when they appear in a program.
- You can redefine **INPUT** and **OUTPUT** in your program.
- The declaration of the variable determines external binding. For example:

```
PROGRAM USER(OUTPUT,f,g);
   DEF
      f: INTEGER;     (* bound to an external symbol *)
   VAR
      g: INTEGER;     (* global automatic variable   *)
```

  If you define a program parameter as anything other than a variable identifier, XL Pascal issues a warning diagnostic message.
- Program parameters can have defining points anywhere in the program block.

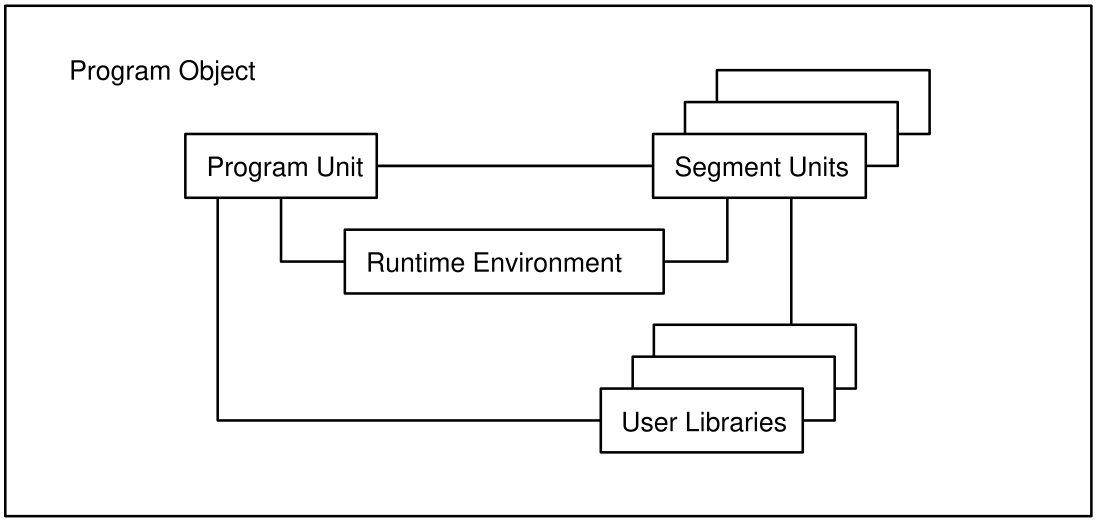
Linking Units

An XL Pascal program is formed by linking a program unit to:

- The XL Pascal runtime environment
- Segment units, if any
- Other libraries you might supply

The following figure illustrates the relationship between program and segment units, the XL Pascal runtime environment, and additional user-supplied libraries.

Linking a Program Unit with a Segment Unit



Related Information

For information about linking, libraries, and the IBM AIX XL Pascal runtime environment, refer to the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

Standard Files

XL Pascal supplies the following three standard files. They are predefined as variables of type **TEXT**:

INPUT	is the standard file from which you do input by **READ**, **READLN**, and **GET** routines. This default is associated with the standard input.
OUTPUT	is the standard file to which you direct output by **WRITE**, **WRITELN**, and **PUT** routines. This default is associated with the standard output.
STDERR	is the standard error output file. It is defined for VS mode only. This default is associated with the standard error file.

Chapter 4. Declarations

Declarations associate identifiers with program objects, such as data types, variables, and routines, so that they can be used in the program. You must predefine or declare each identifier before you use it. There is one exception to this rule: a pointer definition can refer to an identifier as the domain type of the pointer before it is declared. The domain type identifier must be declared later, or XL Pascal generates a compile-time diagnostic message.

This chapter describes: lexical scope of identifiers, declarations in standard mode, and declarations in VS mode.

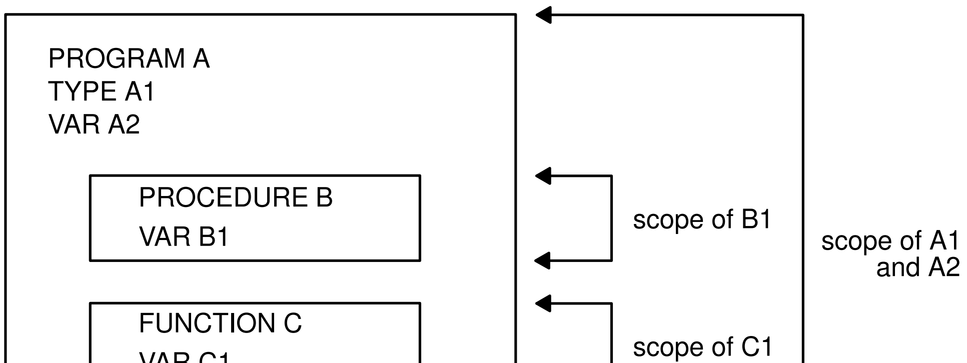
Lexical Scope of Identifiers

The *lexical scope*, or *scope*, of an identifier is the portion of a program where the identifier is accessible. The scope of an identifier can be *global* or *local*.

Local identifier	is associated with a variable defined in a function or procedure. A local variable is not accessible to an outside function, procedure, or main program.
Global identifier	is associated with a variable defined in a main program. You can use, refer to, or change a global variable anywhere in the program and include functions or procedures.

For example, in the following figure, variable `A2`, defined in `PROGRAM A`, is global. Because `PROGRAM A` contains `PROCEDURE B` and `FUNCTION C`, `PROCEDURE B` and `FUNCTION C` can refer to `A2`, declared in `PROGRAM A`. Identifier `B1`, however, is declared within `PROCEDURE B`, and is not accessible either from within the body of `PROGRAM A`, or from within `FUNCTION C`.

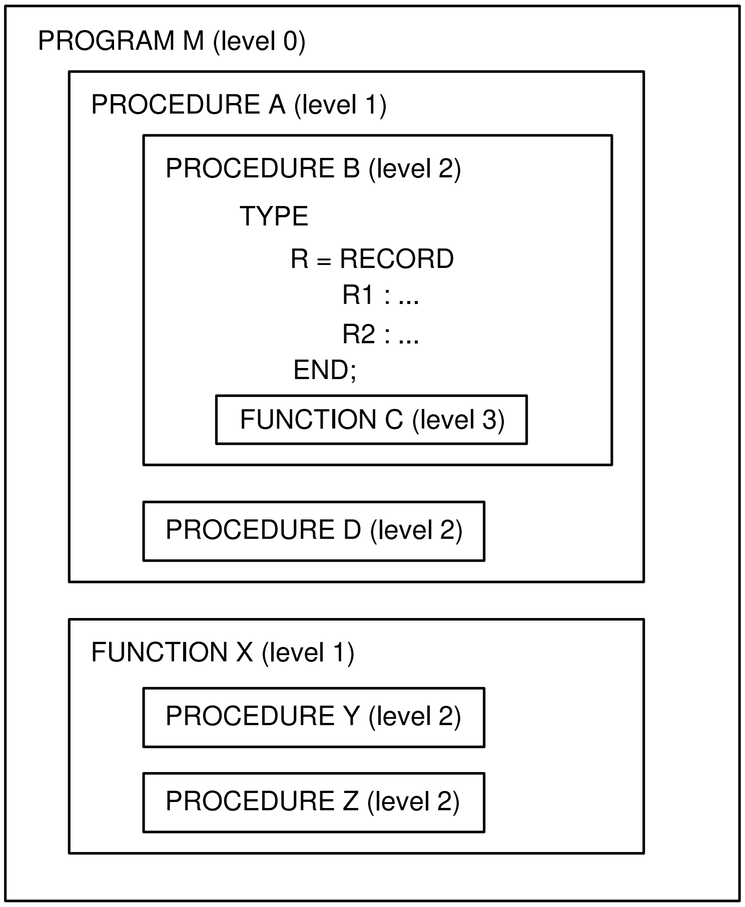
Scope of Identifiers



The scope of any particular identifier depends on the structure of the routine declarations within the unit in which it appears. The scope of an identifier is the entire routine (or unit) in which it was declared, including all routines nested within the routine. Record definitions also define a lexical scope for the record fields. You can define each identifier only once within a lexical scope.

Because routines can be nested within other routines, a *lexical level* is associated with each routine. A program unit is at lexical level 0, and routines defined within the unit are at lexical level 1. In general, identifiers defined in a routine defined in level *i* are accessible at level (*i*+1).

Nesting Structure of a Program



Identifiers Declared in	**are Accessible in**
PROGRAM M	M, A, B, C, D, X, Y, Z
PROCEDURE A	A, B, C, D
PROCEDURE B	B, C
TYPE R	B, C
FUNCTION C	C
PROCEDURE D	D
FUNCTION X	X, Y, Z
PROCEDURE Y	Y
PROCEDURE Z	Z

Note: The scope of a field identifier defined within a record definition is limited to the record or to the scope of any variable defined to be of that record type. A field of a record can be accessed using either field referencing or the **WITH** statement.

When an identifier is declared in a routine nested in the scope of another identifier with the same name, the new identifier is the one recognized when its name appears in the routine. The inner routine has no access to the first identifier. In other words, the only identifier that can be used is the one declared at the innermost level.

For example, in the figure above, `FUNCTION C` is nested in `PROCEDURE B`, `PROCEDURE B` is nested in `PROCEDURE A`, and `PROCEDURE A` is nested in `PROGRAM M`. If both `PROGRAM M` and `PROCEDURE B` declared an identifier `T`, a conflict could arise. To resolve the conflict, use the most recent declaration of `T`. The identifier `T` declared in `PROGRAM M` would be used for `PROGRAM M, PROCEDURE A` and `FUNCTION X`. In `PROCEDURE B` and `FUNCTION C`, the identifier `T` declared in `PROCEDURE B` would be used.

The XL Pascal compiler inserts a *prime file* of precompiled declarations at the beginning of every unit it compiles. These declarations comprise predefined types, constants, routines, and variables. The scope of the prime file encompasses the entire unit.

Related Information

Prime files are described in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

Declarations in Standard Mode

The required order of declaration sections for standard mode XL Pascal is:

1. **LABEL**
2. **CONST**
3. **TYPE**
4. **VAR**
5. **PROCEDURE**
6. **FUNCTION**

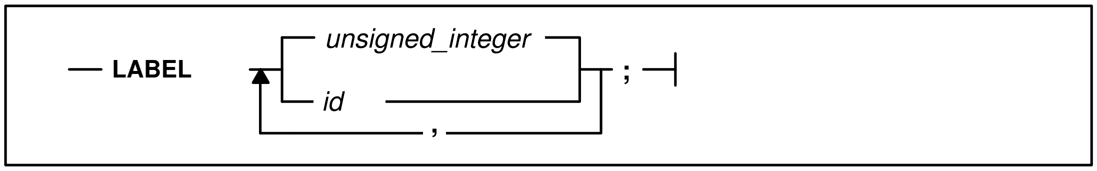
The standard mode declarations are described in the following sections. Procedure and function declarations are discussed in "Routine Declarations" on page 133.

LABEL

Purpose

Declares labels referred to by a **GOTO** statement within a routine.

Syntax



Parameters

LABEL	is the declaration reserved word.
unsigned–integer	is a name assigned to a label. It must be in the range 0 to 9999.
id	is an identifier name assigned to a label (VS mode only).

Description

To declare two or more labels in the declaration, use commas to separate the label names. You must declare all labels defined within a routine in a **LABEL** declaration.

Example

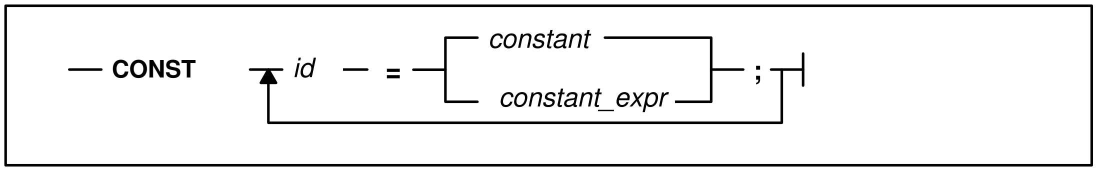
```
LABEL
   10,
   1,
   2,
   label_a,
   error_exit;
```

CONST

Purpose

Defines identifiers to use as synonyms for constant expressions.

Syntax



Parameters

CONST	is the declaration reserved word.
id	is an identifier assigned to a constant or, in VS mode, a constant expression.
constant	is any constant.
constant_expr	is any constant expression (VS mode only).

Description

All constant names and their associated values are local to a program, procedure, or function definition. The type of the expression in the declaration determines the type of a constant identifier.

CONST Declarations in VS Mode:

VS mode allows you to specify the value of a **CONST** identifier by using either a simple constant or a constant expression.

Example

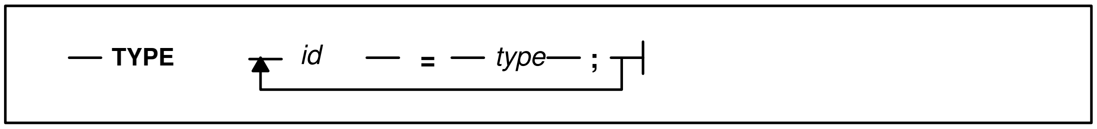
```
CONST
   blank     = ' ';
   blanks    = '  ';
   fifty     = 50;
   a         = fifty;
   pi        = 3.14159265358;
   letters   = [ 'A'..'Z','a'..'z' ];
   b         = fifty * 10 / ( 3 + 2 );
   c_squared = a * a + b * b;
   ord_of_a  = ORD( 'a' );
   mask      = '8000'X | '0400'X;
```

TYPE

Purpose

Defines a data type and associates a name with that type. Once declared, such a name can be used in the same way as a predefined type name.

Syntax



Parameters

TYPE	is the declaration reserved word.
id	is an identifier for a type.
type	is the type.

Example

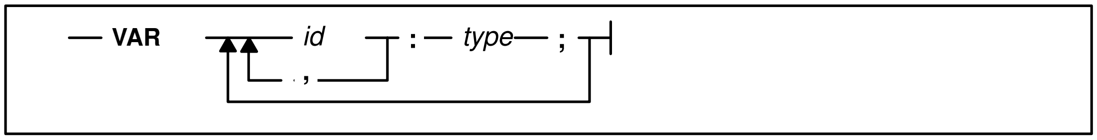
```
TYPE
   card_value = 1..13;
   card_suit  = ( spade, heart, club, diamond );
   card_type  = RECORD
                   rank : card_value;
                   suit : card_suit;
                   face_up : BOOLEAN;
                END;
   game_hand  = ARRAY[card_value] OF card_type;
```

VAR

Purpose

Declares *automatic variables*, which are variables allocated when a routine is called and deallocated when the corresponding return is made.

Syntax



Parameters

VAR	is the declaration reserved word.
id	is an identifier for a VAR variable.
type	is the type of the VAR variable.

Description

If a routine is called recursively, each invocation of the routine allocates separate copies of all automatic variables to be used by that invocation.

To declare two or more identifiers of the same type in the declaration, use commas to separate the identifiers. This is a shorthand notation for two separate declarations.

Example:

```
VAR
   i : INTEGER;
   sysin : TEXT;
   x, y, z : REAL;
   card :  RECORD
              rank : 1..13;
              suit : ( spade, heart, diamond, club )
           END;
```

VAR Declarations Shared between Units

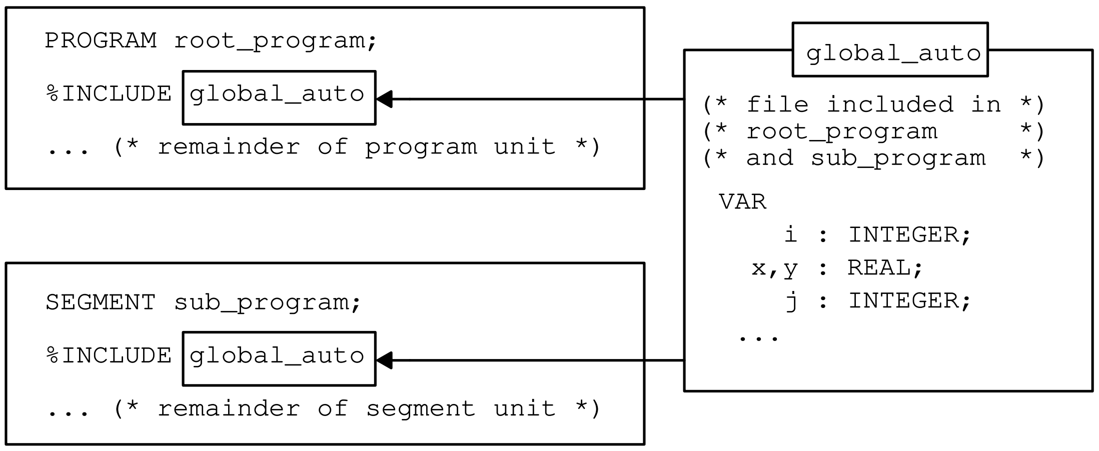
In VS mode, all variables declared with **VAR** in the outermost nesting level of a program or segment unit are global automatic variables. They are accessible throughout that unit. When a program and one or more segments are linked, the global automatic variables of all compilation units occupy the same storage locations to give all units access to the same global automatic variables.

The following example shows a **VAR** declaration shared between a program and a segment:

```
PROGRAM main;
   VAR
      i : INTEGER;
      x, y : REAL;
      j : INTEGER;
   ...(* remainder of program unit *)

SEGMENT seg;
   VAR
      i : INTEGER;
      x, y : REAL;
      j : INTEGER;
   ...(* remainder of segment unit *)
```

Global automatic variable declarations that are not identical to those in the program yield unpredictable results. XL Pascal does not detect differences between the global automatic declarations in different program and segment units. You should define the global area once with an **%INCLUDE** statement to insert identical copies of variable declarations in all separately compiled units.

Preferred Method of Sharing VAR Declarations between Programs and Segments


```
PROGRAM root_program;

%INCLUDE global_auto

... (* remainder of program unit *)
```

```
SEGMENT sub_program;

%INCLUDE global_auto

... (* remainder of segment unit *)
```

```
global_auto

(* file included in *)
(* root_program     *)
(* and sub_program  *)

VAR
     i : INTEGER;
   x,y : REAL;
     j : INTEGER;
  ...
```

Declarations in VS Mode

Declaration sections can be in any order in VS mode XL Pascal, and multiple declaration sections of the same type are permitted. This extension to Standard Pascal is provided primarily to permit source included during compilation to be independent of any ordering already established in the unit.

You can make forward references in declarations, but only to pointer target types.

In addition to those available in standard mode, you can use the following declarations in VS mode:

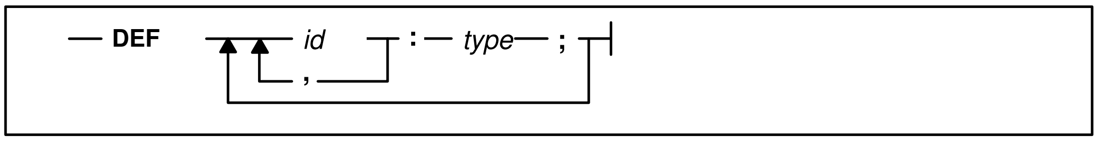
- **DEF**
- **REF**
- **STATIC**
- **VALUE**

DEF

Purpose

Defines and declares *external variables*, which are allocated before run time and can be accessed from more than one unit. External variables follow the same syntactic rules as internal variables.

Syntax



Parameters

DEF	is the declaration reserved word to specify that the program loader is responsible for generating the common storage for the variable.
id	is an identifier for the **DEF** variable.
type	is the type of the variable.

Description

External variables declared as **DEF** with the same name in several units are all allocated to a single common storage location. Variables with the same name must have identical data types in all units. You must assure that the types are the same. A bind-time diagnostic message is generated if **–qEXTCHECK** is on and all **DEF** and **REF** declarations for a given symbol do not have exactly the same type.

To declare two or more identifiers of the same type in the declaration, use commas to separate the identifiers. This is a shorthand notation for two separate declarations.

You can declare a **DEF** variable local to a routine, and the same scope rules apply as for any other declared identifier. If, however, you declare the name of the variable in another scope (even in another unit) as a **DEF** or **REF** variable, both occurrences of the variable refer to the same storage.

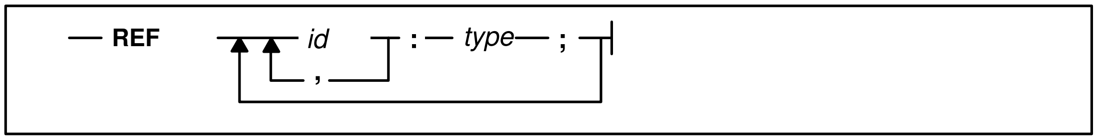
To initialize **DEF** variables at compile time, use a **VALUE** declaration.

REF

Purpose

Declares external variables that are defined elsewhere in the program. These are allocated before run time and can be accessed from more than one unit. External variables follow the same syntactic rules as internal variables.

Syntax



Parameters

REF	is the declaration reserved word, which specifies that storage for the variable is defined in another unit.
id	is an identifier for the **REF** variable.
type	is the type of the variable.

Description

A single common storage location is allocated to an external variable declared as **REF** with the same name in several units. Variables with the same name must have identical data types in all units. You must ensure that the types are the same.

To declare two or more identifiers of the same type in the declaration, use commas to separate the identifiers.

Variables declared **REF** remain unresolved until the encompassing unit is combined with a unit in which the variable is either declared as a **DEF** variable, or defined in a non-Pascal program as external. For example, you can use **REF** variables to access external data declared in a program written in assembler language. A bind-time diagnostic message is generated for any **REF** variables that remain unresolved.

A bind-time diagnostic message is also generated if **–qEXTCHECK** is on and all **REF** declarations for a given symbol do not have exactly the same type. A **REF** variable can be declared local to a routine, and the same scope rules apply as for any other declared identifier. If you declare the name of the variable as a **REF** or **DEF** variable in another scope (even in another unit), both occurrences of the variable refer to the same storage.

Example

In the following example, the external variable `x` in procedures `a`, `b`, and `c` refers to the same storage. The variable `x` declared in segment `p` and in procedure `d` each refer to storage separate from the external variable `x`.

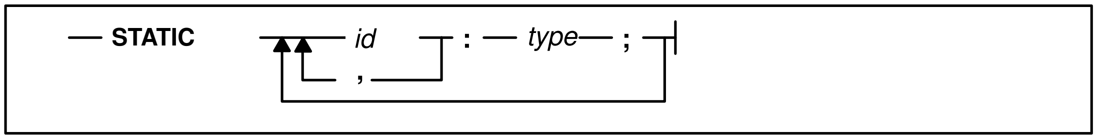
```
SEGMENT m;

PROCEDURE a;
   DEF
      x : REAL;   (* same as x in b and c  *)
   BEGIN
   ...
   END;

PROCEDURE b;
   REF
      x : REAL;   (* same as x in a and c  *)
   BEGIN
   ...
   END;.

SEGMENT p;
STATIC
   x : REAL;      (* local to p       *)
PROCEDURE c;
   REF
      x : REAL;   (* same as x in a,b *)
   BEGIN
   ...
   END;

PROCEDURE d;
   VAR
      x : REAL;   (* local to d       *)
   BEGIN
   ...
   END;.
```

STATIC

Purpose

Declares *static variables*, whose memory is allocated at the beginning of the program and which are local to the program, segments, or routines in which they are defined. This memory allocation occurs for the life of the program.

Syntax



Parameters

STATIC	is the declaration reserved word.
id	is an identifier for the **STATIC** variable.
type	is the type of the **STATIC** variable.

Description

You refer to static variables in your program according to the normal lexical scope rules. Even when they have the same name, XL Pascal treats static variables with different scopes as different variables.

To declare two or more identifiers of the same type in the declaration, use commas to separate the identifiers. This is a shorthand notation for two separate declarations.

Data in static variables local to a routine is preserved over separate calls to the routine. When such a routine is called recursively or repeatedly, it accesses the same instance of each static variable.

To initialize static variables at compile time, use a **VALUE** declaration.

Example

The following program demonstrates the effect of declaring variables using the **VAR** and **STATIC** declarations. Note that you cannot initialize the variable `auto_var` in procedure `auto_static` in the same way as in `static_var` using a **VALUE** declaration.

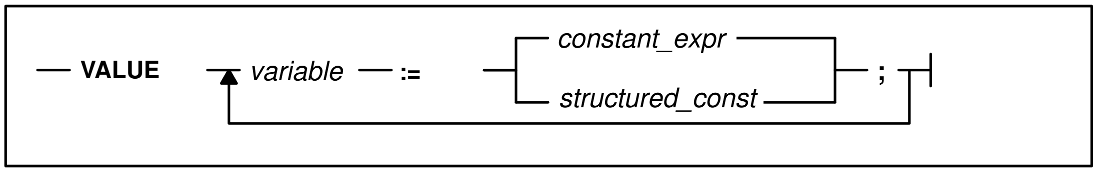
```
PROGRAM statauto;

VAR
   i : INTEGER;

PROCEDURE auto_static;
   VAR
      auto_var : INTEGER ;(* cannot be init'd at compile time *)
   STATIC
      static_var : INTEGER ;
   VALUE
      static_var := 0;     (* STATIC variable initialization   *)
   BEGIN                   (* start of auto_static             *)
      static_var := static_var + 1;
                           (* value of static_var is preserved *)
                           (* across each call to procedure    *)
      WRITELN( auto_var );(* value of auto_var is undefined   *)
   END;                    (* end of auto_static               *)

BEGIN                   (* start of statauto *)
   FOR i := 0 TO 10 DO
      auto_static ;
END.                    (* end of statauto *)
```

VALUE

Purpose

Specifies initial values for **STATIC** and **DEF** variables. It consists of a list of value assignments separated by semicolons.

Syntax



Parameters

VALUE	is the declaration reserved word.
variable	is a variable to be assigned a value.
constant_expr	is any constant expression.
structured_const	is any structured constant.

Description

The assignments in a **VALUE** declaration have the same form as the assignments in the body of a routine, except that all subscripts and expressions must be able to be evaluated at compile time. For example:

```
(* Initializing a three-dimensional array *)
TYPE
   cube = ARRAY[1..10,1..10,1..10] OF REAL;

STATIC
   block : cube ;

(* the following assignments take place at compile time  *)
VALUE
   block := cube ( ( (0.0:10):10 ):10 ) ;
```

You can use **VALUE** declarations to initialize separate scalar components of a **DEF** or **STATIC** variable, for example:

```
TYPE
   complex = RECORD
                re,im: REAL
             END;
   vector = ARRAY[1..7] of INTEGER;

STATIC
   c : complex;
   v : vector;
   v1 : vector;

DEF
   i : INTEGER;
   q : ARRAY[1..10] OF complex;


(* the following assignments take place at compile time *)
VALUE
   c := complex( 3.0, 4.0 );
   v := vector( 1, 0 : 5, 7 );
   v1 := vector( , , , 4 );
   v[2] := 2;
   v[3] := 3 * 4 - 1;
   i := 0;
   q[1].re := 3.1415926 / 2;
   q[1].im := 1.414;
```

You cannot specify more than one initial value for any scalar component of a **DEF** or **STATIC** variable within one program or segment unit.

For example, all of the statements in the following example can be in one program or segment. Together they initialize the odd-numbered elements of array `a`, and each element has no more than one **VALUE** specification.

```
TYPE
   atype = ARRAY[1..10] OF INTEGER;

DEF
   a : atype;

VALUE
   a := atype( 111 , , 333 , , , , 777 );
   a[5] := 555;
   a[9] := 999;
```

You do not need to initialize all scalar components of a **DEF** array or record. You cannot initialize any of them more than once in one program or segment unit.

You can use **VALUE** to initialize a **DEF** variable in any of the program or segment units that declare the **DEF** variable. The loader does not use the initial values in all but the first program or segment unit that is linked together into a program. In the following example, the program unit initializes the first and third elements of array `B`:

```
PROGRAM defvaldemo(OUTPUT);

PROCEDURE defvaldup; EXTERNAL;

   TYPE
      Bt = ARRAY[1..4] OF INTEGER;

   DEF
      B : Bt;

   VALUE
      B[1] := 111;
      B[3] := 333;
   .
   .
```

The following example shows a segment unit that contains the external procedure called by the program `defvaldemo`. This segment initializes the second and fourth elements of array `B`:

```
SEGMENT defvalseg;

TYPE
   Bt = ARRAY[1..4] OF INTEGER;

DEF
   B : Bt;

VALUE
   B[2] := 222;
   B[4] := 444;

PROCEDURE defvaldup; EXTERNAL;
PROCEDURE defvaldup;
   BEGIN
   .
   .
```

If the program unit `defvaldemo` and the segment unit `defvalseg` are compiled and linked by the command

```
xlp defvaldemo.pas defvalseg.pas
```

the first value the linkage editor uses is the initial value of `B` specified in program unit `defvaldemo`. Therefore, the linkage editor does not use the initial value of `B` specified in the segment unit `defvalseg` even though it specifies values of different scalar components of `B`. When the complete program is run, `B` has only the initial values specified in the unit `defvaldemo`.

The compiler does not detect conflicting initial value specifications made in different compilation units. The results are unpredictable if you use **VALUE** specifications to give different initial values to a **DEF** variable in more than one program or segment. To avoid unpredictable initialization, you should specify the initial value of a **DEF** variable in one of two ways:

- In an **%INCLUDE** file so that all compilation units have the same initial values for it. It makes no difference which value is first.
- Only in the program unit, where it has only one initial value.

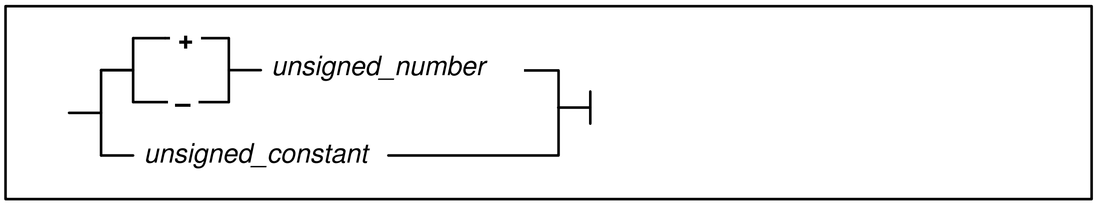
Chapter 5. Constants

The *constants* are either literal values, identifiers declared as constant names by **CONST** declarations, or structured constants. Literals represent values of simple types and string types. Structured constants represent values of structured types. You can use structured constants only in VS mode.

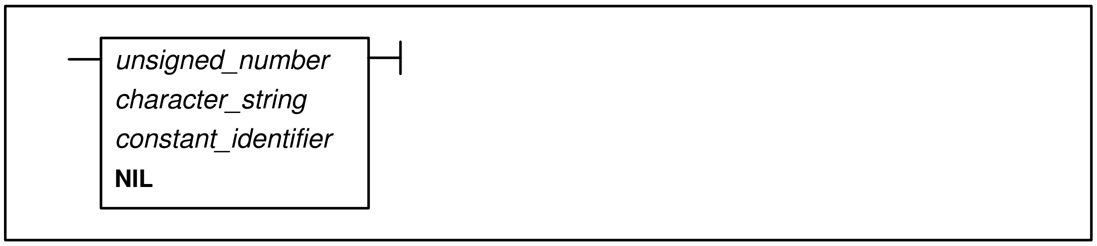
This chapter describes the predefined constants and structured constants of XL Pascal.

Syntax

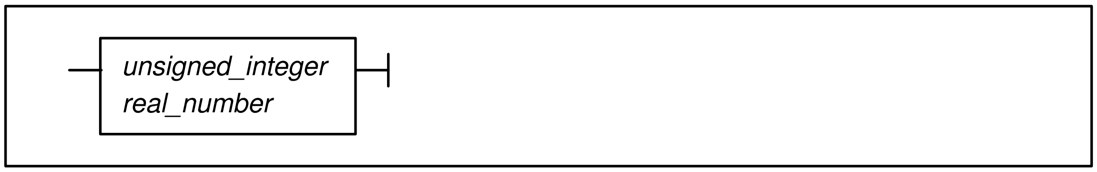
Constant



Unsigned Constant



Unsigned Number



Note: In “Constant”, signed constant identifiers must represent numeric values.

XL Pascal in VS mode permits constant expressions in places where standard mode permits only constants. Constant expressions are evaluated and replaced by a single result at compile time.

Related Information

Literal values are described on page 17.

The **CONST** declarations are described on page 30.

Constant expressions are described on page 105.

Predefined Constants

Identifiers already defined within XL Pascal are known as *predefined constants*. They are declared in the default prime file, so you need not define them.

In Standard Mode

FALSE	Constant of type **BOOLEAN**, **FALSE** < **TRUE**
MAXINT	Maximum value of type **INTEGER**: 2147483647
NIL	Constant of any pointer type representing an empty pointer value
TRUE	Constant of type **BOOLEAN**, **TRUE** > **FALSE**.

In VS Mode

ALFALEN	Length of type **ALFA**, value is 8
ALPHALEN	Length of type **ALPHA**, value is 16
EPSREAL	The smallest **REAL** value that, when added to 1, is detectable: 2.220446049250E–016 ('3CB0000000000000'XR)
EPSSREAL	The smallest **SHORTREAL** value that, when added to 1, is detectable: 1.192092895508E–007 ('3E80000000000000'XR)
MAXCHAR	Maximum value of type **CHAR**: 'FF'XC
MAXREAL	Maximum value of type **REAL**: 1.797693134862E+308 ('7FEFFFFFFFFFFFFF'XR)
MININT	Minimum value of type **INTEGER**: –2147483648
MINREAL	Minimum positive value of type **REAL**: 4.940656458412E–324 ('0000000000000001'XR)
MAXSREAL	Maximum value of type **SHORTREAL**: 3.402823466385E+038 ('47EFFFFFE0000000'XR)
MINSREAL	Minimum positive value of type **SHORTREAL**: 1.401298464325E–045 ('36A0000000000000'XR).

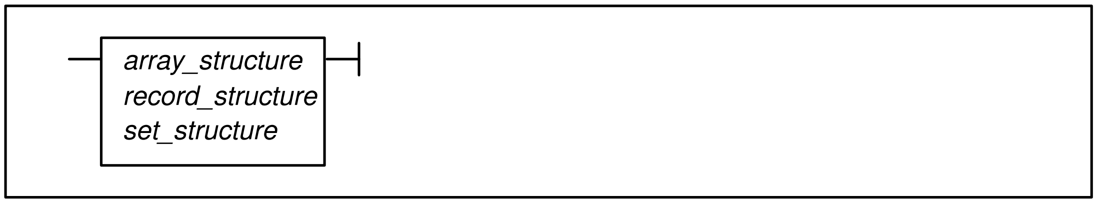
Related Information

Prime files are described in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

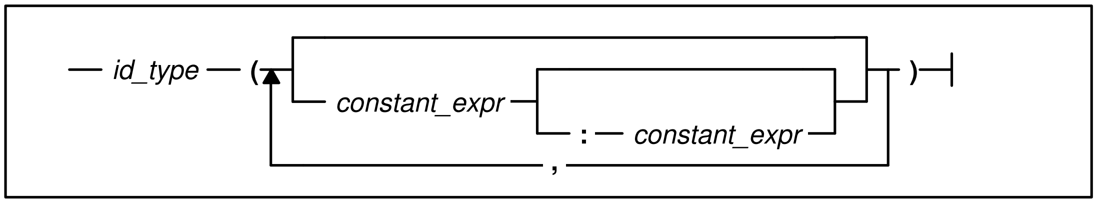
Structured Constants

Structured constants provide a convenient way of specifying a structured data element. They are expressions of structured type. Type definitions are determined by the type identifier in the constant's definition. Structured constants can be used in value declarations, other constant declarations, or in expressions.

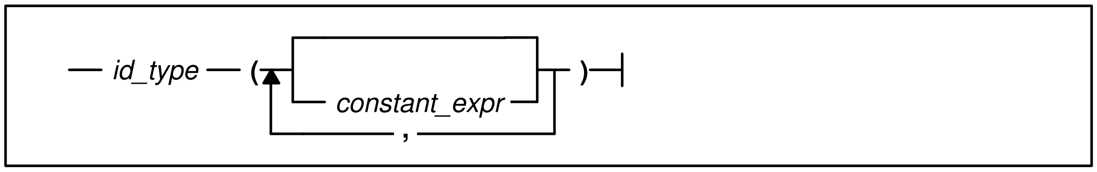
Syntax



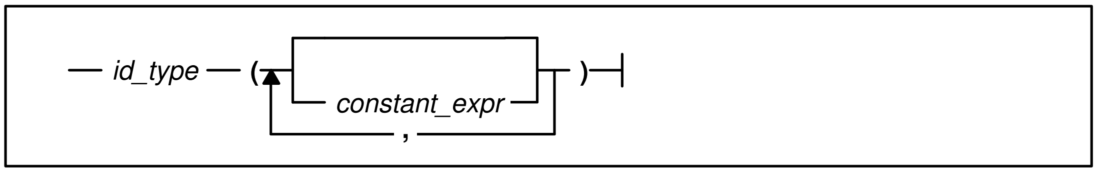
Array Structure



Record Structure



Set Structure



Parameters

constant_expr	is any constant expression.
id_type	is an array, record, or set type that does not contain a file.

Note: In "Array Structure", the repetition of *constant expression* after the colon (:) must be evaluated to a positive integer.

For structured constants imbedded within other structured constants, you can omit the type identifier that begins the constant. This simplifies the syntax for structured constants that are multidimensional arrays or records with structured fields.

XL Pascal allows three types of structured constants:

- Array constants
- Record constants
- Set constants

Array Constants

Array constants are specified by a list of constant expressions in which each expression defines one element of the array.

To omit an element of the array within the list, specify nothing between the commas, as shown in the definition of `vector_2`. You can omit an element either within the list or at the end of the array; in either case, the value of that element is not defined. Commas are necessary even in empty lists.

To specify that the value of the constant expression is to be placed in a specified number of array elements, follow the constant expression with a colon and a repetition expression, as shown in the definition of `vector_1`. This has the same effect as having a series of values separated by commas, as shown in the following example:

```
TYPE
   vector = ARRAY[1..7] OF INTEGER;
   tetra  = ARRAY[1..3,1..2,1..4] OF INTEGER;

CONST
   (* Structured Constants   *)
   vector_1   = vector( 7, 0 : 5, 1);
   vector_2   = vector( 2, 3, , 4 );
   zero_tetra = tetra( ( ( 0 : 4 ) : 2 ),
                       ( ( 0 : 4 ), ( 0 : 4 ) ),
                       ( ( 0, 0, 0, 0 ), ( 0, 0, 0, 0 ) ) );
```

Related Information

Constant expressions are described on page 105.

Record Constants

Record constants are specified by a list of constant expressions where each expression defines one field of the record in the order declared. You can omit a field of the record within the list by specifying nothing between two commas; the value of that field is not defined. Commas are necessary even in empty lists.

```
TYPE
   complex = RECORD
                re, im: REAL
             END;
CONST
   (* Structured Constants   *)
   threefour = complex(3.0,4.0);
```

Values within the list may correspond to fields of a record’s variant part. To tell the compiler which variant is being referenced, you must specify the tag field value immediately before those values to be assigned to the variant fields. When only a tag type is specified, you must specify the tag field even if it is not a field, as shown in the following example.

```
TYPE
   form  = ( fchar, finteger, freal, fstring );
   konst = RECORD
              size : INTEGER ;
              CASE f : form OF
                 fchar :
                    ( c : char );
                 finteger :
                    ( CASE size : OF
                      4 :
                            ( s : SHORTREAL );
                      8 :
                            ( r : REAL )
                    );
                 fstring :
                    ( CASE BOOLEAN OF
                         TRUE :
                            ( len : packed 0..32767;
                              a : ALPHA
                            );
                         FALSE :
                            ( st : STRING( 16 ) )
                    );
             END;

CONST
   a      = konst(1,fchar,’A’);
   int    = konst(4,finteger,3);
   short  = konst(4,freal,4,1.2345);
   pi     = konst(8,freal,3.14159);
   blank  = konst(1,fstring,FALSE,’ ’);
   stars  = konst(4,fstring,TRUE,4,’****’);
   bars   = konst(4,fstring,FALSE,’----’);
```

A refer-back tag field must be specified twice in the list: once to be assigned a value, and once to identify the variant being referenced. Both occurrences can specify different values for the refer-back tag field, but the compiler checks whether the same value is specified in both places. If a conflict occurs, a warning is issued and the second value is used, which is the value specified at the location of the variant part of the record.

The following example shows an array and a record constant combined:

```
TYPE
   complex = RECORD
                re,im: REAL
             END;
   carray = ARRAY[0..9] OF complex;

CONST
   (* the following two declarations are equivalent *)
   vector_3  = carray ( complex ( 1.0, 0.0 ),
                        complex ( 1.0, 1.0 ) : 8,
                        complex ( 0.0, 1.0 )     );
   vector_4  = carray ( ( 1.0, 0.0 ),
                        ( 1.0, 1.0 ) : 8,
                        ( 0.0, 1.0 )     ) ;
```

Related Information

Refer-back tag fields are described in “Variant Part” on page 75.

Set Constants

Structured constants can also be set-valued. These allow a set type to be specified as part of the constant. Like the other types of structured constants, set constants are specified by a list of constant expressions where each expression defines one element of the set, as shown in the following example:

```
TYPE
   smallnums = SET OF 0..127;

CONST
   small_powers_of_two = smallnums( 1, 2, 4, 8, 16, 32, 64 );
```

Set members can be in any order, and you can specify the same element more than once. A set-valued structured constant is equivalent to the disjunction (the **OR** operation) of the members of the base type specified as constant expressions in the set constant.

Chapter 6. Data Types

Every variable and constant in a Pascal program has a type associated with it. A data type determines the permissible values that a variable can assume or a function can return. It also determines the operations that can be performed on variables and constants. For example, integers can be multiplied; characters cannot. XL Pascal provides several predefined data types. You can also define your own data types using **TYPE** declarations.

This chapter describes data types and strings, and shows how to create your own data types.

Basic Data Types

Three kinds of data types are simple, pointer, and structured.

Simple

Standard Mode:

Boolean	The enumerated type whose values are (**FALSE**, **TRUE**).
Char	All the values of the American National Standard Code for Information Interchange (ASCII) character set.
Enumerated	An ordered set of values defined by listing the identifiers that stand for specific values. For example, you can enumerate the days of the week as Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday.
Integer	A positive or negative whole number, or zero.
Real	A positive or negative double-precision floating-point number, or zero.
Subrange	The minimum and maximum values permitted for a previously defined data type. A subrange is not permitted for **REAL** data type.

VS Mode:

Gchar	All the values of a multibyte character set (MBCS).
Shortreal	A positive or negative single-precision floating-point number.
Subrange	A subrange is not permitted for **SHORTREAL** or **GCHAR** data types.

Ordinal and Scalar

Any of the simple data types can fall under one or both of the following general categories:

Scalar	A type whose values contain only one element. All simple data types are scalars.
Ordinal	A scalar type whose values are mapped to a continuous range of integers. All scalars are ordinal except **REAL**, **SHORTREAL**, and **GCHAR**.

Pointer

Pointer data types reference dynamic variables, which are variables whose storage is allocated at run time.

String Pointer (STRINGPTR)

In VS mode programs, a **STRINGPTR** data type defines a pointer to a dynamic string variable. The maximum length of a string pointer target is determined at run time.

Multibyte Character String Pointer (GSTRINGPTR)

In VS mode programs, a **GSTRINGPTR** data type defines a pointer to a dynamic variable that is a string of multibyte characters. The maximum length of a **GSTRINGPTR** target is determined at run time.

Structured

Structured types are collections of data defined by describing the types of the components and indicating their structuring method. How components are structured depends on the way they are selected and the operations that apply to them.

Standard Mode:

Array	An indexed list of elements of the same data type.
File	A sequence of components of the same type.
Record	A list of named components, or fields, of related data that can be of different types.
Set	A collection of objects of an ordinal type.
Text	A predefined data type representing a file of character lines.

VS Mode:

Alfa	A predefined data type representing a fixed string of 8 characters.
Alpha	A predefined data type representing a fixed string of 16 characters.
Gstring	A predefined data type representing a packed array of multibyte characters (a graphic string) whose length varies at run time up to a maximum specified during compilation.
Space	A variable whose components can be positioned at any byte in the total storage area of the variable.
String	A predefined data type representing a packed array of characters whose length varies at run time up to a maximum specified during compilation.

Strings and Fixed Strings

A *fixed string* is a variable or constant that has an associated type of **PACKED ARRAY** [1..*n*] **OF CHAR**, where *n* is a positive integer constant. In standard mode XL Pascal, this value must be greater than 1, and any two strings compared or assigned must have the same length.

Note: The relative magnitude of two fixed strings is based on the collating sequence of ASCII.

The following operators are defined for fixed strings and MBCS fixed strings: =, <, < =, >=, >, <> or ~= (VS Mode only).

The following predefined routines apply to fixed string data in standard mode:

- **PACK**
- **UNPACK**

Strings in VS Mode

The following predefined routines apply to fixed string and MBCS data in VS mode:

- **ADDR**
- **HBOUND**
- **LBOUND**
- **SIZEOF**

In VS mode fixed strings, the upper bound of the array can equal 1. Strings being compared or assigned need not be the same length.

The predefined routine **STR** applies to fixed string data, and the predefined routine **GSTR** applies to MBCS fixed string data. An MBCS fixed string is a variable or constant that has an associated type of **PACKED ARRAY** [1..*n*] **OF GCHAR**, where *n* is a positive integer constant.

XL Pascal supports varying-length, or *dynamic* strings; that is, strings with lengths that can vary at run time. A variable can be declared as a varying-length string with the predefined type **STRING**. Throughout this book, the term *string* refers to an object of the predefined type **STRING**.

Notes:

1. Relative magnitude of two MBCS fixed strings is based on the binary value of MBCS codes.
2. If two strings or MBCS strings being compared are of different lengths, the shorter is assumed to be padded with blanks on the right until the lengths match.

Related Information

Sections in Chapter 10, “Routines,” describe the predefined routines in detail. A table showing the full ASCII character set is in the *User’s Guide for IBM AIX XL Pascal Compiler/6000*. More information about dynamic strings is in “STRING” on page 82.

Packed Types

For each variable declared with a particular type, XL Pascal allocates a specific amount of storage on a specific alignment boundary. The **PACKED** attribute directs XL Pascal to minimize the number of bytes of storage required for data of a given type. Packed data occupies less space and is more compact, but code compiled to use packed data is processed less efficiently, and programs with packed data may take longer to run.

Components of Packed Records and Arrays

Each component of a structured type usually has proper alignment. Offset is assigned sequentially, and the components are padded as necessary for boundary alignment as if they were separate variables.

Packing the array or record allocates storage more efficiently by leaving no unused bytes between components. Packed arrays and records have each component start in the next byte following the previous component.

Components of packed arrays or records that are arrays or records with named types do not inherit the packing attributes of the main, or *parent*, array or record. Each component has independent packing attributes. Therefore, if field type is a named type, field is not packed unless it is of a type declared as **PACKED**; elements of a packed array are not packed unless they are of a type declared as **PACKED**.

Packing saves space, but it affects normal alignment rules and restricts how components of packed records and arrays can be passed as parameters. Standard mode XL Pascal does not permit elements of packed arrays or records to be passed by **VAR** to user-defined procedures.

Sets

The **PACKED** attribute affects only sets whose base types are subrange types. Unpacked sets of subrange type always occupy 32 bytes as a 256-bit string. Packed sets of subrange type use only enough bytes to allocate one bit per value in the subrange.

Subranges

An unpacked subrange occupies as much storage as its base type. A packed subrange occupies just enough bytes to store the largest and smallest values in the subrange.

Other Packed Types

Declaring the **PACKED** attribute in type declarations other than records, arrays, sets, and subranges has no effect on storage. For example, a **PACKED FILE** is no different from a **FILE**.

Anonymous Types

Data types declared without a type name are said to be *anonymous* types. A field of a packed **RECORD** type that is an anonymous subrecord is implicitly packed. An anonymous array type that is a field of packed **RECORD** is not implicitly packed.

Related Information

The implementation requirements of packed types and other characteristics of data representation are described in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

Type Compatibility

XL Pascal supports *strong typing* of data; that is, objects of one type cannot be combined in operations with objects of a different type. Strong typing puts strict rules on data types that are the same. These rules define *type compatibility* and require you to declare data carefully. Strong typing permits XL Pascal to check the validity of many operations at compile time, which helps to produce reliable programs.

Implicit Type Conversion

In general, XL Pascal does not perform implicit type conversions on data. The following are the implicit conversions:

Implicit Type Conversion in Standard Mode:

- An **INTEGER** value is converted to a **REAL** value when one operand of a binary operation is an **INTEGER**, and the other is a **REAL**.
- An **INTEGER** value is converted to a **REAL** value when it is assigned to a **REAL** variable.
- An **INTEGER** value is converted to a **REAL** value if it is used in a floating-point divide operation (/).
- An **INTEGER** value is converted to a **REAL** value if it is passed by **VALUE** to a parameter requiring a **REAL** value.

Implicit Type Conversion in VS Mode:

- An **INTEGER** value is converted to a **SHORTREAL** value when one operand of a binary operation is an **INTEGER**, and the other is a **SHORTREAL**.
- An **INTEGER** value is converted to a **SHORTREAL** value when it is assigned to a **SHORTREAL** variable.
- An **INTEGER** value is converted to a **SHORTREAL** value if it is used in a floating-point divide operation (/) where the other operand is a **SHORTREAL**.
- An **INTEGER** value is converted to a **SHORTREAL** value if it is passed by **VALUE** or by **CONST** to a parameter requiring a **SHORTREAL** value.
- An **INTEGER** value is converted to a **REAL** value if it is passed by **CONST** to a parameter requiring a **REAL** value.
- A **SHORTREAL** value is converted to a **REAL** when one operand of a binary operation is a **SHORTREAL** and the other is a **REAL**.
- A **SHORTREAL** value is converted to a **REAL** when it is assigned to a **REAL** variable. A **REAL** value is converted to a **SHORTREAL** value when it is assigned to a **SHORTREAL** variable.
- A **SHORTREAL** value is converted to a **REAL** if it is passed by **VALUE** or by **CONST** to a parameter requiring a **REAL** value. A **REAL** value is converted to a **SHORTREAL** value if it is passed by **VALUE** or by **CONST** to a parameter requiring a **SHORTREAL** value.
- A **STRING** value is converted to a fixed string on assignment to a fixed-string variable. The string is padded with blanks on the right if it is shorter than the array to which it is being assigned. The **STRING** value is truncated on the right if it is longer than the array to which it is being assigned. Truncation causes a runtime error if checking is enabled.
- A **STRING** value being passed by **VALUE** or by **CONST** to a fixed string formal parameter is converted to a fixed string. The string is padded with blanks on the right if it is shorter than the array to which it is being passed. The **STRING** value is truncated on the right if it is longer.
- A **GSTRING** value is converted to an MBCS fixed string on assignment to an MBCS fixed-string variable. The MBCS string is padded with blanks on the right if it is shorter than the array to which it is being assigned. The **GSTRING** value is truncated on the right if it is longer than the array to which it is being assigned. Truncation causes a runtime error if checking is enabled.
- A **GSTRING** value being passed by **VALUE** or by **CONST** to an MBCS fixed string formal parameter is converted to an MBCS fixed string. The MBCS string is padded with blanks on the right if it is shorter than the array to which it is being passed. The **GSTRING** value is truncated on the right if it is longer than the array to which it is being passed.

Same Data Types

Two variables are said to be of the same type if the declarations of the variables are either of the following:

- Both refer to the same type identifier
- Both refer to different type identifiers defined as equivalent by a type definition of the form:

  ```
  TYPE T1 = T2
  ```

  where `T2` is a type identifier.

Compatible Data Types

You can do binary operations on two values of *compatible types*. Any object of type **SET** is compatible with the empty set. Any object that is a pointer type is compatible with the value **NIL**.

Standard Mode

Two types are compatible when any one of the following is true:

- Both types are the same.
- One type is a subrange of the other.
- Both types are subranges of the same type.
- One value is a character constant, the other is a fixed string and both have the same number of characters.
- Both are set types with compatible base types, and both are either packed or unpacked. In VS mode, the packing of the sets need not match.
- Both are fixed strings and both have the same number of characters. In VS mode, the number of characters need not be the same.

VS Mode

Two types are compatible when any one of the following is true:

- One value is an MBCS character constant, and the other is an MBCS fixed string
- Both are type **STRING** or both are type **GSTRING** of the same maximum length
- One value is a string literal, and the other is a fixed string
- One value is a string literal of one character, and the other is a **CHAR**
- One value is an MBCS string literal, and the other is an MBCS fixed string
- One value is an MBCS string literal of one character, and the other is a **GCHAR**

String constants are compatible with character, fixed string, or varying-length string values, assuming that all length requirements are met.

A packed array can be assigned to another packed array of a larger size.

Assignment Compatibility

You can assign a value to a variable if the types are *assignment compatible*. In the assignment statement V := E, an expression E is assignment-compatible with variable V when any one of the following is true.

Standard Mode

- Both V and E are the same type, and neither V nor E is a file type nor contains a file type
- V is of type **REAL**, and E is compatible with type **INTEGER**
- V is a compatible subrange of E, and the value to be assigned is within the allowable subrange of V
- V and E have compatible set types, and all members of E are permissible members of V
- Both V and E are fixed strings of the same length

VS Mode

- V is of type **REAL** or **SHORTREAL**, and E is compatible with type **INTEGER**
- V is an MBCS fixed string, and E is a **GSTRING** whose current length is less than or equal to the length of V
- Both V and E are MBCS fixed strings of the same length
- V is type **REAL**, and E is type **SHORTREAL**
- V is type **SHORTREAL**, and E is type **REAL**
- V is a fixed string, and E is a dynamic string whose current length is less than or equal to the length of V
- Both V and E are type **STRING** or **GSTRING**, and the current length of E is less than or equal to the maximum length of V
- Both V and E are packed arrays of characters, and the size of V is less than the size of E
- V is type **POINTER**, and E is any pointer type

Examples

Given the following declarations:

```
TYPE
   x = ARRAY [1..10] OF INTEGER;
   days = ( mon, tues, wed, thurs, fri, sat, sun );
   weekday = mon..fri;

VAR
   a : ARRAY [1..10] OF INTEGER;
   b : ARRAY [1..10] OF INTEGER;
   c, d : ARRAY [1..10] OF CHAR;
   e : x;
   f : x;
   w1 : days;
   w2 : weekday;
```

the following type compatibilities apply:

Variable	**Is Compatible With**	**Has the Same Type As**
a	a	a
b	b	b
c	c, d	c, d
d	d, c	d, c
e	e, f	e, f
f	f, e	f, e
w1	w1, w2	w1
w2	w2, w1	w2

Creating Your Own Data Types

Using the **TYPE** definition, you can create your own data types. You can then use the identifiers for these new data types in type declarations for variables. You might want to define a data type `color` with the values `yellow`, `cyan`, and `magenta`, and then define a variable `ink` as being type `color`, as shown in the following example:

```
TYPE
    color = ( yellow, cyan, magenta) ;

VAR
    ink : color ;
begin
    .
    .
    ink := cyan ;
    .
    .
end ;
```

A type identifier such as `color` can be used wherever a type definition is needed:

- In a variable declaration (**VAR**, **STATIC**, **DEF**, or **REF**)
- As the type of a formal parameter
- As a result type in a function
- In a field declaration within a record definition
- In another **TYPE** declaration

A type has an associated size but reserves no storage itself. Storage is only reserved when you declare a variable as an instance of that type.

Summary of Data Types

Standard Mode

Data Type	Subtypes	Consists Of
Enumerated scalar		A list of permitted values
Subrange scalar		A subset of consecutive values of a previously defined ordinal type
Predefined scalar	BOOLEAN	An enumerated type with the values (FALSE,TRUE)
	CHAR	All the values of the ASCII character set
	INTEGER	The subset of the whole numbers from negative MAXINT to MAXINT (2147483647)
	REAL	Double-precision floating-point data
ARRAY		A collection of homogeneous elements
FILE		A one-dimensional sequence of components of the same type
RECORD		A collection of heterogeneous elements
SET		A collection of values taken from the same ordinal type
Predefined structure	TEXT	A file of character lines
Pointer		The address of a dynamic variable

VS Mode

Data Type	**Subtypes**	**Consists Of**
Predefined scalar	GCHAR	All the values of the multibyte character set (MBCS)
	SHORTREAL	Single-precision floating-point data
	INTEGER	The subset of the whole numbers from MININT (-2147483648) to MAXINT (2147483647)
SPACE		A storage allocation for data of varying length
Predefined structure	ALFA	A fixed string of 8 characters
	ALPHA	A fixed string of 16 characters
	GSTRING	An MBCS fixed string whose length varies up to a specified maximum
	STRING	A fixed string whose length varies up to a specified maximum
Predefined Pointers	POINTER	A value assignment compatible with any pointer type
	GSTRINGPTR	A pointer to a variable of type GSTRING
	STRINGPTR	A pointer to a variable of type STRING

Related Information

Operators are described on page 100. The predefined routines are described in Chapter 10, “Routines”.

ALFA (VS Mode)

Purpose

The predefined type **ALFA** is defined as:

```
CONST
   ALFALEN = 8;

TYPE
   ALFA = PACKED ARRAY [1..ALFALEN] OF CHAR;
```

Description

The **ALFA** data type is a predefined 8-character fixed string. The predefined constant **ALFALEN** has a value of 8.

Operations

The following operators and predefined functions apply to the **ALFA** data type.

- Operators =, <, <=, >=, >, <> or ~=
- **ADDR**
- **HBOUND**
- **LBOUND**
- **SIZEOF**
- **STR**

ALPHA (VS Mode)

Purpose

The predefined type **ALPHA** is defined as:

```
CONST
   ALPHALEN = 16;

TYPE
   ALPHA = PACKED ARRAY [1..ALPHALEN] OF CHAR;
```

Description

The **ALPHA** data type is a predefined 16-character fixed string. The predefined constant **ALPHALEN** has a value of 16.

Operations

The following operators and predefined functions apply to the variables of the predefined type **ALPHA**.

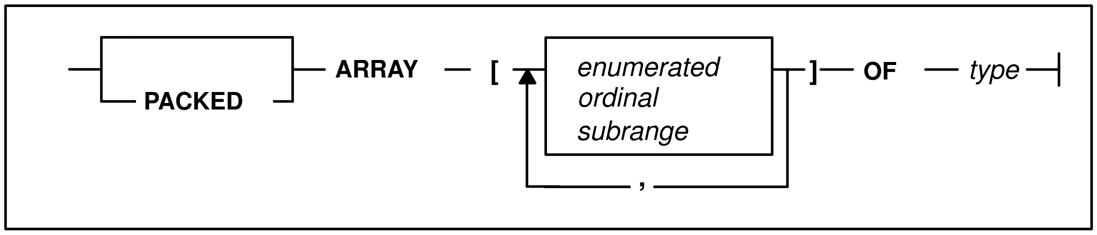
- Operators =, <, <=, >=, >, <> or ~=
- **ADDR**
- **HBOUND**
- **LBOUND**
- **SIZEOF**
- **STR**

ARRAY

Purpose

Defines a list of homogeneous elements in which each element is paired with one value of an index. The index can be any finite ordinal type.

Syntax



Parameters

enumerated	is an enumerated scalar data type.
ordinal	is an ordinal type name.
subrange	is a subrange data type.
type	is any type.

Description

An element of the array is accessed through its subscript. To subscript a variable, you must specify an index. The number of elements in the array is the number of values potentially assumable by the index. The index type cannot define more than **MAXINT** potentially assumable values.

Each element of the array is of the same type, called the *element type* of the array. The element type can be any valid XL Pascal type (including **FILE** types). Entire arrays can be assigned if they are of the same type.

Pascal uses square brackets [and] in the declaration of arrays. Because these symbols are not directly available on many I/O devices, you can use (. and .) as an alternative to the square brackets.

An array defined with more than one index is said to be *multidimensional*. Such an array is equal to an array of arrays. For example, the following is an array definition:

```
ARRAY [i,j,...] OF t
```

The following is an abbreviated form of the array definition:

```
ARRAY [i] OF
   ARRAY [j] OF
      ... t
```

where `i` and `j` are scalar type definitions.

Operations

The following predefined routines operate on the **ARRAY** data type:

Standard Mode:

- **PACK**
- **UNPACK**

VS Mode:

- **ADDR**
- **HBOUND**
- **LBOUND**
- **SIZEOF**

Examples

In the following example, the first and second type declarations are alternatives for the same structure.

```
TYPE
   matrix = ARRAY [1.. 10, 1..10] OF REAL;
   matrix0 = ARRAY [1..10] OF
      ARRAY [1..10] OF REAL;
   able = ARRAY [BOOLEAN] OF INTEGER;
   color = ( red, yellow, blue );
   intensity = PACKED ARRAY [color] OF REAL;
   ALFA = PACKED ARRAY [1..ALFALEN] OF CHAR;
```

BOOLEAN

Purpose

Is a predefined enumerated scalar whose constant values are **FALSE** and **TRUE** as defined in the following type declaration:

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

Operations

The logical operators shown in the following table form Boolean functions.

Name	**Operation**	**Result**
AND (&)	FALSE & FALSE	FALSE
	FALSE & TRUE	FALSE
	TRUE & FALSE	FALSE
	TRUE & TRUE	TRUE
OR (|)	FALSE | FALSE	FALSE
	FALSE | TRUE	TRUE
	TRUE | FALSE	TRUE
	TRUE | TRUE	TRUE
NOT (~)	~FALSE	TRUE
	~TRUE	FALSE
XOR (&&)	FALSE && FALSE	FALSE
	FALSE && TRUE	TRUE
	TRUE && FALSE	TRUE
	TRUE && TRUE	FALSE

The predefined functions that operate on enumerated scalar types also apply to type **BOOLEAN**.

The following operators apply to the standard type **BOOLEAN**:

Standard Mode:

- =, <>, <, <=, >=, >
- **NOT**
- **AND**
- **OR**

VS Mode:

- ~, &, |, ~=
- **XOR**, &&

XL Pascal makes the evaluation of Boolean expressions involving **AND** (&) and **OR** (|) more efficient so that the right operand of the expression is not evaluated if the result of the operation can be determined by evaluating the left operand.

Related Information

Boolean Expressions are described on page 106.

CHAR

Purpose

Is a predefined ordinal type consisting of all of the values of the ASCII character set. Variables of this type occupy 1 byte of storage and are aligned on a byte boundary.

If the context so dictates, a string constant with a single character is a **CHAR** constant. For example, the following assignment statement sets variable `c` to the ASCII code for the character `'A'`:

```
VAR
   c : CHAR;

BEGIN
   ...
   c := 'A';
   ...
END;
```

Operations

The following operations and predefined functions apply to the standard type **CHAR**:

Standard Mode:

- The operators =, <>, <, <=, >=, >
- **ORD**
- **PRED**
- **SUCC**

VS Mode:

- The operator ~=
- **ADDR**
- **HIGHEST**
- **LOWEST**
- **MAX**
- **MIN**
- **SIZEOF**
- **STR**

VS Mode Predefined Character Constant

VS mode provides a predefined constant, **MAXCHAR**, representing the maximum value of the type **CHAR**. Its value is 'FF'XC.

Related Information

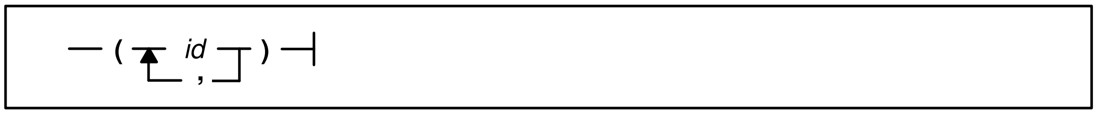
The tables on page 83 describe applying relational operators to characters and converting characters on assignment. A table showing the full ASCII character set is in the *User's Guide for IBM AIX XL Pascal Compiler/6000.*

Enumerated Scalar

Purpose

Is formed by listing each value permitted for a particular type of variable. A meaningful name is associated with each value.

Syntax



Parameter

id	is an identifier treated as a self-defining constant.

Description

An enumerated scalar data type definition declares the identifiers in the enumeration list as constants of the same type as the enumerated scalar being defined. The lexical scope of the newly defined constants is the same as any other identifier declared explicitly at the same lexical level. These constants are ordered so that the first value is less than the second, the second less than the third, and so forth.

Note: Two enumerated scalar type definitions must not have any elements of the same name in the same lexical scope.

Operations

The following predefined functions operate on enumerated scalars.

Standard Mode:

- **ORD**
- **PRED**
- **SUCC**

VS Mode:

- **ADDR**
- **HIGHEST**
- **LOWEST**
- **MAX**
- **MIN**
- **SIZEOF**

Examples

In the type declarations in the following example, no value is less than the first, or greater than the last.

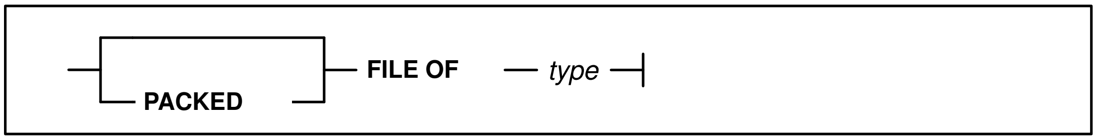
```
TYPE
   days = ( mon, tues, wed, thurs, fri, sat, sun );
   months = ( jan, feb, mar, apr, may, jun,
              jul, aug, sep, oct, nov, dec );
VAR
   shape : ( triangle, rectangle, square, circle );
   rec : record
            suit : ( spade, heart, diamond, club );
            day : days
         END;
   month : months;
```

FILE

Purpose

A *file* is a structure consisting of a sequence of components where each component is of the same type.

Syntax



Parameter

type	is any data type not containing a file

Description

Input and output in Pascal are usually done through a file. Variables of this type refer to the components with pointers called *file pointers*. A file pointer can be thought of as a pointer into an input/output buffer.

Declaring a file **PACKED** has no effect on its storage requirements.

The association of a file variable to an actual file of the system is implementation dependent, and is described in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

Operations

The following predefined routines allow access to file variables.

Standard Mode:

- **EOF**
- **GET**
- **PUT**
- **READ**
- **RESET**
- **REWRITE**
- **WRITE**

VS Mode:
- **ADDR**
- **CLOSE**
- **SEEK**
- **SIZEOF**
- **UPDATE**

Examples

```
TYPE
   line = FILE OF PACKED ARRAY [1..80] OF CHAR;
   pfile = FILE OF RECORD
              name : PACKED ARRAY [1..25] OF CHAR;
              person_no : INTEGER;
              date_employed : PACKED ARRAY [1..8] OF CHAR;
              weekly_salary : INTEGER
           END;
```

GCHAR (VS Mode)

Purpose

The predefined data type **GCHAR** is a scalar representing an MBCS character in the AIX National Language operating environment established during the program run. Variables of this type occupy 2 bytes of storage.

Because values of **GCHAR** are not mapped on consecutive integers, **GCHAR** is not an ordinal type. The **GCHAR** data type cannot be used in the following situations:

- In subranges or sets of **GCHAR**
- As an array index type
- As a **CASE** selector
- As the type name of an ordinal conversion routine
- As the type of variable in a **FOR** loop index
- As a type of variant selector
- In the predefined functions **SUCC**, **PRED**, **ORD**, **HIGHEST**, and **LOWEST**

In certain contexts, an MBCS string constant is regarded as a **GCHAR** constant. For example, the following assignment statement sets variable `c` to the MBCS code value for the multibyte character **D**:

```
VAR
   c: GCHAR;

BEGIN
   ...
   c := 'D'G;
   ...
END;
```

Note: **D** represents one MBCS character.

Operations

The following operators and predefined functions apply to the data type **GCHAR**:

- The operators =, <>, ~=, <, <=, >=, >
- **ADDR**
- **GSTR**
- **MAX**
- **MIN**
- **SIZEOF**

Related Information

Applying relational operators to MBCS strings and MBCS fixed strings is described on page 66.

Converting MBCS strings and MBCS fixed strings on assignment is described on page 66.

GSTRING (VS Mode)

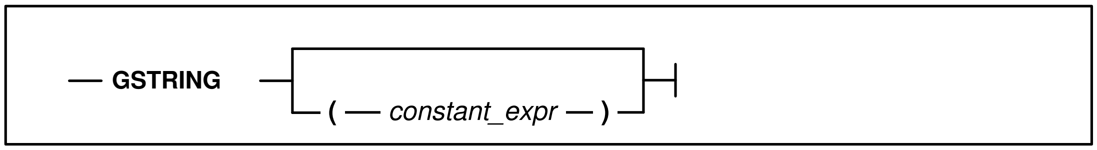
Purpose

The predefined data type **GSTRING** is defined as:

```
TYPE
   GSTRING = PACKED ARRAY [1..n] OF GCHAR;
```

The length of **GSTRING** varies at run time up to a compile-time specified maximum given by the constant expression.

Syntax



Parameter

constant_expr is any constant expression.

Description

A variable declared as **GSTRING** is a string of multibyte characters. The constant expression gives the maximum number of **GCHAR**s that **GSTRING** can contain. The default maximum is 255. The lower bound is always 1.

Assignment and concatenation operations determine the number of **GCHAR** elements considered active. This number must not exceed the maximum specified at the time you declared the variable. The maximum upper bound for **GSTRING** is 16382.

The length of the array is obtained during run time by the **LENGTH** function. The length is managed implicitly by the operators and functions that apply to **GSTRING**s. The maximum length of the array is obtained during run time by the **MAXLENGTH** function. The length of a **GSTRING** variable is determined when the variable is assigned.

A **GSTRING** variable can be subscripted with an integer expression to refer to a multibyte characters. A subscript of 1 refers to the first multibyte character. The subscript value must neither be less than 1, nor exceed the length of the **GSTRING**.

Any variable of type **GSTRING** is compatible with any other variable of type **GSTRING**. That is, the maximum length field of a type definition has no bearing in type compatibility tests. Implicit conversion is performed when a **GSTRING** is assigned to a variable whose type is **PACKED ARRAY** [1..*n*] **OF GCHAR**. All other conversions must be done explicitly.

The assignment of one **GSTRING** to another may cause a runtime error if the length of the source **GSTRING** is greater than the maximum length of the target.

Operations

The following operators and predefined routines apply to variables of type **GSTRING**:

- Operators =, <>, ~=, <, <=, >=, >, ||, +
- **ADDR**
- **COMPRESS**
- **DELETE**
- **GSTR**
- **GTOSTR**
- **HBOUND**
- **INDEX**
- **LBOUND**
- **LENGTH**
- **LPAD**
- **LTRIM**
- **MAXLENGTH**
- **PACK**
- **RINDEX**
- **RPAD**
- **SIZEOF**
- **SUBSTR**
- **TRIM**
- **UNPACK**

Note: Both operands must be of type **GSTRING**.

Applying Relational Operators to MBCS Data

The following table shows how to apply relational operators to variables of type **GCHAR** and **GSTRING** and to MBCS fixed strings:

Left Operand	**Right Operand**	**Result**
GCHAR	GCHAR	Allowed
	PACKED ARRAY OF GCHAR	Not permitted
	GSTRING	Use GSTR on the GCHAR
PACKED ARRAY OF GCHAR	GCHAR	Not permitted
	PACKED ARRAY OF GCHAR	Allowed if operands are type compatible
	GSTRING	Use GSTR on the array
GSTRING	GCHAR	Use GSTR on the GCHAR
	PACKED ARRAY OF GCHAR	Use GSTR on the array
	GSTRING	Allowed

Converting MBCS Strings on Assignment

Target Variable	**Source Expression**	**Result**
GCHAR	GCHAR	Allowed
	PACKED ARRAY OF GCHAR	Not permitted
	GSTRING	Use string indexing to obtain a GCHAR
PACKED ARRAY OF GCHAR	GCHAR	Not permitted
	PACKED ARRAY OF GCHAR	Allowed if operands are type compatible
	GSTRING	GSTRING is converted. If truncation is required, an error results.
GSTRING	GCHAR	Use GSTR to convert the GCHAR to a GSTRING
	PACKED ARRAY OF GCHAR	Use GSTR to convert the array to a GSTRING
	GSTRING	Allowed

Examples

```
FUNCTION Getgchar( CONST s : GSTRING ; idx : INTEGER ) : GCHAR;
   BEGIN
      Getgchar := s[idx]; (* subscripted GSTRING object *)
   END;
   .
   .
   VAR
      gs1 : GSTRING( 10 );
      gs2 : GSTRING( 5 );
      gc : GCHAR;

   BEGIN
      gs1 := 'DDDDD'G;       (* for this example assume  *)
      .                       (* that D represents a     *)
      .                      (* multibyte character      *)
      gs2 := 'BBBB'G;
      gs1 := gs1 || gs2;       (* pad gs1 with blanks    *)
      gc := getgchar( gs1, 4 );(* gets the fourth GCHAR  *)
                               (* from gs1               *)
   END;
```

GSTRINGPTR (VS Mode)

Purpose

Defines a pointer to a dynamic MBCS string variable, which is a **GSTRING** with no maximum length associated with it until run time.

GSTRINGPTR is equivalent to:

```
TYPE
  GSTRINGPTR = @GSTRING;
```

The procedure **NEW** allocates storage for the **GSTRINGPTR** data type. An integer expression is passed to the procedure that specifies the maximum length of the allocated **GSTRING**.

Variables of type **GSTRING** have two lengths associated with them:

- The current length that defines the number of multibyte characters in the **GSTRING**
- The maximum length that defines the storage required for the **GSTRING**

Operations

The following operators and predefined routines are valid for the **GSTRINGPTR** data type:

- Operators =, <>, ~=
- **ADDR**
- **DISPOSE**
- **DISPOSEHEAP**
- **MARK**
- **NEW**
- **NEWHEAP**
- **QUERYHEAP**
- **RELEASE**
- **SIZEOF**
- **USEHEAP**

Example

```
VAR
   gp : gstringptr;
   i : 1..16000;

BEGIN
   i := 50;
   NEW( gp, i ); (* create new gstring variable that can     *)
                 (* contain up to 50 GCHARs. Set gp to       *)
                 (* point to that variable                   *)
   READLN( gp@ );(* read from INPUT, GCHAR data into the     *)
                 (* new string                               *)
   WRITELN( LENGTH ( gp@ ) )(* write the length of data read *)
END;
```

INTEGER

Purpose

Represents the subset of whole numbers defined as follows:

```
TYPE
   INTEGER = MININT..MAXINT;
```

where **MININT** is a VS mode predefined integer constant whose value is –2147483648, and **MAXINT** is a predefined integer constant whose value is 2147483647. The predefined type **INTEGER** represents 32-bit values in 2's complement notation. Variables of the type **INTEGER** are word aligned.

Operations

The following operations and predefined functions apply to values of the standard type **INTEGER**:

Standard Mode:

- The operators **DIV**, **MOD**, +, –, *, /, =, <>, <, <=, >=, >
- **ABS**
- **CHR**
- **ODD**
- **PRED**
- **SQR**
- **SUCC**

VS Mode:

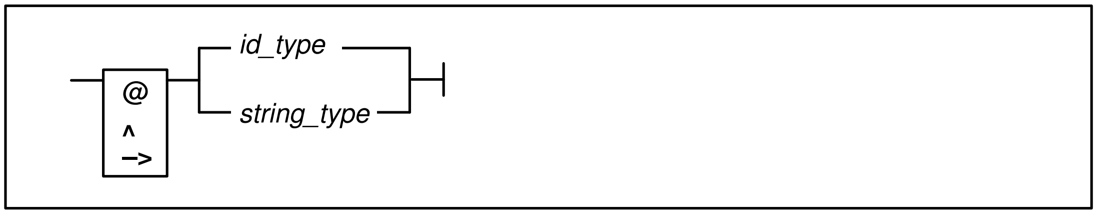
- The operators **NOT**, ~, **OR**, |, **AND**, &, **XOR**, &&, <<, >>, ~=
- **FLOAT**
- **HIGHEST**
- **LOWEST**
- **MAX**
- **MIN**
- **SIZEOF**

Pointer Data Types

Purpose

Track *dynamic variables* by maintaining their addresses in storage using *pointer variables*. The predefined procedure **NEW** lets you create dynamic variables, which are variables under your explicit control during program run time.

Syntax



Parameters

id_type	is any type.
string_type	is any string data type.

Note: The pointer reference symbol –> is available in VS mode only.

Description

NEW creates a new variable of the appropriate type and assigns its address to the argument of **NEW**. You must explicitly deallocate a dynamic variable with either of the predefined procedures **DISPOSE** and **RELEASE**; otherwise, dynamic variables are deallocated at the end of the program.

Pointers are constrained to point to a particular type. When you declare a pointer you must specify the type of the dynamic variable that is generated or referenced.

XL Pascal defines the named constant **NIL** as the value of an *empty pointer*, which is a pointer that does not point to any dynamic variable. The **NIL** is type-compatible to every pointer type.

Operations

The only operators that can be applied to variables of pointer types are the tests for equality and inequality. The following operators and predefined routines apply to the pointer data types.

Standard Mode:

- Operators =, <>
- **DISPOSE**
- **NEW**

VS Mode:

- Operator ~=
- **ADDR**
- **DISPOSEHEAP**
- **MARK**
- **NEWHEAP**
- **QUERYHEAP**
- **RELEASE**
- **SIZEOF**
- **USEHEAP**

Example

In the following example, the data type `element` is referred to before it is declared. Usually you must not refer to an identifier before declaring it. A type identifier used as the base type in a pointer declaration is, however, an exception to this rule. The example illustrates a data type used to build a tree structure.

```
TYPE
   ptr = @ element;
   element = RECORD
                parent : ptr;
                child : ptr;
                sibling : ptr
             END;
```

POINTER (VS Mode)

Purpose

Is assignment compatible with any pointer type. This predefined data type does not have a target type. A variable of type **POINTER** can be assigned to a variable of any pointer type. The **NEW** and **DISPOSE** procedures cannot be used on a parameter of type **POINTER**.

Operations

The following operators and predefined routines are valid for the **POINTER** data type:

- Operators =, <>, ~=
- **ADDR**
- **SIZEOF**

Example

```
TYPE
   reckind = ( big, small );
   smallrec = RECORD
                 .
                 .
                 END;
   bigrec = RECORD
               .
               .
               END;

PROCEDURE process_record( ptrparam : POINTER;
                          whichkind : reckind );

   VAR
      bigptr : @bigrec;
      smallptr : @smallrec;

   BEGIN
      CASE whichkind OF
         big : BEGIN
                  bigptr := ptrparam;
                  (* process object "bigptr@" of type "bigrec" *)
               END
         small : BEGIN
                    smallptr := ptrparam;
                    (* process object "smallptr@" *)
                    (* of type "smallrec"         *)
                 END ;
      END ;                                  (* CASE *)
   END ;                (* BEGIN - process_record *)
```

In procedure `process_record`, parameter `ptrparam` is of type **POINTER** and is compatible with any pointer type. Parameter `whichkind` determines whether `ptrparam` points to a variable of type `bigrec` or `smallrec`. If you assign `ptrparam` to a pointer variable with the same target type as the variable `ptrparam` points, you can process that variable. You cannot process `ptrparam@` directly as a variable because it does not have a specified type.

REAL

Purpose

Represents floating-point data. Variables of this type occupy 8 bytes of storage and are aligned on a double word boundary. All **REAL** arithmetic is done using double-precision floating-point instructions.

Operations

The following operations and predefined functions apply to values of type **REAL**:

Standard Mode:

- Operators +, –, *, /, =, <>, <, <=, >=, >
- **ABS**
- **ARCTAN**
- **COS**

- **EXP**
- **LN**
- **ROUND**
- **SIN**
- **SQR**
- **SQRT**
- **TRUNC**

VS Mode:

- Operator ~=
- **ADDR**
- **MAX**
- **MIN**
- **SIZEOF**

VS Mode Predefined Constants

XL Pascal provides three predefined **REAL** constant values in VS mode:

- **MAXREAL** is a predefined constant whose value is the largest floating-point magnitude of the type **REAL**.
- **MINREAL** is a predefined constant whose value is the smallest nonzero floating-point magnitude of the type **REAL**.
- **EPSREAL** is the smallest **REAL** value that, when added to 1, is detectable.

Restrictions

The **REAL** type is not ordinal. It has restrictions that other scalar types do not have and cannot be used under the following circumstances:

- In subranges or sets of **REAL**
- As an array index type
- In the predefined functions **SUCC**, **PRED**, **ORD**, **HIGHEST**, and **LOWEST**
- As the type of variable in a **FOR** loop index
- As a **CASE** selector
- As the type of variant selector

Example

In the following example, `n` cannot be **REAL** in the subrange `[1..n]`, and the variable `i` cannot be of type **REAL**.

```
VAR
   reals : ARRAY [1..n] OF REAL;
     .
     .
BEGIN
   reals [i] := 3.14159
```

Related Information

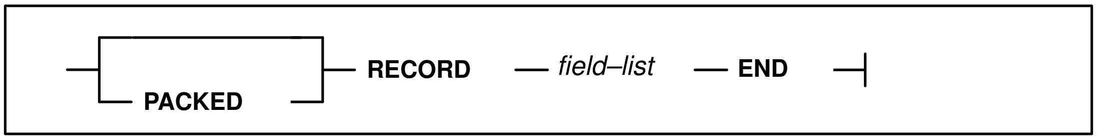
Other data types are converted to **REAL** under some operations, as described in "Implicit Type Conversion" on page 50.

RECORD

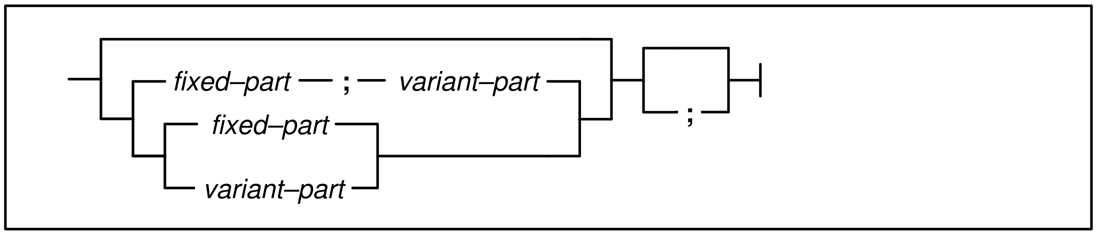
Purpose

A record is a data structure composed of heterogeneous components called *fields*. Each field can be of a different type.

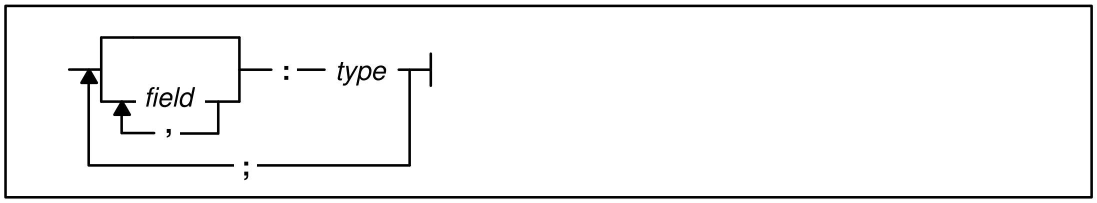
Syntax



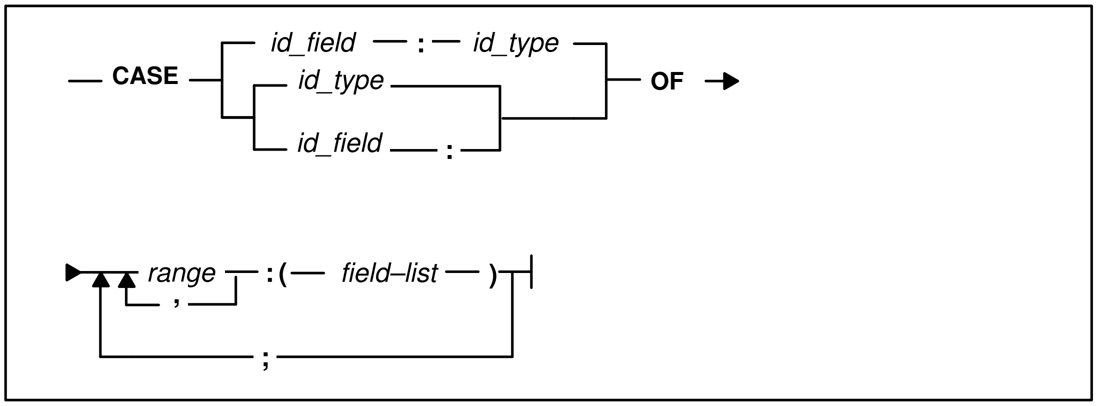
Field–list



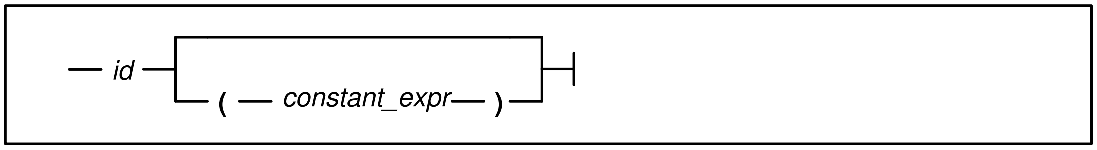
Fixed–part



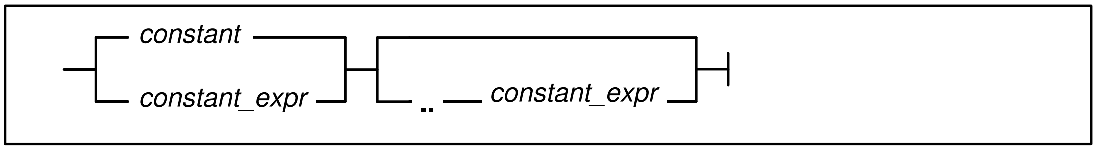
Variant–part



Field



Range



Parameters

id	is the name of variable
type	is the type of record field
id_type	is the type of tag field
constant	is any constant
id_field	is a tag field identifier (The *id_field* without an *id_type* parameter is used in VS mode only)
constant_expr	is any constant expression (VS mode only)

Operations

In VS mode only, the following predefined functions operate on the **RECORD** data type:

- **ADDR**
- **SIZEOF**

Naming a Field

A field is referred to by its name. The scope of a field name is within the record to which the field belongs. You can use the same field identifier in more than one record, but every field name within a record must be unique, even if that name appears in a variant part. No field name can be the same as that of any field type in the same record.

In VS mode, the field of a record need not be named, and the field identifier may be missing. In such a case, the field serves only as padding and cannot be referred to.

Examples

```
TYPE
   rec = RECORD
            a, b : INTEGER;
                 : CHAR;   (* unnamed; VS mode only *)
            c : CHAR
         END;
   date = RECORD
             day : 1..31;
             month : 1..12;
             year : 1900..2100
          END;
   person = RECORD
               last_name, first_name : ALFA;
               middle_initial : CHAR;
               age : 0..99;
               employed : BOOLEAN
            END;
```

Fixed Part

The fixed part of a record is a series of fields in every variable declared to be of that record type. If present, it always appears before the variant part.

Variant Selector

The *variant selector* follows the reserved word **CASE** in the variant part of the record. This is an ordinal type that indicates which variant of the record is active.

When the variant selector is followed by a colon and a type, a new *tag field* is defined within the record. For example, the following results in `i` being a tag field of type **INTEGER**:

```
CASE i: INTEGER OF
```

The variant part of a record need not have a tag field at all. In this case, only a type identifier is specified in the **CASE** construct. For example, the following means no tag field is present:

```
CASE INTEGER OF
```

The variants are denoted by integer values in the variant declaration. You must still refer to the variant fields by their names, but it is your responsibility to keep track of which variant is active (that is, contains valid data) at run time.

Variant Selectors in VS Mode

If the type identifier of the tag field is missing, the tag field name must be one previously defined within the record. The tag field can appear anywhere in the fixed part of the record. For example, the following means that `i` is the tag field and it must have been declared in the fixed part:

```
CASE i: OF
```

The type of `i` is given in the field definition of `i`.

Variant Part

The variant part of a record permits you to define an alternative structure in the record. The record structure adopts one of the variants at a time.

All variant tags must be assignment compatible with the tag type, and all possible values of a tag type must correspond to a variant. A variant part can only occur after the fixed part, and it can only occur once within each record.

Variant Parts in VS Mode

You can omit the tag constants you do not want to be used.

Examples

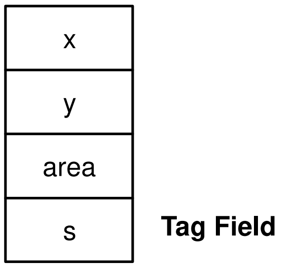
```
TYPE
   shape = ( triangle , rectangle , square , circle );
   coordinates = (* fixed part: *)
                 RECORD
                    x, y : REAL;
                    area : REAL;
                    CASE s : shape OF
                       (* variant part: *)
                       triangle : ( side : REAL;
                                    base : REAL );
                       rectangle : ( sidea, sideb : REAL );
                       square : ( edge : REAL );
                       circle : ( radius : REAL )
                 END;
```

In this example, the record type defined as `coordinates` contains a variant part. The tag field is `s`, its type is `shape`, and its value (whether `triangle`, `rectangle`, `square`, or `circle`) indicates which variant is in effect. The fields `side`, `sidea`, `edge`, and `radius` occupy the same offset within the record.

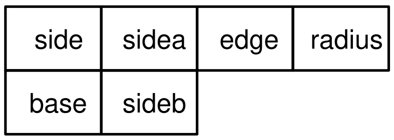
The following figure shows how the record would look in storage.

Storage of a Record with a Tag Field

Fixed Part:



Variant Part:



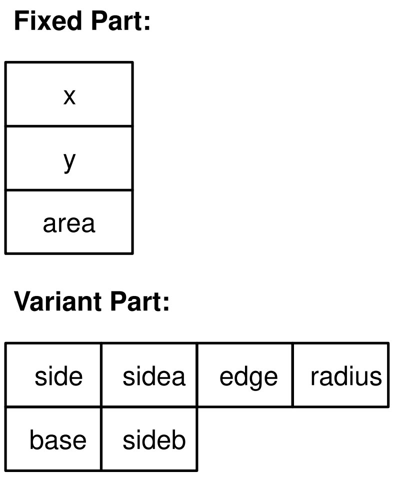
Each column in the variant represents one alternative for the variant.

If you prefer the tag field to be absent altogether, define the record as follows:

```
coordinates = RECORD
                 x, y : REAL;
                 area : REAL;
                 CASE shape OF
                    (* variant part: *)
                    triangle : ( side : REAL;
                                 base : REAL );
                    rectangle : ( sidea, sideb : REAL );
                    square : ( edge : REAL );
                    circle : ( radius : REAL )
              END;
```

The following figure shows how the record would look in storage.

Storage of a Record Variant with No Tag Field

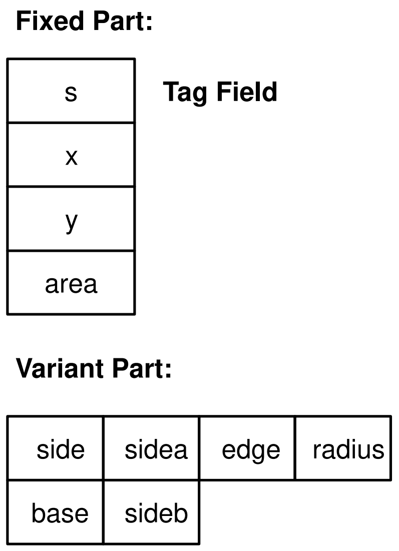


In VS mode only, if you prefer the tag field to be the first field instead of the fourth, define it as follows:

```
coordinates = RECORD
                 s : shape;
                 x, y : REAL;
                 area : REAL;
                 CASE s : OF
                    (* variant part: *)
                    triangle : ( side : REAL;
                                 base : REAL );
                    rectangle : ( sidea, sideb : REAL );
                    square : ( edge : REAL );
                    circle : ( radius : REAL )
              END;
```

The following figure shows how the record would look in storage.

Storage of a Record with a Back-Reference Tag Field



Offset Qualification of Fields (VS Mode Only)

XL Pascal allows you to force the fields of a record to begin at a specified byte offset in the record. A field name can be followed by an integer constant expression enclosed in parentheses. This expression represents the byte offset within the record where the field begins. All fields so specified must be in consecutive order according to offsets. If the offset is not specified and the record is not packed, the field is assigned the next offset required for boundary alignment. If the record is packed, the field is byte-aligned to the next available offset. If an offset specification tries to assign an incorrect boundary for a field and the record is not packed, a compile-time error message is issued.

Examples

Assume that a large control block of 100 bytes is needed in which four fields at various offsets must be referenced. The fields of the control block, and how the control block can be represented in XL Pascal, are shown in the following example.

Byte Displacement	Information
0	Field a (integer)
36	Field b (8 characters)
80	Field c (4 flags)
92	Field d (integer)

```
TYPE
   flags = SET OF ( f1 , f2 , f3 , f4 );
   padding = PACKED ARRAY [1..4] OF CHAR;
   cb = PACKED RECORD
           a : INTEGER;
           b( 36 ) : ALFA;
           c( 80 ) : flags;
           d( 92 ) : INTEGER;
                   : padding
        END;

VAR
   block : cb;
```

You cannot use an offset qualifier on the variant part tag field. To set the tag field at a certain offset, make the tag field a backward reference, give the last identifier of the fixed part the same name as the tag field, and put the offset qualifier on this last identifier. The following example illustrates this:

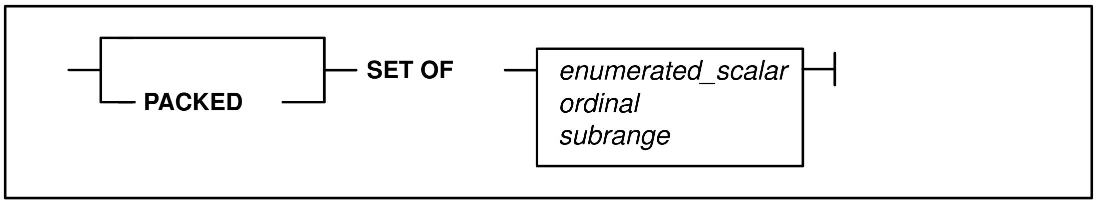
```
TYPE tag = PACKED RECORD
              a : BOOLEAN;
              b( 2 ) : BOOLEAN;
              CASE b : OF
                 TRUE : ( c : CHAR );
                 FALSE : ( d : REAL )
           END;

VAR
   block : TAG;
```

SET

Purpose

Contains any combination of values taken from the *base scalar type*.

Syntax



Parameters

enumerated_scalar	is an enumerated scalar data type.
ordinal	is an ordinal type name.
subrange	is a subrange data type.

Description

A value is either in the set or it is not in the set. XL Pascal sets can be used in many of the same ways as bit strings. Each bit corresponds to one element of the base type, and is set to a binary one when that element is a member of the set. For example, a set operation such as intersection, whose operator is an asterisk (*), is the same as taking the Boolean **AND** of two bit strings.

Operations

The following operators and predefined functions apply to variables of the type **SET**:

Standard Mode:

- The operators **IN**, =, <>,< =, >=, +, *, –

VS Mode:

- The operators **XOR**, **NOT**, &&, ~=, ~
- **ADDR**
- **SIZEOF**.

Example

```
TYPE
   days = ( monday, tuesday, wednesday, thursday, friday );
   chars = SET OF CHAR;
   daysofmon = PACKED SET OF 1..31;
   daysofweek = SET OF monday..friday;
   flags = SET OF ( a, b, c, d, e, f, g, h );
```

SHORTREAL (VS Mode)

Purpose

Represents floating-point data. Variables of this type occupy 4 bytes of storage and are aligned on a word boundary.

To pass a **SHORTREAL** as an operand to a function or procedure that requires a **REAL** parameter, that parameter must be passed by value or by **CONST**.

Predefined Constants

XL Pascal provides three predefined constant **SHORTREAL** values:

- **MAXSREAL** is a predefined constant whose value is the largest floating-point magnitude of the type **SHORTREAL**.
- **MINSREAL** is a predefined constant whose value is the smallest nonzero floating-point magnitude of the type **SHORTREAL**.
- **EPSSREAL** is the smallest **SHORTREAL** value that, when added to 1, is detectable.

Operations

Operations between data of type **REAL** and **SHORTREAL** are done using double-precision floating-point instructions. The **SHORTREAL** operand in such operations is implicitly converted to a **REAL**, as described in “Implicit Type Conversion” on page 50.

The following operations and predefined functions apply to values declared as **SHORTREAL**:

- The operators +, –, *, /, =, <>, ~=, <, <=, >=, >
- **ABS**
- **ADDR**
- **ARCTAN**
- **COS**
- **EXP**
- **LN**
- **MAX**
- **MIN**
- **ROUND**
- **SIN**
- **SIZEOF**
- **SQR**
- **SQRT**
- **TRUNC**

Restrictions

The **SHORTREAL** type is not ordinal; therefore, it has restrictions other scalar types do not have. It cannot be used under the following circumstances:

- In subranges or sets of **SHORTREAL**
- As an array index type
- In the predefined functions **SUCC**, **PRED**, **ORD**, **HIGHEST**, and **LOWEST**
- As the type of variable in a **FOR** loop index
- As a **CASE** selector
- As the type of variant selector

SPACE (VS Mode)

Represents a collection of objects of the same element type. The components of a **SPACE** variable can be of different lengths.

Syntax

```
── SPACE ── [── constant_expr ── ]── OF ── type ─┤
```

Parameters

constant_expr	is the size of the storage area (in bytes) of the type.
type	is any type except **FILE** or **TEXT**.

Description

A variable of the type **SPACE** occupies the number of bytes indicated in the length specifier of the type definition and is byte-aligned.

Unlike an array, where an element is accessed by an index value, an element of a **SPACE** variable is accessed with an **INTEGER** expression that represents the byte offset of the element within the **SPACE** storage area. The offset is specified with an origin of zero.

You can pass an element of a **SPACE** variable by **CONST** or **VALUE**.

Operations

The following predefined functions are valid for the **SPACE** data type.

- **ADDR**
- **HBOUND**
- **LBOUND**
- **SIZEOF**

Related Information

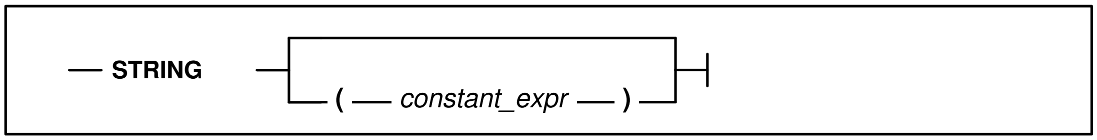
Refer to page 95 for examples of the **SPACE** data type.

STRING (VS Mode)

Purpose

The predefined data type **STRING** is defined as a 2-byte length field plus a **PACKED ARRAY** [1..*n*] **OF CHAR** whose length varies at run time up to a compile-time specified maximum given by the constant expression. The value of the constant expression defining the length of the string must be in the range 0..32767. The default maximum length is 255.

Syntax



Parameter

constant_expr is any integer constant expression.

Description

The current length of the array is obtained during run time by the **LENGTH** function. The length is managed implicitly by the operators and functions that apply to strings. The maximum length of the array is obtained during run time by the **MAXLENGTH** function. The length of a string variable is determined when the variable is assigned.

If the length of the source strings is greater than the maximum length of the target, the assignment of one string to another may cause a runtime error. XL Pascal does not automatically truncate strings unless given explicit directions to do so.

A string variable can be subscripted with an integer expression to refer to individual characters. A subscript of 1 refers to the first character. The subscript value must not be less than 1 nor exceed the length of the string.

Implicit conversion is performed when a string is assigned to a variable whose type is **PACKED ARRAY** [1..*n*] **OF CHAR**. All other conversions must be done explicitly.

Operations

The following operations and predefined routines apply to variables of type **STRING**.

- The operators =, <>, ~=, <, <=, >=, > , ||, +
- **ADDR**
- **COMPRESS**
- **DELETE**
- **HBOUND**
- **INDEX**
- **LBOUND**
- **LENGTH**
- **LPAD**
- **LTRIM**
- **MAXLENGTH**
- **PACK**
- **PICTURE**
- **READSTR**
- **RINDEX**
- **RPAD**
- **SIZEOF**
- **STOGSTR**
- **STR**
- **SUBSTR**
- **TRIM**
- **UNPACK**
- **WRITESTR**

Notes:

1. Both operands must be of type **STRING**.
2. If two strings being compared are of different lengths, the shorter is assumed to be padded with blanks on the right until the lengths match.
3. Relative magnitude of two strings is based upon the collating sequence of ASCII.

Applying Relational Operators to String Data

The following table shows how to apply relational operators to variables of type **CHAR** and **STRING** and to fixed strings:

Left Operand	**Right Operand**	**Result**
CHAR	CHAR	Allowed
	PACKED ARRAY OF CHAR	Not Allowed
	STRING	Use STR on the CHAR
PACKED ARRAY OF CHAR	CHAR	Not permitted
	PACKED ARRAY OF CHAR	Allowed if operands are type compatible
	STRING	Use STR on the array
STRING	CHAR	Use STR on the CHAR
	PACKED ARRAY OF CHAR	Use STR on the array
	STRING	Allowed

Converting Strings on Assignment

Target Variable	**Source Expression**	**Result**
CHAR	CHAR	Allowed
	PACKED ARRAY OF CHAR	Not permitted
	STRING	Use string indexing to obtain a CHAR
PACKED ARRAY OF CHAR	CHAR	Not permitted
	PACKED ARRAY OF CHAR	Allowed if operands are type compatible
	STRING	STRING is converted. If truncation is required, an error results.
STRING	CHAR	Use STR to convert the CHAR to a string
	PACKED ARRAY OF CHAR	Use STR to convert the array to a string
	STRING	Allowed

Mixed Strings

Strings with a mixture of single-byte characters and multibyte characters are called *mixed strings*. XL Pascal provides a set of routines that allow you to manipulate mixed strings. These routines recognize and preserve multibyte characters by counting them as a unit.

Note: Mixed string routines cannot have **GSTRING** operands.

The following routines apply to mixed strings:

- **MCOMPRESS**
- **MDELETE**
- **MINDEX**
- **MLENGTH**
- **MLTRIM**
- **MRINDEX**
- **MSUBSTR**
- **MTRIM**

Examples

```
FUNCTION getchar( CONST s : STRING; idx : INTEGER ) : CHAR;
   BEGIN
      getchar := s[idx]    (* Subscripted string variable *)
   END;
   ...
   VAR
      s1 : STRING( 10 );
      s2 : STRING( 5 );
      c : CHAR;
   BEGIN
      s1 := 'MESSAGE:';
      c := getchar( s1, 4 );  (* Returns 4th character in the *)
      ...                     (* string s1; c is assigned 'S' *)
      s2 := 'FIVE';
      c := getchar( s2, 2 );  (* Returns 2nd character in the *)
                              (* string s2; c is assigned 'I' *)
   END;
```

STRINGPTR (VS Mode)

Purpose

Defines a pointer to a dynamic string variable, which is a string with no maximum length associated with it until run time.

STRINGPTR is equivalent to:

```
TYPE
   STRINGPTR = @STRING;
```

The procedure **NEW** allocates storage for the **STRINGPTR** data type. An integer expression is passed to the procedure that specifies the maximum length of the allocated string.

Variables of type **STRING** have two lengths associated with them:

- The current length that defines the number of characters in the string at any instant.
- The maximum length that defines the storage required for the string.

Operations

The following operators and predefined routines apply to the **STRINGPTR** data type:

- Operators =, <>, ~=
- **ADDR**
- **DISPOSE**
- **DISPOSEHEAP**
- **MARK**
- **NEW**
- **NEWHEAP**
- **QUERYHEAP**
- **RELEASE**
- **SIZEOF**
- **USEHEAP**

Example

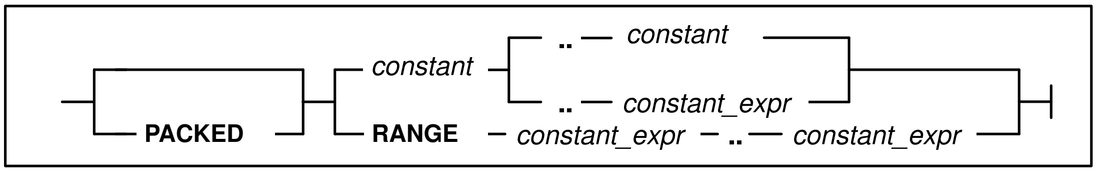
```
VAR
   p : STRINGPTR;
   q : STRINGPTR;
   i : 0..32767;

BEGIN
   ...
   i := 30;
   NEW( p, i );              (* allocates a string variable   *)
                             (* with maximum length 30, and   *)
                             (* sets pointer p to point at it *)
   p@ := 'abc';
   WRITELN( MAXLENGTH( p@ ) );     (* writes '30' to output   *)
   WRITELN( LENGTH( p@ ) );        (* writes '3' to output    *)
   NEW( q, 5 );
   q@ := '1234567890';(* causes a truncation error at run time*)

END;
```

Subrange Scalar

Purpose

Is a subset of consecutive values of a previously defined ordinal scalar. Any operation permitted on a scalar is also permitted on any subrange of it.

Syntax



Parameters

constant	is any ordinal constant.
constant_expr	is any ordinal constant expression (VS mode only).

Description

A subrange is defined by specifying the minimum and maximum values permitted for data declared with that type. The lower bound of a subrange type must not be greater than the upper bound, and both bounds must be of identical scalar types.

Subranges in VS Mode

For packed subranges in VS mode, XL Pascal assigns the smallest number of bytes required to represent a value of that type.

The VS mode reserved word **RANGE** allows you to use a constant expression for the minimum value. If the reserved word **RANGE** appears in the subrange definition, both the minimum and maximum values can be any expression that can be computed at compile time. If you do not use the reserved word **RANGE**, the minimum value of the range must be a simple constant, while the maximum value can still be any expression that can be computed at compile time.

Operations

The following predefined routines operate on subrange expressions.

Standard Mode:

- **ORD**
- **PRED**
- **SUCC**

VS Mode:

- **HIGHEST**
- **LOWEST**
- **MAX**
- **MIN**

Restrictions

Standard Mode:

- A subrange of type **REAL** is not permitted.
- The number of values in a subrange of type **CHAR** is determined by the collating sequence of the ASCII character set.
- **PACKED** subranges are not allowed.

VS Mode:

- Subranges of type **SHORTREAL** or **GCHAR** are not permitted.
- The lower bound of a subrange definition not prefixed with **RANGE** must be a simple constant instead of a generalized constant expression.

Examples

In the following VS mode example, `some_upper_case`, `one_hundred`, `codes`, and `index` are subrange scalar types. All of the **VAR** declarations define subrange scalar variables.

```
CONST
   size = 1000;

TYPE
   days = ( su, mo, tu, we, th, fr, sa );
   months = ( jan, feb, mar, apr, may, jun,
              jul, aug, sep, oct, nov, dec );
   some_upper_case = ’A’..’I’;
   one_hundred = 0..99;
   codes = RANGE CHR(0)..CHR( 255 );
   index = PACKED 1..size + 1;

VAR
   work_day : mo..fr;
   summer : jun..aug;
   smallint : PACKED 0..255;
   year : 1900..2000;
```

The following example illustrates two subrange types defined over the same base type. Operations are permitted between these two variables because they have the same base type.

```
VAR
   neg : MININT..-1;
   pos : 1..MAXINT;
```

Related Information

A table showing the full ASCII character set is in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

TEXT

Purpose

The predefined data type **TEXT** is a file of character data divided into lines by the ASCII new line character (`X'0A'`).

XL Pascal predefines two **TEXT** variables named **INPUT** and **OUTPUT**. The default for **INPUT** and **OUTPUT** is terminal I/O. XL Pascal in standard mode requires that the files **INPUT** and **OUTPUT** be listed in the program header if you use them in the program. VS mode defines **STDERR** as an additional predefined variable of the type **TEXT**. Because these files are predefined, you do not need to explicitly declare them in your program.

Operations

The following predefined routines apply to the **TEXT** data type:

Standard Mode:

- **EOF**
- **EOLN**
- **GET**
- **PAGE**
- **PUT**
- **READ**
- **READLN**
- **RESET**
- **REWRITE**
- **WRITE**
- **WRITELN**

VS Mode:

- **ADDR**
- **CLOSE**
- **COLS**
- **SIZEOF**
- **TERMIN**
- **TERMOUT**

Chapter 7. Variables

XL Pascal divides variables into five classes, depending on how they are declared:

- Automatic (**VAR** variables)
- Dynamic (**POINTER** variables)
- Parameter (declared in a routine header)
- Static (**STATIC** variables)
- External **(DEF**/**REF** variables)

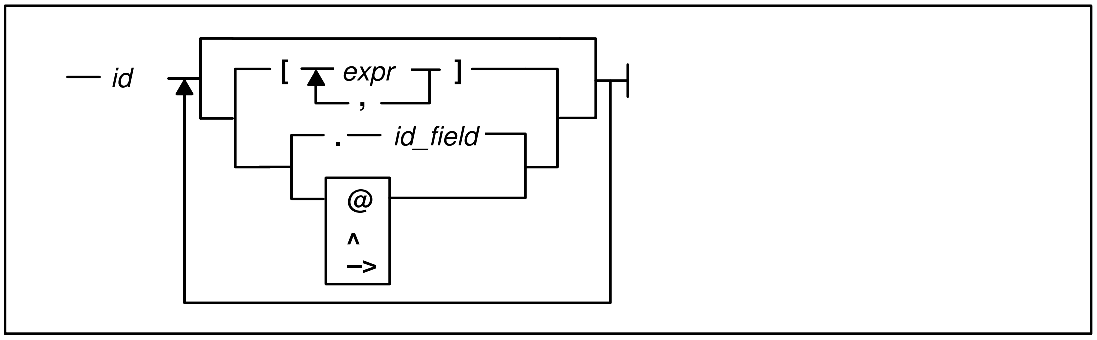
A variable has a type and a storage area in memory. Initially, it has no value associated with it. At any given time after acquiring a value, a variable takes on one value out of the collection of values that define its type.

You must declare all variables in an XL Pascal program before they are used. To avoid unpredictable results, you should explicitly initialize all variables before they are used.

Variable References

Depending on its type, you can refer to a variable in several ways. You can reference the entire variable specifying its name, and you can refer to a dynamic variable or a component of a structured variable by using the following syntax.

Syntax



Parameters

id	is the name of the variable
@ or ^	is a pointer or file reference
–>	is a pointer or file reference (VS mode only)
expr	is a subscripted variable reference (index expression)
id_field	is a field reference

Example

```
PROGRAM test ;

TYPE
   mytype = @INTEGER ;
   info = RECORD
             current : ARRAY [ 1..5 ] OF mytype ;
             other : INTEGER ;
          END ;

VAR
   abc : INTEGER ;
   storage : ARRAY [ 1..3, 2..5 ] OF info ;
BEGIN
   .
   .
   abc := storage[ 1, 2 ].current[ 1 ]@ ; (*variable reference*)
   .
   .
END.
```

Predefined Variables

XL Pascal provides three predefined variables:

INPUT	Default input file, **TEXT** data type
OUTPUT	Default output file, **TEXT** data type
STDERR	Default standard error file, **TEXT** data type (VS mode only)

Lifetime of Variables and Parameters

Standard Mode

Local Variables
The lifetime of a local variable is the same as that of the block in which it is declared. Allocation occurs on each entry to that block, and local variables are deallocated on each exit from that block.

Global Variables
The lifetime of global variables is that of the entire program. Variables outside of procedures or functions in a program or a segment unit are considered global.

Dynamic Variables
The allocation operation **NEW** establishes, but does not initialize, dynamic variables. Dynamic variables become undefined when they are explicitly deallocated by the **DISPOSE** or **RELEASE** procedures, or when no pointer variable points to them.

Formal Parameters
The lifetime of a formal parameter is the same as that of the procedure or function containing it. A formal parameter is established with each entry to the procedure or function and becomes undefined upon exit from it.

VS Mode

Static Variables
These are allocated before you run a program, and they exist for the life of the program run. Separate invocations of a routine preserve data in static variables local to that routine. Use **VALUE** declarations to initialize static variables at compile time.

External Variables
These are allocated before running a program and exist for the life of a program. Unlike static variables, they are accessible to several separately compiled routines (that is, throughout the program). Use **VALUE** declarations to initialize external variables at compile time.

Related Information

For more information about the **VALUE** declaration, refer to page 36.

Subscripted Variables

Array

Select an element of an array by placing an indexing expression enclosed within square brackets after the name of that array. The indexing expression must be assignment compatible with the type declared on the corresponding array index definition.

A multidimensional array can be referenced as an array of arrays. For example, let variable `A` be declared as follows:

```
A: ARRAY [a..b, c..d] OF T
```

As explained in "ARRAY Data Type" on page 57, this declaration is equivalent to:

```
A: ARRAY [a..b] OF
   ARRAY [c..d] OF T
```

A reference of the form `A[i]` represents a single row in array `A`, and is a variable of type:

```
ARRAY [c..d] OF T
```

A reference of the form `A[i][j]` is a variable of type `T`, and represents the jth element of the ith row of array `A`. This latter reference is customarily abbreviated as:

```
A[i,j]
```

You can abbreviate any array reference with two or more subscript indexes by writing the subscripts in order, separated by commas. That is, you can write `A[i][j]...` as:

```
A[i,j,...]
```

Example

```
TYPE
   matrix = ARRAY [1..10, 1..10] OF REAL;
   matrix0 = ARRAY [1..10] OF  (* An alternative declaration *)
                ARRAY [1..10] OF REAL; (* for matrix, above  *)
   color = ( red, yellow, blue ) ;
   intensity = PACKED ARRAY [color] OF REAL;

VAR
   m : matrix;
   hue : intensity;

BEGIN
   (* assign ten element array *)
   m[1] := m[2];
   (* assign one element of a two dimensional array two ways *)
   m[1,1] := 3.14159;
   m[1] [1] := 3.14159;
   (* this is a reddish orange *)
   hue [red] := 0.7;
   hue [yellow] := 0.7;
   hue [blue] := 0.7;
END;
```

STRING and GSTRING

In VS mode, variables of type **STRING** or **GSTRING** can be subscripted with integer expressions to reference individual characters. The value of the subscript must not be less than 1 or greater than the length of the string. Subscripting a **STRING** returns a **CHAR**; subscripting a **GSTRING** returns a **GCHAR**.

Error Checking

If the **%CHECK SUBSCRIPT** option is turned on, the index expression is checked at run time to make sure its value lies within the subscript range of the array. A runtime error diagnostic occurs if the value lies outside of the prescribed range.

Field References

To select a field of a record, write the record variable name followed by a period, and then the name of the field.

Examples

```
VAR
   i : INTEGER;
   person : RECORD
               first_name, last_name : STRING( 15 );
            END;
   date : RECORD
             day : 1..31;
             month : 1..12;
             year : 1900..2000
          END;
   deck : ARRAY [1..52] OF
             RECORD
                card : 1..13;
                suit : ( spade, heart, diamond, club )
             END;
   .
   .

BEGIN
   i := 1;
   person.first_name := 'JOE';
   person.last_name  := 'SMITH';
   date.year         := 1993;
   deck[ i ].card    := 2;
   deck[ i ].suit    := spade;
END;
```

Pointer References

You create a dynamic variable with the predefined procedure **NEW**. You can refer either to the pointer or to the dynamic variable. To refer to the dynamic variable, you must use the pointer notation. This is also called *dereferencing* the pointer.

For example, in the following declaration, a reference to `p` refers to the pointer, and a reference to `p@` refers to the dynamic variable to which `p` points:

```
VAR
   p : @ r;
```

Note: The pointer reference `p@` cannot refer to a pointer-valued function.

Any attempt to refer to a dynamic variable using a pointer with the value **NIL** when **%CHECK POINTER** is specified results in a runtime error message.

Example

```
TYPE
   info = RECORD
             age : 1..99;
             weight : 1..400;
          END;
   family = RECORD
               father, mother, self : @info;
               kids : 0..20
            END;

VAR
   family_pointer : @family;
   .
   .
   NEW( family_pointer );
   family_pointer@.kids := 2;
   NEW( family_pointer@.father );
   family_pointer@.father@.age := 35;
   .
   .
```

Related Information

For more information on pointer-valued functions, refer to page 140.

File References

Use pointer notation to select a component of a file from the file buffer. The file variable is established by using the predefined procedures **GET** and **PUT**. Each call to these procedures moves the current component to the output file (**PUT**) or assigns a new component from the input file (**GET**).

Example

```
VAR
   INPUT : TEXT;
   OUTPUT : TEXT;
   line1 : ARRAY [1..80] OF CHAR;
   i : INTEGER;
   .
   .

(* scan off blanks from a file of CHAR *)
GET( INPUT );
WHILE INPUT@ = ' ' DO
   GET( INPUT );
(* transfer a line to the OUTPUT file *)
FOR i := 1 TO 80 DO
   BEGIN
      OUTPUT@ := line1[i];
      PUT( OUTPUT )
   END;
```

Related Information

The **GET** Procedure is described on page 154. The **PUT** procedure is described on page 176.

SPACE References

Just as in array references, you select a component of a **SPACE** by placing an index expression, enclosed within square brackets, after the **SPACE** variable.

The indexing expression must be of type **INTEGER** or a subrange of **INTEGER**. The value of the index is the offset within the **SPACE** at which the component is to be accessed. The unit of the index is the byte. The index is always based upon a zero origin; the index range of the **SPACE** is from zero to one less than the value of the size of the **SPACE**.

If the **%CHECK SUBSCRIPT** option is enabled, XL Pascal checks the index expression at run time to make sure that the computed address does lies within the storage occupied by the space. If the value is not valid, a runtime error occurs, and a diagnostic message is issued.

Examples

```
VAR
   s : SPACE[100] OF RECORD
          (* declare a SPACE variable with index range 0..99 *)
          a, b : INTEGER
       END;

BEGIN
   (* base record begins at offset 10 WITHIN SPACE    *)
   s[10].a := 26;
   s[10].b := 0;
END;
```

No check is made for values that extend past the end of a **SPACE** unless the **%CHECK SUBSCRIPT** option is turned on. If the option is specified, the index expression is checked at run time to make sure its value lies within the subscript range of the array. The results are unpredictable if you try to make such an assignment or reference, as shown in the following example.

```
VAR
   s : SPACE[100] OF INTEGER;
   i : INTEGER;

BEGIN
   s[98] := i; (* not valid - extends past end of space *)
END;
```

Related Information

For a description of the **%CHECK** compiler directive, refer to page 205.

String References

You can refer to string variables as single entities when the entire string is being operated on. Single characters from a string can be referenced like a **PACKED ARRAY OF CHAR**. Either of the following allows you to assign values to string variables:

- Assignment statements
- Predefined string functions

String indexing starts from 1.

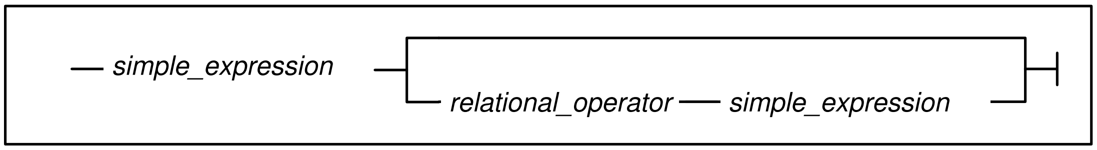
Always use an index value less than or equal to the current string length. It is an error to reference a string with an index greater than the length of the string.

Chapter 8. Expressions

An *expression*, when evaluated, returns a value. XL Pascal expressions are similar in function and form to expressions in other high-level programming languages. Expressions permit you to combine and manipulate data according to specific computational rules.

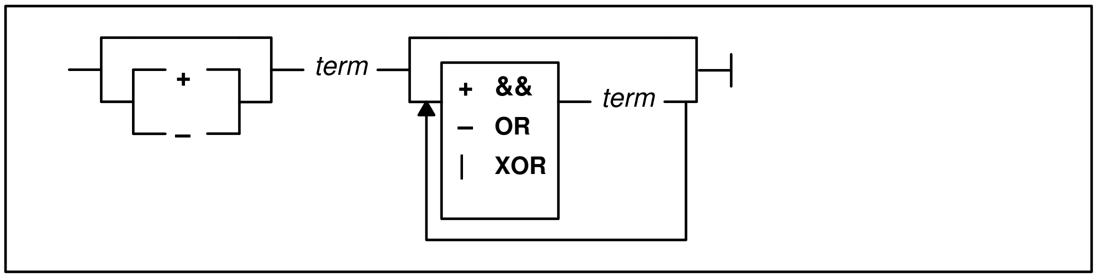
Syntax

Constant Expression or Expression

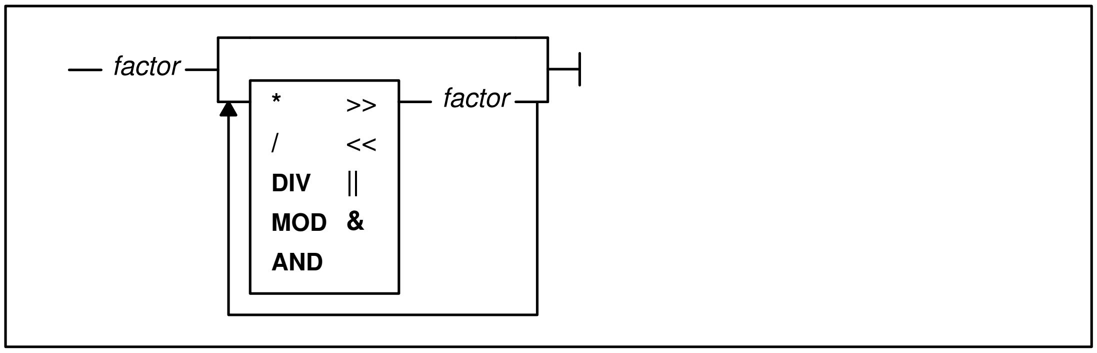


A *relational_operator* is one of the following : =, >=, <>, >, <, <=, **IN**, or ~= (VS Mode only).

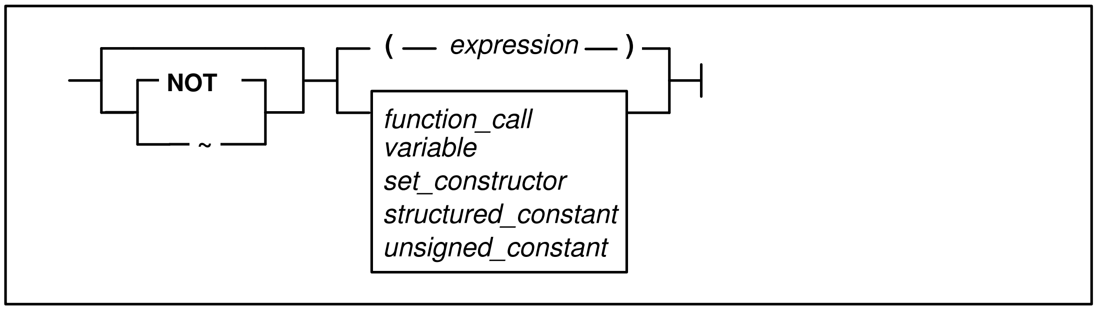
Simple expression



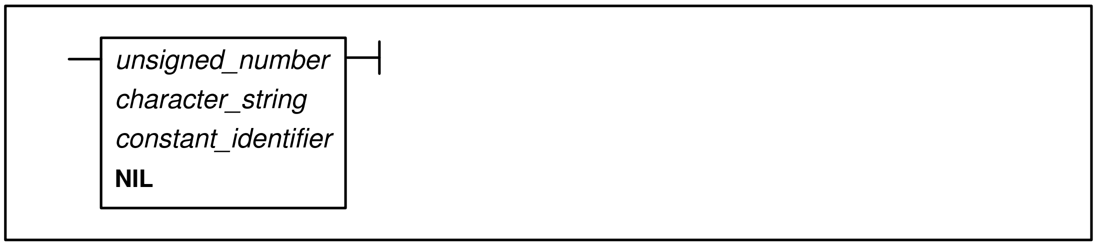
Term



Factor



Unsigned Constant



Examples

```
CONST
   acme = 'ACME';
TYPE
   color = ( red, yellow, blue );
   shade = SET OF color;
   days = ( sun, mon, tues, wed, thur, fri, sat );
   months = ( jan, feb, mar, apr, may, jun,
              jul, aug, sep, oct, nov, dec );
VAR
   a_color : color;
   a_set   : shade;
   bool    : BOOLEAN;
   month   : months;
   i, j    : INTEGER;
```

The following are examples of factors, terms, and expressions derived from the above example:

Factors:	**Description:**
`i`	`variable`
`15`	`unsigned constant`
`(i*8+j)`	`parenthesized expression`
`[ red ]`	`set of one element`
`[ ]`	`empty set`
`ODD(i*j)`	`function call`
`NOT bool`	`complement expression`
`acme`	`constant reference`
`color( 1 )`	`scalar type converter`

Terms:	**Description:**
`i`	`factor`
`i * j`	`multiplication`
`i DIV j`	`integer division`
`a_set * [ red ]`	`set intersection`
`bool & ODD(I)`	`Boolean AND`
`acme || ' TRUCKING'`	`concatenation`
`i & 'FF00'X`	`logical AND on integers`

Simple expressions:	**Description:**
`i * j`	`term`
`i + j`	`addition`
`a_set + [ blue ]`	`set union`
`- i`	`unary minus on an integer`
`i | '80000000'X`	`logical OR on integers`

Expressions:	**Description:**
`i + j`	`simple expression`
`red = a_color`	`relational operations`
`red IN a_set`	`test for set inclusion`

Evaluating Expressions

The type of calculation to be performed in an expression is directed by operators grouped into four classes according to the following precedence:

1. The **NOT** operator (highest)
2. The multiplication operators
3. The addition operators
4. The relational operators (lowest)

An expression is evaluated by applying the operators of highest precedence first, operators of the next precedence second, and so forth. Operators of equal precedence are applied in left-to-right order. If an operator has an operand that is a subexpression in parentheses, the subexpression is evaluated before applying the operator.

The operands of an expression may be evaluated in either order; that is, the left operand of a binary operator may be evaluated after the right operand. If either operand changes a global variable through a function call, and if the other operand uses that value, the value used is not necessarily the updated value. The only exception is in Boolean expressions involving the logical operations of **AND** (&) and **OR** (|); for these operations, the right operand is not evaluated if the result can be determined from the left operand.

Because simple expressions are prefaced with arithmetic signs, confusion can arise when using signs on the operands. The following example shows correct and incorrect use of signs in simple expressions:

```
CONST
   c = -7;
   x = c MOD 4;    (* yields 1                        *)
   y = -7 MOD 4;   (* yields -3 because it is treated  *)
                   (* as -(7 MOD 4)                    *)
   z = 7 DIV -4    (* error: -4 must be in parentheses *)
```

Out-of-Range Values

Avoid arithmetic expressions that yield results outside of the defined range of values for a given type. Compiling and running a program containing such expressions can give unpredictable results.

Operators

The *operators* are the logical or algebraic processes, such as addition or multiplication, that can be performed on a value or pair of values to produce a new value. The *operands* are the values manipulated by the operators to give a *result*. The following sections summarize the four categories of operators used in XL Pascal.

The following tables describe the types of operand that you can use with these operators and the result types for each operation.

NOT Operator

The **NOT** operator applies to operands of the type **BOOLEAN**, **INTEGER**, or **SET**. When applied to type **BOOLEAN**, it means negation, for example, `NOT TRUE = FALSE`. When applied to type **INTEGER**, the **NOT** operator negates all the bits in the operand. This is called *one's-complement negation*. When applied to type **SET**, the **NOT** operator returns the complement of the set. For example, if set `x` is `SET of 1..4` and has the value `[1]`, the complement of set `x` is `[2,3,4]`.

Standard Mode

Operator	**Operation**	**Operand**	**Result**
NOT	Boolean NOT	BOOLEAN	BOOLEAN

VS Mode

Operator	**Operation**	**Operand**	**Result**
NOT (~)	Boolean NOT	BOOLEAN	BOOLEAN
	Logical one's complement	INTEGER	INTEGER
	Set complement	SET of *t*	SET of *t*

Multiplication Operators

The multiplication operators have the next highest precedence after the **NOT** operator.

Standard Mode

Operator	**Operation**	**Operand**	**Result**
*	Multiplication	INTEGER	INTEGER
		REAL, INTEGER	REAL
	Set Intersection	SET of *t*	SET of *t*
/	Real Division	REAL, INTEGER	REAL
DIV	Integer Division	INTEGER	INTEGER
MOD	Modulus	INTEGER	INTEGER
AND	Boolean AND	BOOLEAN	BOOLEAN

VS Mode

Operator	**Operation**	**Operand**	**Result**
*	Multiplication	SHORTREAL	SHORTREAL
		SHORTREAL, INTEGER	SHORTREAL
		SHORTREAL, REAL	REAL
||	String Concatenation	STRING	STRING
		GSTRING	GSTRING
AND (&)	Boolean AND	BOOLEAN	BOOLEAN
	Logical AND	INTEGER	INTEGER
<<	Left Logical Shift	INTEGER	INTEGER
>>	Right Logical Shift	INTEGER	INTEGER

The **DIV** operator represents truncating division. **DIV** always truncates toward zero. The right operand cannot be zero. **DIV** is defined as

```
a DIV b = TRUNC ( a / b ), b ~= 0
```

Operands with the same sign yield a positive result; those with different signs return a negative value.

The **MOD** operator defines the modulus operation between two integer values. The right operand of **MOD** must be positive. The **MOD** operator is interpreted this way:

```
a MOD b = a − ( a DIV b ) * b, a >= 0, b > 0
a MOD b = b − ABS ( a ) MOD b, a < 0, b > 0
```

Addition Operators

The addition operators have the next highest precedence after the multiplication operators.

Standard Mode

Operator	**Operation**	**Operand**	**Result**
+	Addition	INTEGER	INTEGER
		REAL, INTEGER	REAL
	Set Union	SET of *t*	SET of *t*
−	Subtraction	INTEGER	INTEGER
		REAL, INTEGER	REAL
	Set Difference	SET of *t*	SET of *t*
OR	Boolean OR	BOOLEAN	BOOLEAN

VS Mode

Operator	**Operation**	**Operand**	**Result**
+	Addition	SHORTREAL	SHORTREAL
		SHORTREAL, INTEGER	SHORTREAL
		SHORTREAL, REAL	REAL
	String	STRING	STRING
	Concatenation	GSTRING	GSTRING
−	Subtraction	SHORTREAL	SHORTREAL
		SHORTREAL, INTEGER	SHORTREAL
		SHORTREAL, REAL	REAL
OR (|)	Boolean OR	BOOLEAN	BOOLEAN
	Logical OR	INTEGER	INTEGER
XOR (&&)	Logical XOR	INTEGER	INTEGER
	Boolean XOR	BOOLEAN	BOOLEAN
	Set exclusive union	SET of *t*	SET of *t*

Relational Operators

All scalar type operands for relational operations define ordered sets of values.

Standard Mode

Operator	Operation	Operand	Result
=	Compare equal	Any set, scalar type, pointer, or character	BOOLEAN
<>	Compare not equal	Any set, scalar type, pointer, or character	BOOLEAN
<	Compare less than	Scalar type or character type	BOOLEAN
<=	Compare less than or equal to	Scalar type or character type	BOOLEAN
	Subset	SET of *t*	BOOLEAN
>	Compare greater than	Scalar type or character type	BOOLEAN
>=	Compare greater than or equal	Scalar type or character type	BOOLEAN
	Superset	SET of *t*	BOOLEAN
IN	Set membership	*t* and SET of *t*	BOOLEAN

VS Mode

Operator	Operation	Operand	Result
~=	Compare not equal	Any set, scalar type, pointer, or character	BOOLEAN

Comparison According to Type

The following types can be compared with each other with predictable results:

- Scalar types
- Pointer types
- Strings
- Sets

Comparison of Scalars:

Scalar operands are ordered according to type. For numeric operands, the ordering is defined as follows:

1. **INTEGER**
2. **SHORTREAL**
3. **REAL**

For operands of different numeric types, the lower type of operand is converted to the level of the higher operand type before the two are compared. For example, in the following expression, the **SHORTREAL** is converted to **REAL** before the comparison is made:

```
SHORTREAL value < REAL value
```

For operands of type **BOOLEAN**, the ordering is defined as:

```
FALSE < TRUE
```

The ASCII collating sequence determines the ordering of operands of type **CHAR** and **GCHAR**.

The ordering of enumerated types depends on the order in which the values were specified in the type definition.

Direct Pointer Comparison:

You can compare two pointers if they are pointers to identical types. To compare pointers of different types, find their **ORD** value as described in "ORD Function" on page 170.

Compare pointers for equality or inequality only. Two pointers with the value **NIL** are always equal.

Character Comparison:

The relational operators compare both **PACKED ARRAY** [1..*n*] **OF CHAR** and **STRING** values. For a **PACKED ARRAY** [1..*n*] **OF CHAR** both operands must be the same size. The compiler does not check for the size difference.

In VS mode, you can compare **STRING** type operands of different lengths or **GSTRING** type operands of different lengths. In either situation, trailing blanks are not significant. You cannot compare a **GSTRING** operand with a **STRING** operand. The collating sequence of the ASCII character set determines the alphabetical ordering of **STRING** type operands or **PACKED ARRAY** [1..*n*] **OF CHAR** operands.

The type of a string literal is converted to the type of the other operand when you compare a string literal with a **STRING** operand or with a **PACKED ARRAY** [1..*n*] **OF CHAR** operand. Similarly, the type of a multibyte character literal is converted to the type of the other operand when you compare such a literal with a **GSTRING** operand or with a **PACKED ARRAY** [1..*n*] **OF GCHAR** operand.

Set Comparison:

The **IN** operator determines if a scalar value is a member of set. The base type of the set must be the same as the base type of the scalar.

The following operations are defined between two set values of the same base type. For two sets, *S1* and *S2*, and element *e1*:

Operation	Result
S1 = *S2*	True if all members of *S1* are contained in *S2*, and all members of *S2* are contained in *S1*
S1 <> *S2*	True when *S1*= *S2* is false
S1 <= *S2*	True if all members of S1 are also members of *S2*
S1 >= *S2*	True if all members of *S2* are also members of *S1*
e1 **IN** *S2*	True if *e1* is a member of *S2*

Noncomparable Types:

You cannot compare the following XL Pascal types:

- **ARRAY**
- **FILE**
- **RECORD**

Note: You can compare **PACKED ARRAY** [1..*n*] **OF CHAR** operands of the same size because they can be compared as strings.

Related Information

A table showing the full ASCII character set is in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

Constant Expressions

Constant expressions are expressions that can be evaluated by the compiler and replaced with a result at compile time. By definition, a constant expression cannot contain a reference to a variable, or to a user-defined function.

The following predefined functions are permitted in constant expressions.

Standard Mode:

- **ABS**
- **CHR**
- **ODD**
- **ORD**
- **PRED**
- **ROUND**
- **SQR**
- **SUCC**
- **TRUNC**

VS Mode:

- **FLOAT**
- **HBOUND**
- **HIGHEST**
- **LBOUND**
- **LENGTH**
- **LOWEST**
- **MAX**
- **MAXLENGTH**
- **MIN**
- Ordinal conversion
- **SIZEOF**
- **STR**

In VS mode, constant expressions can appear in constant declarations, record variant tag lists, and **CASE** constant lists.

Examples

Constant Expression	Type of Value
`SUCC(CHR('F0'X))`	**CHAR**
`256 DIV 2`	**INTEGER**
`['0'..'9']`	**SET OF CHAR**
`32768*2-1`	**INTEGER**
`ORD('A')`	**INTEGER**
`'TOKEN'||STR(CHR(0))`	**STRING**
`'8000'X |'0001'X`	**INTEGER**

Related Information

See Chapter 10, “Routines,” for descriptions of the predefined functions.

Boolean Expressions

Pascal assigns the logical operators a higher priority than that of the relational operators. The expression

```
a<b & c<d
```

is evaluated as

```
(a < (b&c)) < d.
```

Thus, to ensure that your expressions are evaluated as you intend, use parentheses when writing expressions of this sort.

XL Pascal makes the evaluation of Boolean expressions involving **AND** (&) and **OR** (|) more efficient so that the right operand of the expression is not evaluated if the result of the operation can be determined by evaluating the left operand. For example, given that `a`, `b`, and `c` are Boolean expressions and `x` is a Boolean variable, the evaluation of

```
x := a or b or c
```

is performed as

```
IF a THEN
   x := TRUE
ELSE
   IF b THEN
      x := TRUE
   ELSE
      x := c
```

The evaluation of

```
x := a and b and c
```

is performed as

```
IF NOT a THEN
   x := FALSE
ELSE
   IF NOT b THEN
      x := FALSE
   ELSE
      x := c
```

This type of evaluation is called *short-circuiting*. The evaluation of the expression is always from left to right. Because some Pascal compilers do not support this interpretation of Boolean expressions, you should write the compound tests in a form that uses nested IF statements to assure portability between XL Pascal and other Pascal implementations.

You should not rely on short-circuiting because your program might not work as you intend. For example, if you use a function in the right operand of a Boolean expression, the function might not be evaluated. To avoid unexpected results, your programs should not permit functions to modify global variables or to contain **VAR** parameters.

Example

The following example demonstrates the logic that depends on the conditional evaluation of the right operand of the **AND** operator. If both operands in the **WHILE** statement were always evaluated, a **NIL** pointer checking error would occur when `p` had the value of **NIL**.

```
TYPE
   recptr = @rec;
   rec = RECORD
            name : ALPHA;
            next : recptr;
         END;

VAR
   p : recptr;
   lname : ALPHA;

BEGIN
   ...
   WHILE (p <> NIL) AND (p@.name <> lname) DO
      p := p@.next;
   END;
   ...
END;
```

Set Expressions

The following operations can be performed on the **SET** data type:

- Set complement produces a set that has all of the elements not in the set undergoing the operation.
- Set union produces a set containing all of the elements that are members of the two operands.
- Set intersection produces a set containing only the elements common to both sets.
- Set difference produces a set that includes all elements from the left operand except those elements in the right operand.
- Set exclusive union produces a set containing all elements from the two operands except those elements common to both operands.
- The **IN** operator tests for membership of a scalar within a set. If the scalar is not a value of the set, **FALSE** is returned.

Logical Expressions

Many of the logical operators provided in XL Pascal perform logical operations on integer operands; the operands are considered as unsigned strings of binary digits instead of signed arithmetic quantities. For example, if the integer value of –1 was used as an operand of a logical operation, it would be seen as a string of binary digits with a hexadecimal value of 'FFFFFFFF'X.

The logical operations are defined to yield 32-bit values. Such an operation on a subrange of an **INTEGER** could yield a result outside the subrange.

The following operators perform logical operations on integer operands:

- & (**AND**) performs a bit-wise **AND** of two integers.
- | (**OR**) performs a bit-wise inclusive **OR**.
- && (**XOR**) performs a bit-wise exclusive **OR**.
- ~ (**NOT**) performs a one's complement negation of an integer.
- << shifts the left operand value left by the amount indicated in the right operand. Zeros are shifted in from the right.
- >> shifts the left operand value right by the amount indicated in the right operand. Zeros are shifted in from the left.

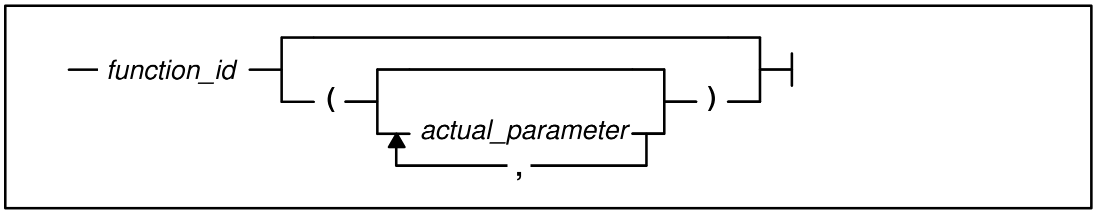
Examples

```
257 & 'FF'X          (* yields  1               *)
2 | 4 | 8            (* yields  14              *)
4 << 2               (* yields  16              *)
-4 << 1              (* yields  -8              *)
8 >> 1               (* yields  4               *)
-8 >> 1              (* yields  '7FFFFFFC'X     *)
'FFFF'X >> 3         (* yields  '1FFF'X         *)
~1 & 'FF'X           (* yields  'FE'X           *)
~0                   (* yields  -1              *)
'FF'X && 8           (* yields  'F7'X           *)
```

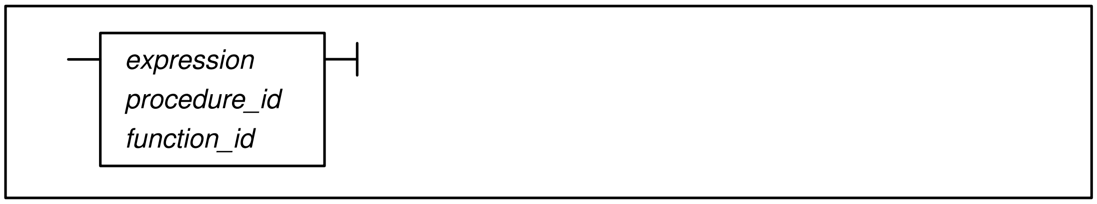
Function Calls

A function returns a value to the caller. A call to a function passes the actual parameters to the corresponding formal parameters.

Syntax



Actual Parameter



Description

Each actual parameter must be assignment compatible with the corresponding formal parameter.

Parameters passed by read/write reference (**VAR**) can only be variables, and never expressions or constants. In standard mode, actual and formal **VAR** parameters must be of the same type. Also in standard mode, you cannot pass fields of a packed record or elements of packed arrays by **VAR**. Parameters passed by value or read-only reference (**CONST**) can be any expression.

In VS Mode, if the function requires no parameters, you can omit the expression in parentheses. To draw attention to a function call with no parameters and make it appear different from a variable reference, follow the function name with an empty set of parentheses. In standard mode, however, the parentheses must be omitted.

Example

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

FUNCTION sum( m, n : INTEGER ) : INTEGER;
   BEGIN
      sum := m + n
   END;
   ...
   BEGIN
      ...
      c := sum( a, b ) * 2;
      ...
   END;
```

Related Information

Parameter compatibility rules are defined in “Routine Parameters” on page 136. The rules for expression compatibility are given in “Type Compatibility” on page 50. See Chapter 10, “Routines”, for information about defining functions.

Ordinal Conversions

In VS mode, the definition of any type identifier that specifies an ordinal type (scalars or subranges) forms an ordinal conversion function. Ordinal conversion functions convert an **INTEGER** into a specified ordinal type.

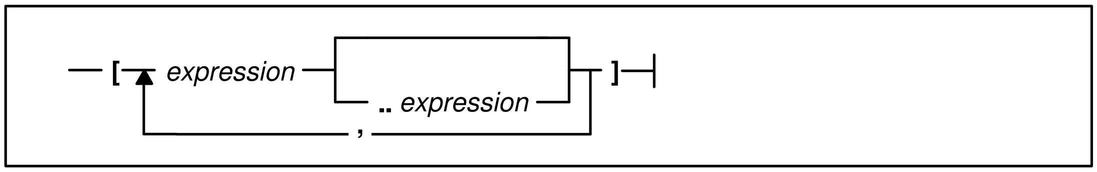
Related Information

Ordinal conversion functions are described on page 170.

Set Constructors

A set constructor computes the value of a **SET**.

Syntax



Description

The set constructor is either a list of expressions separated by commas or a group of expression pairs. Square brackets contain the expressions. An expression pair designates that all values from the first expression through the last expression are to be included in the resulting set. If the value of the first expression is greater than the value of the second, no values are designated.

All expressions must be type compatible, and this type becomes the base scalar type of the set. If the set specifies integer-valued expressions, the base scalar type of the set is 0..255.

Example

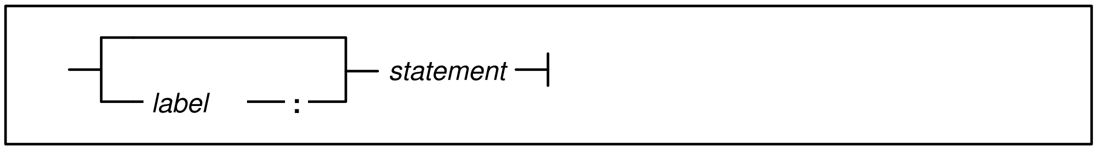
```
TYPE
   days = SET OF ( sun, mon, tue, wed, thu, fri, sat );
   charset = SET OF CHAR;

VAR
   workdays, weekend : days;
   nonletters : charset;
   ...

BEGIN
   workdays := [mon..fri];
   weekend := ~ workdays;
   nonletters := ~ ['a'..'z','A'..'Z'];
   ...
END;
```

Chapter 9. Statements

The *statements* denote algorithmic operations and define what actions a program is to perform on the program objects introduced by type and variable declarations. XL Pascal statements are similar to those found in most high-level programming languages.

This chapter describes statement labels and the statements of XL Pascal.

Syntax



Parameters

label	is an optional label.
statement	is any XL Pascal statement.

Statement Labels

You can label a statement with an unsigned integer constant in the range 0..9999 followed by a colon and the statement you want to label. Labels permit you to explicitly refer to a statement by a **GOTO** statement. On encountering a **GOTO** label sequence, the system processes the statement prefixed by a label as the next statement instead of the statement following the **GOTO**.

In VS mode only, you can also label a statement with an identifier.

Scope of Statement Labels

The scope of a statement label is the body of the routine in which the label is declared and all of its nested routines. The **GOTO** statement cannot transfer control into a routine unless that routine has been activated.

Summary of XL Pascal Statements

Standard Mode

Assignment	Assigns a value to a variable.
CASE	Permits a program to process one member of a list of possible statements based upon the evaluation of an expression.
Compound	Brackets a series of statements to be processed sequentially.
Empty	Serves as a place holder and has no effect on the running of the program.
FOR	Causes a program to repeatedly run a group of statements a specified number of times.
GOTO	Changes the flow of control within a program.

IF	Specifies that one of two statements be processed depending on the evaluation of a Boolean expression.
Procedure call	Calls a procedure.
REPEAT	Causes statements between the statement delimiters **REPEAT** and **UNTIL** to be processed until a control expression is **TRUE**.
WHILE	Processes a statement while a control expression remains **TRUE.**
WITH	Simplifies reference to a record variable by eliminating an addressing description on every reference to a field.

VS Mode

ASSERT	Checks for a condition and signals a runtime error if the condition is not met.
CONTINUE	Causes a jump to the loop-continuation portion of the innermost enclosing **FOR**, **WHILE**, or **REPEAT** statement.
LEAVE	Causes an immediate, unconditional exit from the innermost enclosing **FOR**, **WHILE**, or **REPEAT** loop.
RETURN	Permits an exit from a procedure or function.

ASSERT

Purpose

Checks for a specific condition and signals a runtime error if the condition is not met.

Syntax

```
── ASSERT ── expr ─┤
```

Parameters

ASSERT	is a reserved word
expr	is any Boolean expression

Description

The condition is specified by the expression, which must be evaluated to a Boolean value. If the condition is **FALSE**, a runtime diagnostic message is issued. The compiler may remove the statement from the object program if it can determine that the assertion is always **TRUE**.

Example

In the following example, if `a` is greater than or equal to `b`, the value is **TRUE** and no action is taken; otherwise, an error message is displayed.

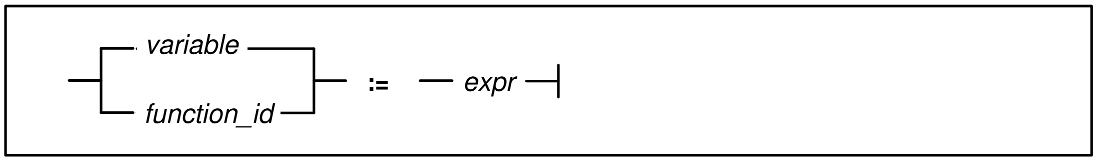
```
ASSERT a >= b ;
```

Assignment

Purpose

Replaces the current value of a variable with a new value derived from an expression evaluation. An assignment statement can also define the value a function variable returns.

Syntax



Parameters

variable	is any variable.
function_id	is the identifier for any enclosing user defined function. Note that *function_id* can only appear to the left of an assignment operator within the body of the function itself.
expr	is any expression assignment compatible with the *variable* or *function_id*, including arrays or records.

Restrictions:

- A variable of type **FILE**, **TEXT**, or of a type containing a file even indirectly cannot appear in an assignment statement.
- VS mode does not permit the assignment of a value to a pass-by-**CONST** parameter.

Assignments to Variables and Functions

The assignment statement consists of a reference to a variable followed by the assignment symbol (:=), followed by an expression that, when evaluated, is the new value. The current value of the referenced variable is then replaced by the value of the expression. The value must be assignment compatible with the variable.

You can assign a string constant to a variable of the type **PACKED ARRAY** [1..*n*] **OF CHAR**, provided that the string value is the same length as the array object.

When you make array assignments (assign one array to another array) or record assignments (assign one record to another), the entire array or record is assigned.

To return a result from a user-defined function, assign a value to the function name before leaving the function.

Example

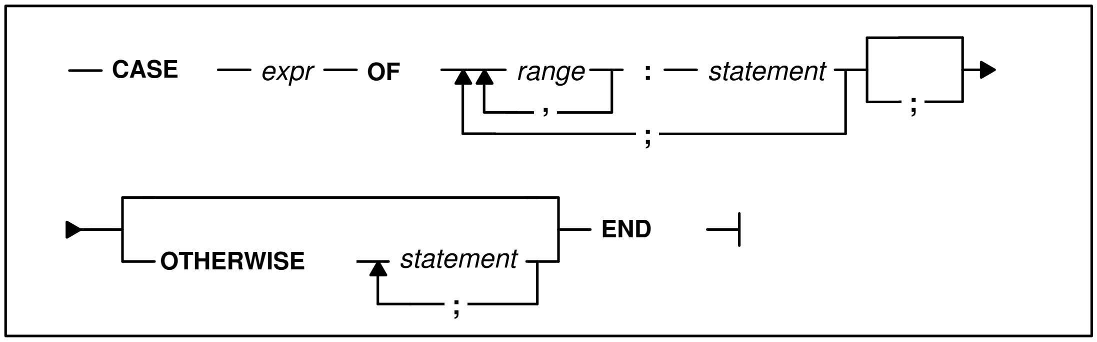
```
TYPE card = RECORD
               suit : ( spade, heart, diamond, club );
               rank : 1..13
            END;

VAR
   p , x , y , z : REAL;
   letters, digits, letter_or_digit : SET OF CHAR;
   i, j, k : INTEGER;
   deck : ARRAY [1..52] OF card;

FUNCTION square( a : REAL ) : REAL;
   BEGIN
      square := a * a;
   END;
   ...
   BEGIN
      i := 1 ;
      z := 0.016 ;
      x := y * z ;
      letters := ['A'..'Z'];
      digits := ['0'..'9'];
      letter_or_digit := letters + digits;
      deck[i].suit := heart;
      deck[j] := deck[k];
      p := square( 2.0 );
   END;
```

CASE

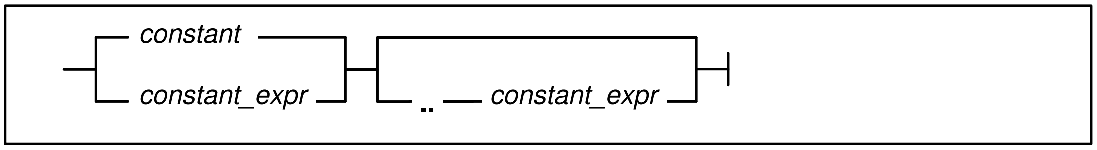
Purpose

Processes one member of a list of possible statements, based on the evaluation of an expression.

Syntax



Range:



Parameters

CASE	is a reserved word.
expr	is any expression that evaluates to an ordinal type.
OF	is a reserved word.
statement	is any statement.
END	is a reserved word.
constant	is any constant assignment compatible with *expr.*
OTHERWISE	is a reserved word (VS mode only).
constant-expr	is any constant expression whose type is compatible with *expr* (VS mode only).

Description

The statement consists of an expression called the *selector* and a list of statements. The selector must be an ordinal type, one of:

- **BOOLEAN**
- **CHAR**
- Enumerated
- **INTEGER**
- Subrange

Each statement is prefixed with one or more ranges of the same type as the selector. Each range is separated by a comma and designates one or more values called *case labels*. Case selectors and case labels must be of the same scalar type. XL Pascal evaluates the selector and processes the statement whose **CASE** range contains the **CASE** label equal to the value of the selector.

You can write the range values of a **CASE** statement in any order, but the **CASE** ranges cannot overlap. The same **CASE** label cannot appear more than once in a **CASE** statement. XL Pascal allows a maximum of 255 **CASE** labels for each **CASE** statement.

VS Mode

If no **CASE** label equals the value of the selector, the **OTHERWISE** statement is processed (if it is present). If no **CASE** label equals the value of the selector and there is no **OTHERWISE** statement, a runtime error results when the **%CHECK CASE** option is on. If the checking is not on, the results are unpredictable.

Examples

```
TYPE
   shape = ( triangle, rectangle, square, circle );
   coordinates = RECORD
                    x, y : REAL;
                    area : REAL;
                    CASE s : shape OF
                       (* variant part of record coordinates *)
                       triangle : ( side : REAL;
                                    base  : REAL );
                       rectangle : ( sidea , sideb : REAL );
                       square : ( edge : REAL );
                       circle :
                       ( radius : REAL )
                 END; (* of record coordinates *)

 VAR
   coord : coordinates;
   .
   .
 WITH coord DO
   CASE s OF
      triangle  : area := 0.5 * side * base;
      rectangle : area := sidea * sideb;
      square    : area := SQR( edge );
      circle    : area := 3.14159 * SQR( radius )
   END;
```

The following example shows a **CASE** statement with the **OTHERWISE** reserved word:

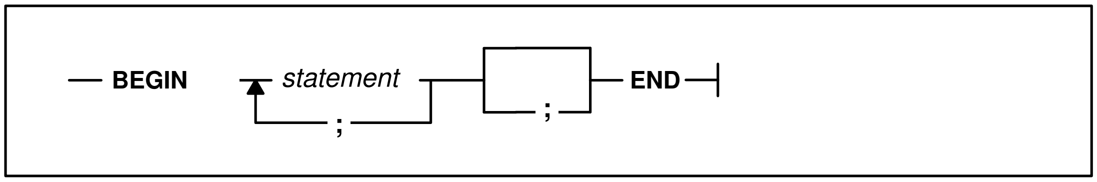
```
TYPE
   rank = ( ace, two, three, four, five, six, seven,
            eight, nine, ten, jack, queen, king );
   suit = ( spade, heart, diamond, club );
   card = RECORD
             r : rank;
             s : suit
          END;

 VAR
   points : INTEGER;
   a_card : card;
   . . .
   CASE a_card.r OF
      ace :  points := 11 ;
      two..ten : points := ORD( a_card.r ) + 1;
      OTHERWISE  points := 10
   END;
   . . .
```

Compound

Purpose

Brackets a series of statements to be processed sequentially as if they were a single statement.

Syntax



Parameters

BEGIN	is a reserved word
statement	is any statement
END	is a reserved word

Description

The reserved words **BEGIN** and **END** delimit the statement list. The statement list can be exited either by an explicit transfer of control to another program unit, or when the last statement in the list is processed.

Example

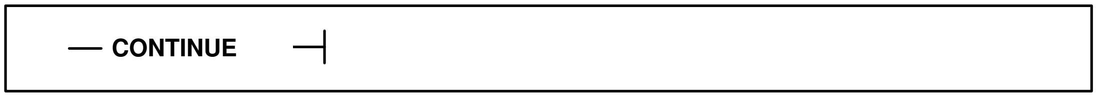
```
IF a > b THEN
   BEGIN      (* swap a and b *)
      temp := a;
      a := b;
      b := temp
   END;
```

CONTINUE

Purpose

Causes a jump to the loop-continuation portion of the innermost enclosing **FOR**, **WHILE**, or **REPEAT** statement, acting as a **GOTO** to the end of the loop body.

Syntax



Examples

The following examples illustrate how the **CONTINUE** statement functions in each of the loop constructs.

FOR Statements:

```
FOR i := expr1 TO expr2 DO
   BEGIN
      ...
      CONTINUE;
      ...
      (* continue jumps to here *)
   END;
```

WHILE Statements:

```
WHILE expr DO
   BEGIN
      ...
      CONTINUE;
      ...
      (* continue jumps to here *)
   END;
```

REPEAT Statements:

```
REPEAT
    ...
    CONTINUE;
    ...
    (* continue jumps to here *)
UNTIL expr;
```

The following example shows a **CONTINUE** statement and its equivalent:

```
WHILE expr DO
   BEGIN
      .
      .
      IF expr THEN
         CONTINUE;
      .
      .
   END;
```

is equivalent to

```
WHILE expr DO
   BEGIN
      .
      .
      IF expr THEN
         GOTO label;
      .
      .
      label:   (* continue jumps to here *)
   END;
```

Empty

Purpose

Is a place holder and has no effect on program processing.

Syntax

Description

The empty statement is often useful when you want to place a label in the program but do not want it attached to another statement (such as at the end of a compound statement). It is also useful in avoiding the ambiguity that arises in nested **IF** statements. You can force an **ELSE** clause to be paired with an outer nested **IF** statement by using an empty statement after an **ELSE** clause in the inner nested **IF** statement.

Example

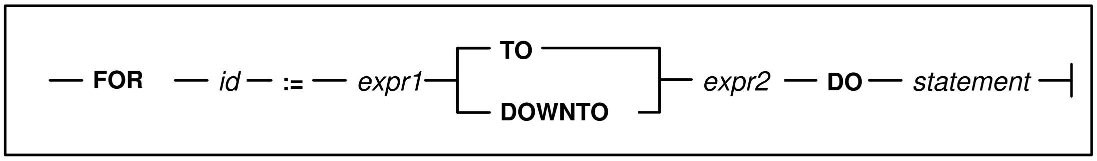
```
IF b1 THEN
   IF b2 THEN
      s1
   ELSE
     (* empty statement *)
ELSE
   s2
```

FOR

Purpose

Processes a group of statements a specified number of times.

Syntax



Parameters

FOR	is a reserved word.
id	is any local, automatic variable of any ordinal type.
expr1	is any valid expression which must be assignment compatible with *id*. Evaluated to an ordinal, this is the initial value of the control variable.
TO	calculates the value of the control variable by the **SUCC** function after processing the statement.

DOWNTO	calculates the value of the control variable by the **PRED** function after processing the statement.
expr2	is any valid expression which must be assignment compatible with *id*. Evaluated to an ordinal, this is the limiting value of the control variable.
DO	is a reserved word.
statement	is any valid statement.

Restrictions:

- The control variable must be an automatic ordinal scalar declared in the routine immediately enclosing.
- The control variable cannot be subscripted, field qualified, or referred to through a pointer.
- The processed statement must not change the control variable. If the control variable is changed within the loop, processing of the resultant loop is not predictable.
- In the statement in the **FOR** loop, and in any routine declared in the routine immediately enclosing the **FOR** loop, the control variable cannot be used:
  - In an assignment statement
  - As the control variable of another **FOR** statement
  - As an actual **VAR** parameter
  - As a parameter to an input routine like **READ** or **READLN**.

Description

The **FOR** loop begins with an identifier initialized to the first control expression. With each iteration of the loop, the value of the identifier is replaced by either its **SUCC** or **PRED** value, depending on the syntax of the statement.

You use the reserved word **TO** between the control expressions to increase the value of the loop control identifier. The new value of the identifier is computed automatically by the **SUCC** function before the statement is processed. Iteration continues as long as the value of the identifier is less than or equal to the value of the second control expression.

You use the reserved word **DOWNTO** between the control expressions to decrease the value of the loop control identifier. The new value of the identifier is computed automatically by the **PRED** function before the statement is processed. Iteration continues if the value of the identifier is greater than or equal to the value of the second control expression.

XL Pascal computes the value of the second expression at the beginning of the **FOR** statement and uses the result for the duration of the statement. Once the value of the second control expression is computed, it must not be changed during the **FOR** statement.

The value of the control variable after the **FOR** statement is processed is undefined on the normal termination of the **FOR** loop. Do not expect the control variable to contain any particular value. If a **GOTO** statement causes the program to exit from the **FOR** statement prematurely, the value of the control variable is defined.

Examples

In the following statement, `i` is an automatic scalar variable, `expr1` and `expr2` are scalar expressions type-compatible with `i`, and `statement` is any arbitrary statement:

```
FOR i := expr1 TO expr2 DO statement
```

The following compound statement is functionally equivalent to this **FOR** statement.

```
BEGIN
   temp1 := expr1;
   temp2 := expr2;
   IF temp1 <= temp2 THEN
      BEGIN
         i := temp1;
         statement;
         WHILE i <> temp2 DO
            BEGIN
               i := SUCC( i );
               statement
            END;
      END;
END;
```

In the following statement, `i` is an automatic scalar variable, `expr1` and `expr2` are scalar expressions type-compatible with `i`, and `statement` is any arbitrary statement:

```
FOR i := expr1 DOWNTO expr2 DO statement
```

The following compound statement is functionally equivalent to this **FOR** statement. Variables `temp1` and `temp2` are compiler-generated temporary variables.

```
BEGIN
   temp1 := expr1;
   temp2 := expr2;
   IF temp1 >= temp2 THEN
      BEGIN
         i := temp1;
         statement;
         WHILE i <> temp2 DO
            BEGIN
               i := PRED( i );
               statement
            END;
      END;
END;
```

Other examples of the **FOR** statement are as follows:

Find the Maximum Integer in an Array of Integers:

```
max := a[1];
largest := 1;
FOR i := 2 TO size_of_a DO
   IF a[i] > max THEN
      BEGIN
         largest := i;
         max := a[i]
      END;
```

Matrix Multiplication:

```
FOR i := 1 TO n DO
   FOR j:= 1 TO n DO
      BEGIN
         x := 0.0;
         FOR k := 1 TO n DO
            x := a[i,k] * b[k,j] + x;
         c[i,j] := x   (* c<-a*b *)
      END;
```

Sum All Hours Worked This Week:

```
sum := 0;
FOR day := mon TO fri DO
   sum := sum + timecard[ day ]
```

GOTO

Purpose

Changes the flow of control within the program. The **GOTO** names a labeled statement as its successor.

Syntax

```
── GOTO ── label ─┤
```

Parameters

GOTO	is a reserved word.
label	is an unsigned integer in the range 0 to 9999. It must be declared in the label declaration part of the routine with the **GOTO** statement. In VS mode, *label* can also be an identifier.

Restrictions

- If a **GOTO** to a nonlocal label exits a function, the function result is not checked.
- The **GOTO** must be contained by the routine that declared the label.
- You cannot branch into a compound statement from a **GOTO** statement.
- You cannot branch into a **THEN** clause or an **ELSE** clause from a **GOTO** statement outside an **IF** statement. You cannot branch between a **THEN** clause and an **ELSE** clause in the same **IF** statement.
- You cannot branch into a **CASE** alternative from outside the **CASE** statement, nor branch between **CASE** alternative statements in the same **CASE** statement.
- You cannot branch into a **FOR**, **REPEAT**, or **WHILE** loop from a **GOTO** statement not contained within the loop.
- You cannot branch into a **WITH** statement from a **GOTO** statement outside the **WITH** statement.
- For a **GOTO** statement that specifies a label defined in an outer routine, the target label cannot be defined within a compound statement or loop.
- You cannot branch out of a procedure declared with the **EXTERNAL** routine directive.

Example

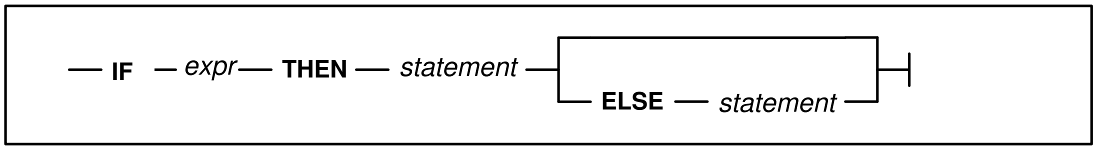
```
PROCEDURE goto_example;

LABEL
   l1, l2, l3, l4;

PROCEDURE inner;
   BEGIN
      GOTO l4;          (* permitted     *)
      GOTO l3;          (* not permitted *)
   END;


BEGIN
   GOTO l3;             (* not permitted *)
   BEGIN
      l3 : GOTO l4;     (* permitted     *)
      GOTO l3;          (* permitted     *)
   END;
   l4:IF expr THEN
      l1: GOTO l2       (* not permitted *)
   ELSE
      l2: GOTO l1       (* not permitted *)
END;
```

IF

Purpose

Specifies that one of two statements is to be processed depending on the evaluation of a Boolean expression.

Syntax



Parameters

IF	is a reserved word
expr	is any Boolean expression
THEN	is a reserved word
statement	is any statement
ELSE	is a reserved word

Description

The expression must be evaluated to a Boolean value. If the result of the expression is **TRUE**, the statement in the **THEN** clause is processed. If the expression evaluates to **FALSE** and there is an **ELSE** clause, the statement in the **ELSE** clause is processed; if there is no **ELSE** clause, control passes to the next statement.

Nested IF Statements

To select one out of many conditions, you can nest **IF** statements in a series of **ELSE**–**IF** clauses. For example:

```
VAR
   inpchar : CHAR
   .
   .

BEGIN
   .
   .
   IF ( inpchar = ’A’ ) THEN
      statement1;
   ELSE
      BEGIN
         statement2;

         IF ( inpchar = ’B’ ) THEN
            statement3;
         ELSE
            BEGIN
               statement4;
               IF ( inpchar = ’C’ ) THEN
                  statement5;
            END;
      END;
   .
   .
END;
```

Like other programming languages, XL Pascal accommodates the so-called *dangling else.* This condition arises when one **IF** statement contains another **IF** but only one **ELSE** clause. The **ELSE** clause must always be paired with the innermost **IF** statement with no **ELSE** clause.

Nesting an **IF** statement within an **IF** statement can be interpreted with two different meanings if only one of the statements has an **ELSE** clause. The following example illustrates this condition, and the two resulting interpretations.

For this statement:

```
IF b1 THEN IF b2 THEN stmt1 ELSE stmt2
```

Interpretation assumed by XL Pascal:

```
IF ( b1 ) THEN
   BEGIN
     IF ( b2 ) THEN
        stmt1
     ELSE
        stmt2;
   END
```

Alternate interpretation:

```
IF ( b1 ) THEN
   BEGIN
      IF ( b2 ) THEN
         stmt1
   END
ELSE
   stmt2;
```

If you prefer the second interpretation, either write it as shown or take advantage of the empty statement, as illustrated in the following example:

```
IF ( b1 ) THEN
   IF ( b2 ) THEN
      stmt1
   ELSE
      (* empty statement *)
ELSE
   stmt2;
```

Examples

The following example shows simple **IF** statements:

```
IF a <= b THEN
   a := ( a + 1.0 ) / 2.0;
IF ODD( i ) THEN
   j := j + 1
ELSE
   j := j DIV 2 + 1;
```

The following example shows an **IF** statement that controls two compound blocks:

```
IF a > b + c THEN
   BEGIN
      WRITELN( 'Found a > b + c' );
      a := a - delta
   END
ELSE
   BEGIN
      WRITELN( 'Found a <= b + c' );
      a := a + delta
   END;
```

LEAVE

Purpose

Causes an immediate, unconditional exit from the innermost enclosing **FOR**, **WHILE**, or **REPEAT** loop.

Syntax

```
── LEAVE ──┤
```

Examples

```
p := first;
WHILE p <> NIL DO
   IF p@.NAME = 'MIKE SMITH' THEN
      LEAVE
   ELSE
      p := p@.NEXT ;
      (* p either points to the desired data or is NIL *)
```

This portion of code with a **LEAVE** statement

```
WHILE expr DO
   BEGIN
      ...
      LEAVE
      .
      .
   END;
```

is equivalent to

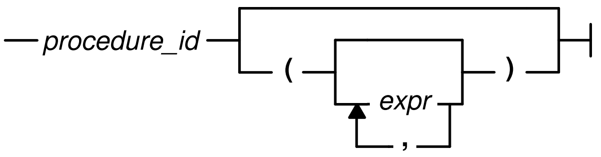
```
WHILE expr DO
   BEGIN
      .
      .
      GOTO label;
      .
      .
   END;
label: ;
```

Procedure Call

Purpose

Passes control to a procedure.

Syntax



Parameters

procedure_id	is any defined procedure
expr	is any valid expression compatible with the type of the corresponding formal parameter

Description

When a procedure is called, the actual parameters are substituted for the corresponding formal parameters. An actual parameter corresponds to the formal parameter that occupies the same ordinal position in the formal parameter list. The actual parameters must conform to the formal parameter types.

Parameters passed by read/write reference (**VAR**) can only be variables, and can never be expressions or constants. In standard mode, actual and formal **VAR** parameters must be of the same type. Also in standard mode, fields of a packed record can neither be passed by **VAR**, nor can elements of packed arrays be passed by **VAR**. Parameters passed by value or read-only reference (**CONST**) can be any expression.

VS Mode

You can pass components of packed objects by **VAR** parameters in VS mode only.

If a user-defined procedure requires no parameters, an empty set of parentheses can be used on a procedure call to distinguish it from a variable.

Example

```
transpose( an_array, num_of_rows, num_of_columns );

matrix_add( a_array, b_array, c_array, n, m );

xyz( i + j, k * l );

matrix_sum( );
```

Related Information

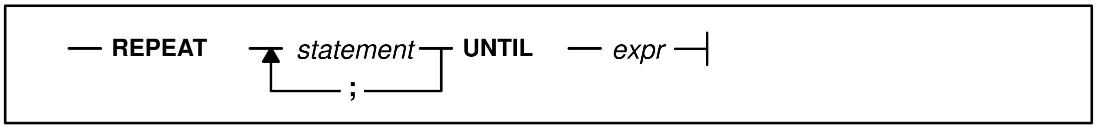
For parameter compatibility rules, see “Routine Parameters” on page 136. The rules for expression compatibility are given in “Type Compatibility” on page 50. See Chapter 10, “Routines”, for information about defining procedures.

REPEAT

Purpose

Controls the repetitive processing of a list of statements.

Syntax



Parameters

REPEAT	is a reserved word.
statement	is any valid statement.
UNTIL	is a reserved word.
expr	is any Boolean expression. This expression is evaluated after each processing of the statement.

Description

The statements contained between the statement delimiters **REPEAT** and **UNTIL** are processed until the control expression is evaluated to **TRUE**. The control expression must be of type **BOOLEAN**. Because a **REPEAT** can contain a list of statements, it can act as a compound statement.

Because the termination test is at the end of the loop, the body of the loop is always processed at least once. Contrast this mechanism with the **WHILE** statement.

Example

The following example shows a **REPEAT** statement in which the greatest common factor of `i` and `j` is stored in `i`.

```
REPEAT
   k := i MOD j;
   i := j;
   j := k
UNTIL j = 0;
```

RETURN

Purpose

Permits an exit from a procedure or function.

Syntax

```
── RETURN ──┤
```

Description

This statement is in effect a **GOTO** to an imaginary label after the last statement in the routine being processed. If the **%CHECK** option is enabled, XL Pascal ensures that a function has been assigned a value before the return from the function. If a value has not been assigned, a runtime error occurs.

Example

```
PROCEDURE p;
   BEGIN
      ...
      IF expr THEN RETURN;
      ...
   END;
```

WHILE

Purpose

Allows you to specify a statement to be processed while a control expression remains **TRUE**.

Syntax

```
── WHILE ── expr ── DO ── statement ─┤
```

Parameters

WHILE	is a reserved word.
expr	is any Boolean expression. The expression is evaluated before each time the statement is processed.
DO	is a reserved word.
statement	is any statement.

Description

The expression must be of type **BOOLEAN**. The condition is tested before the statement is processed the first time through the loop. The statement is processed repeatedly until the control expression is evaluated to **FALSE**. At this point, control passes to the statement after the **WHILE** statement.

If the value of the **BOOLEAN** expression is **FALSE** when the **WHILE** statement is encountered for the first time, the subordinate statement is never processed. Contrast this mechanism with the **REPEAT** statement.

Example

The following example calculates the decimal size of `N` and assumes `N` is greater than or equal to 1:

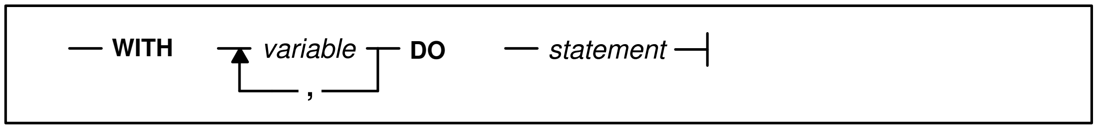
```
i := 0;
j := 1;
WHILE n > 10 DO
   BEGIN
      i := i + 1;
      j := j * 10;
      n := n DIV 10
   END;
(* i is the power of ten of the original n *)
(* j is ten to the i power ; 1 <= n <= 9   *)
```

WITH

Purpose

Simplifies references to a record variable by eliminating an addressing description on every reference to a field.

Syntax



Parameters

WITH	is a reserved word
variable	is any record variable
DO	is a reserved word
statement	is any valid statement

Description

The **WITH** statement makes the fields of a record available as if the fields were variables in the nested statement. Its effect is to insert implicitly the required record identifier before the name of each field of any record variables in the statement.

Examples

In the following example, the variable `father` is a pointer to a dynamic variable of the type `employee`. You must use pointer notation to specify the `employee` record.

For the following declarations:

```
TYPE employee = RECORD
                   name   : STRING( 20 );
                   man_no : 0..999999;
                   salary : INTEGER;
                   id_no  : 0..999999
                END;

VAR
   father : @ employee;
   ...

NEW ( father );
```

this code segment:

```
WITH father@ DO
   BEGIN
      name   := ’SMITH’;
      man_no := 666666;
      salary := weekly_salary;
      id_no  := man_no
   END;
```

is equivalent to

```
BEGIN
   father@.name   := 'SMITH';
   father@.man_no := 666666;
   father@.salary := weekly_salary;
   father@.id_no  := father@.man_no
END;
```

The **WITH** statement in effect computes the address of a record variable upon processing the statement. Any modification to a variable that changes the address computation is not reflected in the precomputed address within the **WITH** statement, as illustrated in the following example:

```
VAR
   a : ARRAY [ 1..10 ] OF RECORD
          field : INTEGER
       END;
   ...
   i := 1;
WITH a[i] DO
   BEGIN
      k := field;   (* k:=a[1].field *)
      i := 2;
      k := field;   (* k:=a[1].field *)
   END;
```

The comma notation of a **WITH** statement is an abbreviation of nested **WITH** statements. The names within a **WITH** statement are in a scope such that the last **WITH** statement takes precedence. A variable with the same name as a field of a record becomes unavailable in a **WITH** statement that specifies the record, as shown in the following example:

```
VAR
   v : RECORD
          v2 : INTEGER;
          v1 : RECORD
                  a : REAL
               END;
          a : INTEGER
       END;
   a : CHAR;
   ...

WITH v, v1 DO
   BEGIN
      v2  := 1;  (* v.v2  := 1                             *)
      a   := 1.0;(* v.v1.a := 1.0                          *)
      v.a := 1   (* v.a   := 1 ; CHAR a is not available here *)
   END;
a := 'A';     (* CHAR a is now available  *)
```

Chapter 10. Routines

This chapter describes *procedures* and *functions*, known collectively as *routines*.

Routines are the building blocks of XL Pascal programs. They define a block of statements to be processed as a unit each time the procedure or function is called.

Procedures can be thought of as adding new blocks of statements to the language. They effectively increase the language to a superset language tailored to your specialized needs. A procedure has no value associated with its name.

Functions add new operators to the language. Because they act as expressions to compute and return a value, they add to your ability to manipulate data exactly as you want.

Procedures and functions can return data three ways:

- Through the function results.
- Through **VAR** parameters.
- By assigning variables outside the lexical scope of the routine making the assignment. These variables are said to be *global* to the routine.

Using a routine identifier within the body of that routine implies *recursive* processing of that routine. Recursive processing does not occur when the function identifier appears on the *left* side of an assignment statement. This placement indicates assignment to the function variable rather than recursive activation of the function.

With XL Pascal, you can nest routines at least 20 levels deep.

Related Information

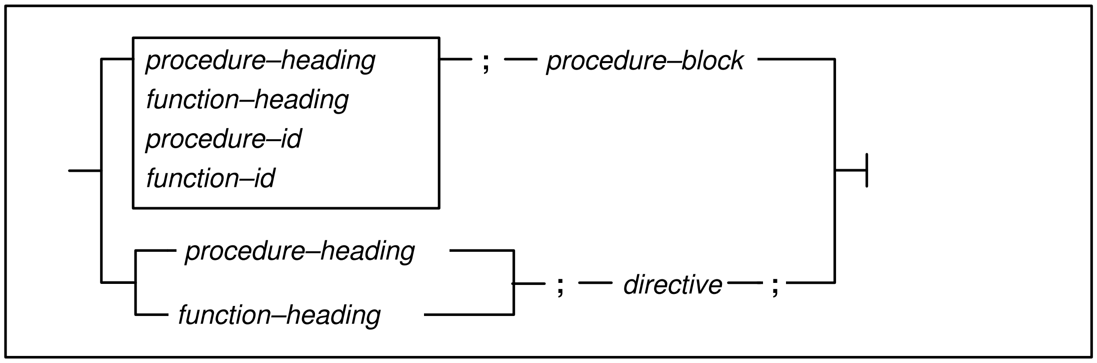
For more information on functions in assignment statements, see “Assignments to Variables and Functions” on page 113.

Routine Declarations

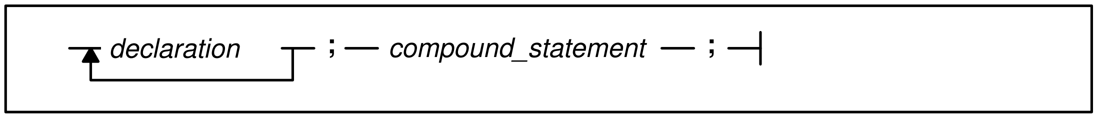
You must declare routines before using them. A routine declaration consists of:

- Routine heading
- Declarations of local labels and identifiers
- One compound statement

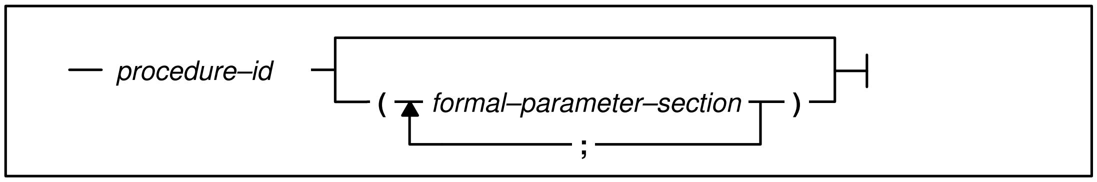
Syntax



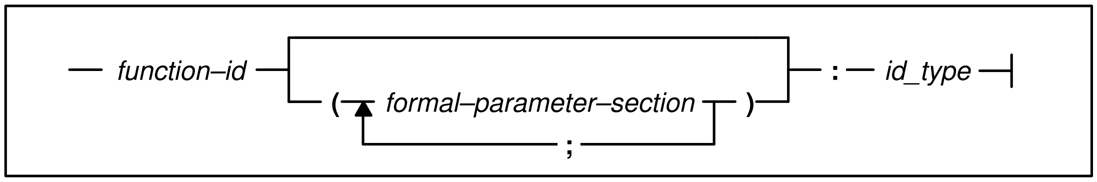
Procedure Block



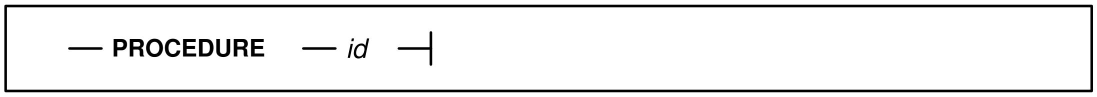
Procedure Heading



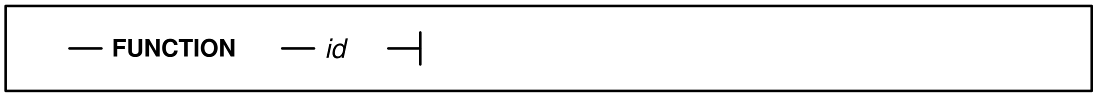
Function Heading



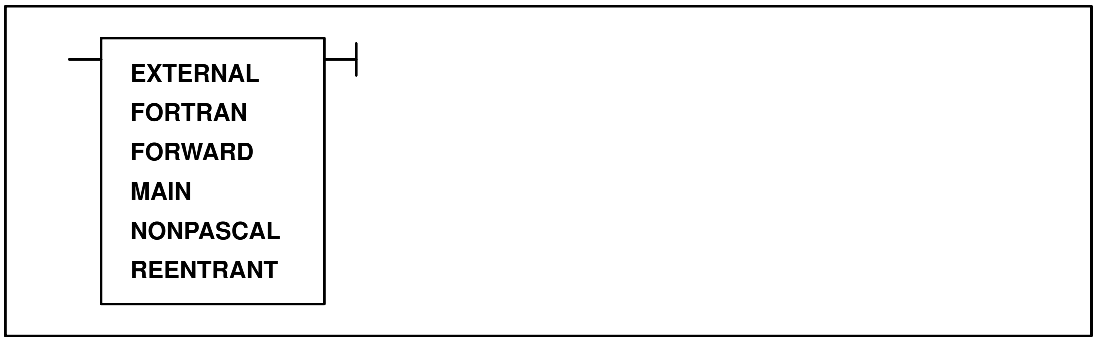
Procedure–id



Function–id

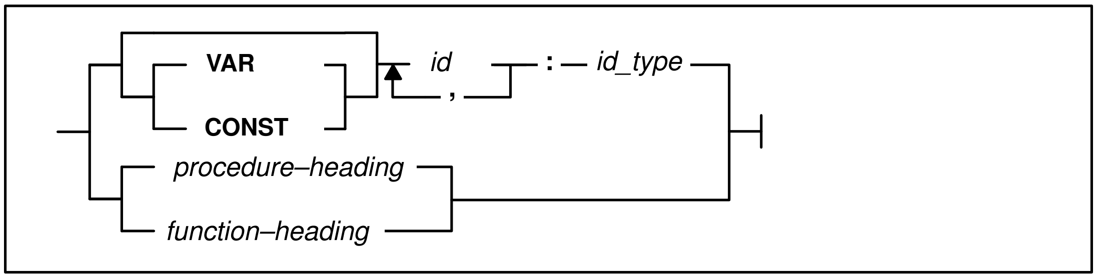


Directive



Note: **MAIN**, **REENTRANT**, and **FORTRAN** directives are provided for VS Pascal compatibility.

Formal–parameter–section



Description

In VS mode, you can declare procedures, functions, labels, constants, and variables in any order. Multiple declarations of these elements are also permitted. In both VS mode and standard mode, a user-defined routine with the same name as a predefined XL Pascal routine takes precedence over the predefined routine.

The routine heading defines the name of the routine, and binds the formal parameters to the routine. The heading of a function declaration also binds the function name to the type of value returned by the function. Formal parameters represent data to be passed to the routine when it is called. The procedure block can contain any number of the following declarations:

- **CONST**
- **DEF** (VS Mode only)
- **LABEL**
- **REF** (VS Mode only)
- **STATIC** (VS Mode only)
- **TYPE**
- **VALUE** (VS Mode only)
- **VAR**
- Routine

The compound statement is processed when the routine is called.

Examples

```
STATIC
   c : CHAR;

FUNCTION getchar : CHAR; EXTERNAL;

PROCEDURE expr ( VAR val : INTEGER ); EXTERNAL;

PROCEDURE factor ( VAR val : INTEGER); EXTERNAL;

PROCEDURE factor;

   BEGIN
      c := getchar;
      IF c = '(' THEN
         BEGIN
            c := getchar;
            expr( val )
         END
      ELSE
      ...
   END;

PROCEDURE expr;
   (* defined above as EXTERNAL, with VAR val : INTEGER *)

   BEGIN
      factor( val );
      ...
   END;
```

Related Information

Declarations are described in Chapter 4.

Routine Parameters

A routine can have *formal parameters* associated with it when it is defined. These parameters define what kind of data can be passed to the routine when it is called; they determine how the data is passed. They also permit a routine to process different sets of data in different invocations of the routines.

When the routine is called, a list of *actual parameters* is built. These are substituted for the formal parameters, which then become local variables initialized to the value of the actual parameters.

The formal parameters declared in a routine and the actual parameters supplied when the routine is activated must agree both in number and type.

Routine Parameters in Standard Mode

XL Pascal in standard mode permits parameters to be passed in the following ways:

- Pass-by-value
- Pass-by-read/write-reference (**VAR**)
- Formal routine parameter

Pass-by-Value Parameters

These parameters are like local variables initialized by the caller. The called routine can change the value of this kind of parameter, but the change is never returned to the caller.

If the actual parameter is a scalar, the parameter list contains the value of the actual parameter. If it is an **ARRAY**, **RECORD**, **SET**, **SPACE**, **STRING**, or has any pointer type, the parameter list contains the address of the actual parameter. The called procedure copies the parameter into its local storage.

All expressions, variables, or constants that are assignment compatible with the declared formal routine parameters can be passed with this mechanism. The exception is a parameter that is a file type or any type that even indirectly contains a file type. If the actual parameter is a scalar, the parameter list contains the value of the actual parameter.

Pass-by-VAR Parameters

Pass-by-**VAR** (variable) parameters are also called *pass-by-read/write-reference* parameters. Parameters passed by **VAR** reflect modifications to the parameters back to the actual parameter. You can use this parameter type as both an input and output parameter. The use of the **VAR** symbol before a parameter indicates that the parameter is to be passed by read/write reference. Only variables can be passed by this mechanism; expressions and constants *cannot* be passed this way.

In standard mode, actual routine parameters must be the same type as the formal routine parameters. The following *cannot* be passed as **VAR** parameters in standard mode:

- Fields of a packed record
- Elements of a packed array
- A field that is the selector of a variant part

The **VAR** parameters should be distinct actual variables. It is poor programming style to supply the same variable to more than one actual parameter in a routine reference, although no compile-time or runtime error results.

All index computations, field selection, and pointer referencing are done at the time the routine reference is made.

VS Mode:
Actual routine parameters corresponding to pass-by-**VAR** formal parameters must either be the same type or subrange variables. A runtime trap could occur if the value being assigned is out of range. Fields of a packed record or elements of a packed array can be passed as **VAR** parameters in VS mode XL Pascal.

Formal Routine Parameters

A procedure or function can be passed to a routine as a formal parameter by specifying the routine in the parameter list. Within the called routine, the formal parameter can be used as a procedure or function.

When you use actual and formal routine parameters, both routines must be either procedures or functions with the same result type. Also, the formal parameter lists of the actual and formal parameters must be congruous. Parameter lists are congruous under the following conditions:

- Both lists contain the same number of formal parameter sections
- Corresponding formal parameter sections match as defined in the ANSI–83 Pascal standard

VS Mode:

Formal parameters match when both are:

- Value parameters with the same type
- **VAR** parameters with the same type
- **CONST** parameters with the same type
- For a procedure, parameters with congruous parameter lists
- For a function, parameters with congruous parameter lists and the same result type

Routine Parameters in VS Mode

XL Pascal in VS mode permits parameters to be passed in the following ways:

- Pass-by-read-only-reference (**CONST**) (VS mode only)
- Pass-by-conformant-string (**VAR** or **CONST**) (VS mode only)

Pass-by-CONST Parameters

These are also called pass-by-read-only-reference parameters. They appear to be constants from the point of view of the called routine, which is not permitted to alter pass-by-**CONST** parameters.

Any expression, variable, or constant can be passed by **CONST**. Fields of a packed record and elements of a packed array can also be passed. If you pass a **CONST** pointer type parameter to either of the predefined procedures **DISPOSE** or **RELEASE**, the parameter is not set to **NIL**.

The use of the **CONST** reserved word in a parameter indicates that the parameter is to be passed by this mechanism. With parameters that are structures (such as strings), passing by **CONST** is usually more efficient than passing by value. Actual routine parameters must be assignment compatible with the declared formal routine parameters.

Conformant String Parameters

You may want to call a procedure or function and pass in a string whose declared length does not match that of the formal parameter. Use the conformant string parameter for this purpose.

You can declare a parameter or function return type as **STRING** with no length specification as follows:

```
FUNCTION ( parm : STRING ) : STRING ;
```

The interpretation of such a declaration depends on its context.

When a conformant string declaration appears as the type for a pass-by-value parameter or as the type for a value returned by a function, it is treated as a **STRING** of default length 255. Because strings are generally compatible, the parameter accepts any string type as its argument; however, truncation errors can occur if the actual string length exceeds 255.

When a conformant string parameter appears as the type for a pass-by-**CONST** or pass-by-**VAR** parameter, no assumption is made about the string size. Strings of any declared length conform to such a parameter. The actual string is available, and you can use the **MAXLENGTH** function to obtain the declared maximum length.

Example:

```
PROCEDURE translate ( VAR s : STRING; CONST table : STRING );

   VAR
      i : 0..32767;
      j : 1..ORD( HIGHEST (CHAR) ) + 1;

   BEGIN
      FOR i := 1 TO LENGTH( s ) DO
         BEGIN
            j := ORD( s[i] ) + 1;
            IF j > LENGTH( table ) THEN
               s[i] := ' '
            ELSE
               s[i] := table[j];
         END;
   END;
```

Routines That Can Be Passed as Parameters

Standard mode XL Pascal does not allow any predefined routines to be passed as actual parameters.

VS mode XL Pascal allows the predefined routines in the following table to be passed as actual routine parameters to another routine.

ARCTAN	EXP	LTOKEN	SIN
CLOCK	GTOSTR	PARMS	SQRT
COLS	HALT	PICTURE	STOGSTR
COS	ITOHS	RANDOM	TOKEN
DATETIME	LN	RETCODE	TRACE

Function Results

To return a result from a function, assign a value to the function name before leaving the function. This value is inserted in the expression at the point of the call. The value must be assignment-compatible with the declared function type.

If a function is used to evaluate itself (such as being on the right side of an assignment to the function itself), it is a recursive call. The following example shows the function `factorial`, which calls itself if `x` is greater than 1.

```
FUNCTION factorial ( x : INTEGER ) : INTEGER;

   BEGIN
      IF x <= 1 THEN
         factorial := 1      (* return result *)
      ELSE
         factorial := x * factorial( x-1 );
         (* recursive function call *)
   END;
```

Standard mode XL Pascal permits a function to return only a scalar or a pointer value.

VS Mode

XL Pascal lets a function return any type except a file or any type containing a file. You can write a Pascal function that returns a record structure as its result. You may want to use a record structure to implement a complex arithmetic library, as illustrated in the following example:

```
TYPE complex = RECORD
                  r, i : REAL
               END;

FUNCTION cadd ( CONST a, b : complex ) : complex;

   VAR
      c : complex;

   BEGIN
      c.r := a.r + b.r;
      c.i := a.i + b.i;
      cadd := c
   END;
```

A function can also return a **STRING** or a **GSTRING**. If you do not specify the maximum length of the **STRING** or **GSTRING**, a value of 255 is returned.

Pointer-Valued Functions

A function that returns a pointer value is known as a *pointer-valued* function. You cannot apply the pointer-dereferencing operator to the name of a pointer-valued function within the function body. You cannot refer to the target of the returned value in a pointer-valued function by using the function name.

Examples

The following example shows a pointer-valued function that returns a **STRINGPTR** and allows access to a dynamic string.

```
(* Function to allocate a dynamic string, provide an initial  *)
(* value and return the stringptr that allows access to       *)
(* the string                                                 *)
FUNCTION dynstr ( sz : INTEGER) : STRINGPTR;

    VAR
       wsp : STRINGPTR;

    BEGIN;
       NEW( wsp, sz );       (* allocate dynamic string with   *)
                             (* max length based on sz         *)
       dynstr := wsp;        (* set return value for function  *)
       wsp@ := '';           (* initialize string to empty     *)
                             (* NOTE: coding this as:          *)
                             (* dynstr@ := '' ;                *)
                             (* would result in a recursive    *)
                             (* invocation of this function,   *)
                             (* which is not correct           *)
    END;
    .
    .

    VAR
       msp : STRINGPTR;

    BEGIN;
       msp := dynstr( 25 ); (* create and initialize dynamic  *)
                            (* string with max size 25        *)
    END;
```

Routine Directives

You must declare a routine before you call it. The compiler can assure the validity of a call by checking parameter compatibility.

XL Pascal provides three routine directives:

- **FORWARD**
- **NONPASCAL**
- **EXTERNAL** (VS mode only)

For VS Pascal compatibility, and to assist in migration of VS Pascal programs, XL Pascal recognizes the following routine directives in both VS mode and standard mode:

- **FORTRAN**
- **MAIN**
- **REENTRANT**

They are equivalent to the **EXTERNAL** directive, but they have no effect on an XL Pascal program. They are not described here; for complete information about their use, see the *VS Pascal Language Reference (Release 2)*.

FORWARD Routines

The **FORWARD** directive identifies a routine whose heading is declared in advance of the routine body. The declaration consists only of the routine heading, followed by the **FORWARD** routine directive. To declare the body of a **FORWARD** routine, once again declare the routine heading, but omit the formal parameter list. Declare only the *procedure-id* and the *procedure-body* or the *function-id* and the *function-body* of the routine.

Declaring a routine **FORWARD** lets you call a routine before actually defining the body of the routine. This is particularly useful when two routines call each other and are at the same nesting level; one of the routines must be declared **FORWARD**.

When a routine makes a recursive call to itself, a **FORWARD** declaration is not needed.

Examples

The following program illustrates the use of the Pascal directives **FORWARD** and **NONPASCAL**. Note that when the procedure `sineof` is defined, the procedure parameters do not have to be declared again, because they were previously declared in the **FORWARD** declaration of the procedure.

```
PROGRAM nf;

PROCEDURE convert ( VAR r : REAL ); NONPASCAL; (* language  *)
                     (* program written in another language  *)

PROCEDURE sineof ( VAR r : REAL ); FORWARD;
    (* procedure declared ahead of its definition *)

FUNCTION calc ( r : REAL ) : REAL;
    BEGIN
       (* function using procedures convert and sineof *)
       convert( r );
       sineof( r );
       calc := r;
    END;

PROCEDURE sineof;        (* procedure sineof defined  *)
   BEGIN
       r := SIN( r );
   END;

BEGIN
    WRITELN( calc ( 2.0 ) );
END.
```

NONPASCAL Routines

The **NONPASCAL** directive is a special case of the **EXTERNAL** directive required by the parameter passing conventions of calls to other XL languages. The **NONPASCAL** directive is a safer, though slightly less efficient, parameter passing mechanism for floating-point data. The previous example shows the **NONPASCAL** routine.

If you are not sure how floating-point values are passed by the programs you are linking, you may want to use **NONPASCAL** instead of the **EXTERNAL** directive. See the *User's Guide for IBM AIX XL Pascal Compiler/6000* for more information on interlanguage communication.

EXTERNAL Routines

An external routine is a procedure or function that can be called from outside of its lexical scope (such as another unit). The **EXTERNAL** directive specifies the heading of such a routine. It is available only in VS mode.

Although many units can call an external routine, only one unit actually contains the body of the routine. The formal parameters defined in the external routine declaration must match those in the unit where the routine is defined. An external routine declaration can refer to an XL Pascal routine located later in the same unit or located in another segment unit. It can also refer to code produced by other means (such as assembler code).

The body of an external routine can be defined only in the outermost nesting level of a unit; that is, it must not be nested in another routine.

Example

The following example illustrates two units, the program unit `test` and a segment unit `seg`, that share a single external routine. Both units can call the routine, but only one contains the definition of the routine.

```
PROGRAM test;

FUNCTION square ( x : REAL ) : REAL; EXTERNAL;

   BEGIN
       WRITELN ( square ( 44 ) );
   END.

SEGMENT seg;

FUNCTION square ( x : REAL ) : REAL; EXTERNAL;

FUNCTION square;

   BEGIN
       square := x * x
   END; .
```

Internal Routines

An internal routine can be called only from within the lexical scope containing the routine definition. It can only have the **FORWARD** directive in its declaration.

Predefined Routines

XL Pascal provides a wide range of predefined procedures and functions. The following is a summary all of the predefined routines by category.

Conversion

These routines perform conversions from one data type to another.

Standard Mode:

The **CHR** function, described on page 147
The **ORD** function, described on page 170
The **ROUND** function, described on page 185
The **TRUNC** function, described on page 194

VS Mode:

The **FLOAT** function, described on page 153
The **GSTR** function, described on page 154
The **GTOSTR** function, described on page 155
The **ITOHS** function, described on page 157
Ordinal conversion functions, described on page 170
The **STOGSTR** function, described on page 188
The **STR** function, described on page 189

Data Access

These routines let you inquire about compile-time and runtime bounds and values.

Standard Mode:

The **ODD** function, described on page 170
The **PRED** function, described on page 175
The **SUCC** function, described on page 191

VS Mode:

The **ADDR** function, described on page 147
The **HBOUND** function, described on page 155
The **HIGHEST** function, described on page 156
The **LBOUND** function, described on page 157
The **LOWEST** function, described on page 159
The **MAX** function, described on page 162
The **MIN** function, described on page 164
The **SIZEOF** function, described on page 187

Data Movement

These routines provide you with efficient ways of reformatting when you are moving large amounts of data.

The **PACK** procedure, described on page 171
The **UNPACK** procedure, described on page 194

General

These VS mode routines provide several useful features of the XL Pascal runtime environment.

The **HALT** procedure, described on page 155
The **TRACE** procedure, described on page 193
The **xl__trap** procedure, described on page 203

Input/Output

An XL Pascal program communicates through input and output (I/O) facilities. Input and output are done using the file data structure. There are two types of files in XL Pascal: **TEXT** files and record files. The predefined input/output routines let you write to and read from these files.

The *User's Guide for IBM AIX XL Pascal Compiler/6000* provides more detail on how to use the XL Pascal I/O routines in the AIX Version 3 Operating System.

Standard Mode:

The **EOF** function (record or **TEXT** files), described on page 152
The **EOLN** function (**TEXT** files only), described on page 152
The **GET** procedure (record or **TEXT** files), described on page 154
The **PAGE** procedure (**TEXT** files only), described on page 172
The **PUT** procedure (record or **TEXT** files), described on page 176
The **READ** procedure (record or **TEXT** files), described on page 177
The **READLN** procedure (**TEXT** files only), described on page 177
The **RESET** procedure (record or **TEXT** files), described on page 183
The **REWRITE** procedure (record or **TEXT** files), described on page 184
The **WRITE** procedure (record or **TEXT** files), described on page 196
The **WRITELN** procedure (**TEXT** files only), described on page 197

VS Mode:

The **CLOSE** procedure (record or **TEXT** files), described on page 148
The **COLS** function (**TEXT** files only), described on page 148
The **SEEK** procedure (record files only), described on page 186
The **TERMIN** procedure (**TEXT** files only), described on page 191
The **TERMOUT** procedure (**TEXT** files only), described on page 191
The **UPDATE** procedure (record files only), described on page 195

Mathematical

These routines define various mathematical operations.

Standard Mode:

The **ABS** function, described on page 147
The **ARCTAN** function, described on page 147
The **COS** function, described on page 149
The **EXP** function, described on page 153
The **LN** function, described on page 159
The **SIN** function, described on page 187
The **SQR** function, described on page 188
The **SQRT** function, described on page 188

VS Mode:

The **RANDOM** function, described on page 176

Mixed String Support (VS Mode)

This set of VS mode routines lets you manipulate strings with a mixture of single-byte characters and Extended, or Multibyte, Character Set (MBCS) characters. These routines recognize and preserve MBCS characters. The arguments of mixed string support routines must be defined as type **STRING**, never type **GSTRING**.

The **MCOMPRESS** function, described on page 163
The **MDELETE** function, described on page 163
The **MINDEX** function, described on page 164
The **MLENGTH** function, described on page 165
The **MLTRIM** function, described on page 165
The **MRINDEX** function, described on page 165
The **MSUBSTR** function, described on page 166
The **MTRIM** function, described on page 167

Storage Management

These routines let you control the allocation of dynamic variables. Several routines manage these variables in a collection called a *heap*.

Standard Mode:

The **DISPOSE** procedure, described on page 150
The **NEW** procedure, described on page 167

VS Mode:

The **DISPOSEHEAP** procedure, described on page 151
The **MARK** procedure, described on page 161
The **NEWHEAP** procedure, described on page 169
The **QUERYHEAP** procedure, described on page 176
The **RELEASE** procedure, described on page 182
The **USEHEAP** procedure, described on page 196

STRING Manipulation (VS Mode)

These VS mode routines provide a convenient means of operating on string data. They manipulate single-byte and mixed strings in a byte-oriented manner.

The **COMPRESS** function, described on page 149
The **DELETE** function, described on page 150
The **INDEX** function, described on page 156
The **LENGTH** function, described on page 158
The **LPAD** procedure, described on page 160
The **LTOKEN** procedure, described on page 160
The **LTRIM** function, described on page 161
The **MAXLENGTH** function, described on page 162
The **PICTURE** function, described on page 173
The **READSTR** procedure, described on page 181
The **RINDEX** function, described on page 184
The **RPAD** procedure, described on page 186
The **SUBSTR** function, described on page 190
The **TOKEN** procedure, described on page 192
The **TRIM** function, described on page 193
The **WRITESTR** procedure, described on page 201

System Interface (VS Mode)

These VS mode routines provide interfaces to system facilities. In general, they are dependent on the implementation of XL Pascal.

The **CLOCK** function, described on page 148
The **DATETIME** procedure, described on page 149
The **PARMS** function, described on page 172
The **RETCODE** procedure, described on page 184

ABS Function

Purpose

Returns the absolute value of the parameter, which can be any numeric type.

Definition

```
  FUNCTION ABS( i : INTEGER )
                  : INTEGER ;
  FUNCTION ABS( r : REAL )
                  : REAL ;
  FUNCTION ABS( s : SHORTREAL )
                  : SHORTREAL ;
```

Where	Represents
i	an **INTEGER** expression
r	a **REAL** expression
s	a **SHORTREAL** expression (VS mode only).

ADDR Function

Purpose

Returns the address in storage of a specified variable. A variable includes qualified variables such as dereferenced pointers, subscripted variables, and fields of records.

Definition

```
  FUNCTION ADDR( v : anytype )
                   : INTEGER;
```

Where	Represents
v	an identifier declared as a variable.

ARCTAN Function

Purpose

Computes the arctangent of a floating-point number. The result is expressed in radians.

Definition

```
  FUNCTION ARCTAN( x : REAL )
                     : REAL;
```

Where	Represents
x	a **REAL** expression.

REAL functions will accept **INTEGER** and **SHORTREAL** arguments. See “Type Compatibility” on page 50 for more information.

CHR Function

Purpose

Returns the ASCII character corresponding to the given **INTEGER** value. It is the inverse of **ORD** for characters. That is, `ORD(CHR(I))=I` if `I` is in the subrange:

```
  ORD( LOWEST( CHAR ) )..ORD( HIGHEST( CHAR ) )
```

If the operand is outside this range when checking is active, a runtime error results. If checking is not active, XL Pascal acts unpredictably. See the *User's Guide for IBM AIX XL Pascal Compiler/6000* for a table of the ASCII character set and information on the values for ASCII characters.

Definition

```
FUNCTION CHR( i : INTEGER )
              : CHAR;
```

Where	**Represents**
i	an **INTEGER** expression that represents the ordinal value of a character.

CLOCK Function

Purpose

Returns the number of microseconds the program has been running.

Definition

```
FUNCTION CLOCK : INTEGER;
```

CLOSE Procedure

Purpose

Ends all processing of a specific file, and leaves the file variable for that file undefined. If a file is already closed, a **CLOSE** call does nothing. You must reopen the file before using it again.

Definition

```
PROCEDURE close( VAR f : filetype );
```

Where	**Represents**
f	a file variable.

COLS Function

Purpose

Returns the current column number (position of the next character to be written) on the output **TEXT** file designated by the file variable.

Definition

```
FUNCTION COLS( CONST f : TEXT )
                       : INTEGER;
```

Where	**Represents**
f	a **TEXT** file open for output.

You can force the output to a specific column with the following code:

```
IF tab > COLS( f ) THEN
    WRITE ( f, ' ' : tab-COLS( f ) );
```

COMPRESS Function

Purpose

Replaces multiple consecutive blanks in the specified source string with a single blank, and sequences of MBCS blanks with a single MBCS blank.

Definition

```
FUNCTION COMPRESS( CONST source : STRING)
                                : STRING;

FUNCTION COMPRESS( CONST source : GSTRING)
                                : GSTRING;
```

Where	Represents
source	a **STRING** or **GSTRING** expression to be compressed.

Note: The **COMPRESS** function works best with pure single-byte character strings or pure MBCS strings. Use the predefined routine **MCOMPRESS** for operating on mixed strings.

Examples

```
k := COMPRESS('a b cd ') ;       (* yields 'a b cd ' *)
k := COMPRESS('BBDBB'G) ;        (* yields 'BDB'G *)
```

Note: The **D** represents one MBCS character, and **B** represents one MBCS blank.

COS Function

Purpose

Computes the cosine of a floating-point number representing an angle in radians.

Definition

```
FUNCTION COS( x : REAL )
                : REAL;
```

Where	Represents
x	a REAL expression.

REAL functions will accept **INTEGER** and **SHORTREAL** arguments. See “Type Compatibility” on page 50 for more information.

DATETIME Procedure

Purpose

Returns the current date and time of day as two **ALFA** arrays.

Definition

```
PROCEDURE DATETIME( VAR date, time : ALFA );
```

Where	Represents
date	the returned date in the format `mm/dd/yy`
time	the returned time in the format `hh:mi:ss`.

Where	Represents
`mm`	the month expressed as a two-digit value
`dd`	the day of the month
`yy`	the year
`hh`	the hour
`mi`	the minute
`ss`	the second

DELETE Function

Purpose

Returns the source string with a specified portion removed.

Definition

```
FUNCTION DELETE ( CONST source : STRING;
                        start : INTEGER;
                         len  : INTEGER ) : STRING;
```

```
FUNCTION DELETE ( CONST source : GSTRING;
                        start : INTEGER;
                         len  : INTEGER ) : GSTRING;
```

Where	Represents
source	a **STRING** or **GSTRING** expression from which a portion will be deleted
start	an **INTEGER** expression that specifies the starting position within the source where characters are to be deleted
len	an optional **INTEGER** expression that specifies the number of characters to be deleted.

The first character of the source string is at position 1. If the length is omitted, all remaining characters are deleted. The string is truncated beginning at position *start*.

The following conditions must exist to avoid an error message at run time:

- *start* must be greater than 0
- *len* must be greater than or equal to 0
- *start*+*len*–1 must be less than or equal to the current length of the string.

If *len* is 0, the whole string is returned.

Note: **DELETE** works best with pure single–byte character strings or pure MBCS strings. The predefined routine **MDELETE** is recommended for operating on mixed strings.

Examples

```
k := DELETE('abcde',2,3) ;   (* yields 'ae'     *)
k := DELETE('abcde',3) ;     (* yields 'ab'     *)
k := DELETE('abcde',3,1) ;   (* yields 'abde'   *)
k := DELETE('abcde',1) ;     (* yields ''       *)
k := DELETE('abcde',6,0) ;   (* yields 'abcde'  *)
k := DELETE('abcde',2,5) ;   (* is an error     *)
```

DISPOSE Procedure

Purpose

Frees storage for a single dynamic variable, and if the pointer is a valid destination for assignment, sets the pointer to **NIL**.

Definition

```
Form 1:
 PROCEDURE DISPOSE( p1  : pointer );

Form 2:
 PROCEDURE DISPOSE( p2  : pointer;
             t1, t2... : ordinal-type);

Form 3:
 PROCEDURE DISPOSE( p3  : STRINGPTR;
                   len  : INTEGER );

 PROCEDURE DISPOSE( p3  : GSTRINGPTR;
                   len  : INTEGER );
```

Where	Represents
p1	a pointer expression returned from a call to **NEW**
p2	a pointer expression to a record returned from a call to **NEW**
t1,t2	ordinal constants representing tag fields
p3	a **STRINGPTR** or **GSTRINGPTR** expression returned from a call to **NEW**
len	an **INTEGER** expression (VS mode only).

DISPOSE frees only the storage referred to by the pointer and does not free any storage that the dynamic variable references. That is, if the dynamic variable is an element of a linked list, **DISPOSE** frees storage only for that single element. If you want to dispose of the whole list, you must explicitly dispose of every element of the list. If you have other pointers that reference the same dynamic variable that has been disposed, you should not use these pointers because the dynamic variable they represented is no longer allocated.

Form 1	deallocates storage for any **POINTER**, **STRINGPTR**, or **GSTRINGPTR** allocated using any form of **NEW**.
Form 2	works only on a pointer to a variant record type with nested variants whose tag types are assignment compatible with the tag field constants.
Form 3	can only be used to deallocate pointers previously allocated with Form 3 of **NEW**. XL Pascal checks that the specified length is an integer but ignores the length value otherwise. Form 3 is available only in VS mode.

Related Information

The **NEW** procedure is described on page 167. For information on how to free an entire subheap, see "RELEASE Procedure" on page 182.

DISPOSEHEAP Procedure

Purpose

Frees storage in a heap.

Definition

```
PROCEDURE DISPOSEHEAP( VAR p : pointer );
```

Where	Represents
p	a pointer to any type. Its value must be a heap–id set by a call to **NEWHEAP** or **QUERYHEAP**.

DISPOSEHEAP frees the dynamic variables and the mark-values in the heap designated by its argument. It also frees storage allocated for the heap-id, which is the pointer passed to a heap created by the **NEWHEAP** procedure.

If the argument of **DISPOSEHEAP** is a variable, it is set to **NIL**. If you dispose of the currently active heap with **DISPOSEHEAP**, the default heap becomes the currently active heap. If you dispose of the default heap with **DISPOSEHEAP**, the contents of the default heap are freed, but the heap-id of the default heap remains allocated.

EOF Function

Purpose

Tests a file for the end-of-file condition. **EOF** is a Boolean function that returns a value of **TRUE** if the end-of-file condition is true for the file; otherwise, it returns **FALSE**. The end-of-file condition occurs on any attempt to read past the last element of an input file. If the file is open for output, this function returns a value of **TRUE**.

Definition

```
FUNCTION EOF( f : filetype )
                : BOOLEAN;
```

Where	Represents
f	an optional file variable. The default is the predefined file **INPUT**.

Example of Testing for End-of-File Condition

In the following example, all the records are read from `sysin` and written to `sysout`.

```
TYPE frec = RECORD
                a, b : INTEGER
             END;

VAR
    sysin, sysout : FILE OF frec;

BEGIN
    RESET( sysin );
    REWRITE( sysout );
    WHILE NOT EOF( sysin ) DO
    BEGIN
       sysout@ := sysin@;
       PUT( sysout );
       GET( sysin )
    END;
END;
```

EOLN Function

Purpose

Tests a **TEXT** file for the end-of-line condition. **EOLN** is a Boolean function that returns a value of **TRUE** if the file pointer is positioned to an end-of-line character; otherwise, it returns a value of **FALSE**.

An end-of-line character is inserted when you use **WRITELN** to write out the data. If the end-of-line condition is true, the file pointer points to a blank.

Definition

```
FUNCTION EOLN( f : TEXT )
                 : BOOLEAN;
```

Where	Represents
f	an optional **TEXT** file variable. The default is the predefined file **INPUT**.

Example of Copying a Text File

In the following example, the file is copied from `sysin` to `sysout`.

```
VAR
    sysin, sysout : TEXT;

BEGIN
    RESET( sysin );
    REWRITE( sysout );
    WHILE NOT EOF( sysin ) DO
       BEGIN
          WHILE NOT EOLN( sysin ) DO
             BEGIN
                sysout@ := sysin@;
                PUT( sysout );
                GET( sysin );
             END;
          WRITELN( sysout );
          READLN( sysin );
       END;
END;
```

EXP Function

Purpose

Computes the value of the base of the natural logarithm, e, raised to the power expressed by a floating-point number.

Definition

```
FUNCTION EXP( x : REAL )
                : REAL;
```

Where	Represents
x	a **REAL** expression.

Functions of type **REAL** accept **INTEGER** and **SHORTREAL** arguments. See “Type Compatibility” on page 50 for more information on implicit type conversion.

FLOAT Function

Purpose

Converts an **INTEGER** value to a **REAL** value. Use **FLOAT** to make this conversion explicit in a program. See “Implicit Type Conversion” on page 50 for more information on type conversion.

Definition

```
FUNCTION FLOAT( i : INTEGER )
                  : REAL;
```

Where	Represents
i	an **INTEGER** expression.

GET Procedure

Purpose

Positions the file pointer of a file (previously opened for input) to the next component in the file. For example, if the file is defined to be made up of strings, each **GET** advances the file pointer to the next string in the file. A call to **GET** on a **TEXT** file causes the file pointer to be advanced to the next single character in the file.

Definition

```
PROCEDURE GET( VAR f : filetype );
```

Where	Represents
f	a file variable.

Note: **GET** can only read pure MBCS data from a **TEXT** file one byte at a time.

GSTR Function

Purpose

Converts a **GSTRING**, **GCHAR**, or **PACKED ARRAY OF GCHAR** to a **GSTRING**. XL Pascal implicitly converts an MBCS literal string to a **GCHAR** or **PACKED ARRAY OF GCHAR** on assignment, but all other conversions require you to explicitly state the conversion. If the parameter is a **GSTRING**, the function is valid but has no effect.

Definition

```
FUNCTION GSTR( x : GCHAR )
                 : GSTRING;

FUNCTION GSTR( x : PACKED ARRAY[1..n] OF GCHAR )
                 : GSTRING;

FUNCTION GSTR( x : GSTRING )
                 : GSTRING;
```

Where	Represents
x	a **GCHAR**, **GSTRING**, or **PACKED ARRAY OF GCHAR** expression.

Example

```
VAR
    gc : GCHAR;
    ga : PACKED ARRAY [1..4] OF GCHAR;
    g4 : GSTRING(4);

BEGIN
    gc := 'D'G;                (* D is stored in gc   *)
    g4 := GSTR(gc);            (* D is stored in g4   *)
    ga := 'DB'G;               (* DB is stored in ga  *)
    g4 := GSTR(ga);            (* DB is stored in g4  *)
END;
```

GTOSTR Function

Purpose

Converts a **GSTRING** to a mixed **STRING**. Data is converted from file code to process code.

Definition

```
FUNCTION GTOSTR( x : GSTRING )
                   : STRING;
```

Where	Represents
x	a **GSTRING** expression.

Example

```
VAR
    g : GSTRING( 4 );
    s : STRING( 10 );

BEGIN
    g := 'DB'G;                    (* DB is stored in g  *)
    s := GTOSTR( g )              (* DB is stored in s  *)
END;
```

HALT Procedure

Purpose

Stops the processing of an XL Pascal program.

Definition

```
PROCEDURE HALT;
```

HBOUND Function

Purpose

Returns the upper bound of an index to an array. The array can be specified in two ways:

- An identifier declared as an array or space in the **TYPE** section
- A variable of type **ARRAY** or **SPACE**

Definition

```
FUNCTION HBOUND( v : array-type;
                 i : integer-const)
                   : ordinal-type;
```

Where	Represents
v	an identifier declared as an array type or variable, or a **SPACE** type or variable
i	an optional constant expression with a positive **INTEGER** value. The default is 1.

The type of the value returned is the same as the type of the index. You use the second parameter for multidimensional arrays to define the dimension of the array for which the upper bound is returned.

The **HBOUND** function also works on **SPACE** types by returning the size of the space in bytes.

Examples

```
TYPE
    grid = ARRAY [-10..10,-5..5] OF REAL;

VAR
    a : grid;
    b : ARRAY [1..100] OF
        ARRAY [0..9] OF CHAR;
    .
    .
    k := HBOUND( a ) ;     (* is 10 *)
    k := HBOUND( grid ) ; (* is 10 *)
    k := HBOUND( b, 2 ) ; (* is 9  *)
    k := HBOUND( b[1] ) ; (* is 9  *)
```

HIGHEST Function

Purpose

Returns the highest value in the ordinal type of the operand. The operand can be either a type identifier or a variable. If it is a type identifier, the value of the function is the highest value that a variable of that type can be assigned. If it is a variable, the value of the function is the highest value that the variable can be assigned.

Definition

```
FUNCTION HIGHEST( s : ordinal-type )
                    : ordinal-type;
```

Where	**Represents**
s	an identifier declared as an ordinal type or variable.

Examples

```
TYPE
    days  = ( sun, mon, tues, wed, thu, fri, sat );
    small = 0..31 ;

VAR
    i : INTEGER;
    j : 0..255;
    .
    .
    k := HIGHEST( days ) ;     (* is SAT    *)
    k := HIGHEST( BOOLEAN ) ; (* is TRUE   *)
    k := HIGHEST( small ) ;   (* is 31     *)
    k := HIGHEST( i ) ;       (* is MAXINT *)
    k := HIGHEST( j ) ;       (* is 255    *)
```

INDEX Function

Purpose

Returns the starting index of the first instance of the second parameter within the first parameter. If the second parameter does not exist in the first parameter, **INDEX** returns a zero. If the second parameter is null, it returns a 1.

Definition

```
FUNCTION INDEX( CONST source : STRING ;
                CONST lookup : STRING )
                             : 0..32767;

FUNCTION INDEX( CONST source : GSTRING ;
                CONST lookup : GSTRING )
                             : 0..16382;
```

Where	Represents
source	a **STRING** or **GSTRING** expression to which lookup is compared
lookup	the **STRING** or **GSTRING** expression to be compared to source.

Note: **INDEX** works best with pure single-byte character strings or pure MBCS strings. Use the predefined routine **MINDEX** for operating on mixed strings.

Examples

```
VAR
    s : STRING( 10 );
    .
    .
    s := ’abcabcabc’;
    .
    .
    k := INDEX( s, ’bc’ ) ;    (* yields 2  *)
    k := INDEX( s, ’x’ ) ;     (* yields 0  *)
```

ITOHS Function

Purpose

Converts an **INTEGER** value into a string containing the hexadecimal representation of the integer.

Definition

```
FUNCTION ITOHS( i : INTEGER )
                  : STRING( 8 );
```

Where	Represents
i	the **INTEGER** expression to be converted.

Example

```
WRITELN(’The value ’,I:0,
        ’ is ’,   ITOHS( I ),
        ’ in hexadecimal.’ );
```

LBOUND Function

Purpose

Returns the lower bound of an index to an array. The array can be specified in two ways:

- As an identifier declared as an array or space in the **TYPE** section
- As a variable of type **ARRAY** or **SPACE**

Definition

```
FUNCTION LBOUND( v : array-type;
                 i : integer-const )
                   : ordinal-type;
```

Where	Represents
v	an identifier declared as an array type or variable, or a **SPACE** type or variable
i	an optional constant expression with a positive **INTEGER** value. The default is 1.

The type of the value returned is the same as the type of the index. You use the second parameter for multidimensional arrays to define the dimension of the array for which the lower bound is returned.

The **LBOUND** function also works on **SPACE** types, returning a value of 0.

Examples

```
TYPE
    grid = ARRAY [-10..10,-5..5] OF REAL ;

VAR
    a : ARRAY [1..100] OF ALFA;
    b : ARRAY [1..100] OF
          ARRAY [0..9] OF CHAR;
    .
    .
    k := LBOUND( a ) ;     (*   is 1     *)
    k := LBOUND( grid ) ; (*   is -10   *)
    k := LBOUND( b, 2 ) ; (*   is 0     *)
    k := LBOUND( b[1] ) ; (*   is 0     *)
```

LENGTH Function

Purpose

Returns the current length of the specified parameter string or **GSTRING**. For **STRING** parameters, the value is in the range 0 to 32767. For parameters of type **GSTRING**, the value is in the range 0 to 16382 characters.

Definition

```
FUNCTION LENGTH( s : STRING )
                   : 0..32767;

FUNCTION LENGTH( s : GSTRING )
                   : 0..16382;
```

Where	Represents
s	a **STRING** or **GSTRING** expression.

Note: **LENGTH** works best with pure single-byte character strings or pure MBCS strings. The predefined routine **MLENGTH** is recommended for operating on mixed strings.

Examples

```
k := LENGTH( 'abcd' ) ;       (* yields 4 *)
k := LENGTH( 'DDDD'G ) ;     (* yields 4 *)
```

Note: **D** represents one MBCS character.

LN Function

Purpose

Computes the natural logarithm of a floating-point number.

Definition

```
FUNCTION LN( x : REAL )
               : REAL;
```

Where	Represents
x	a **REAL** expression with a value greater than 0.

Functions of type **REAL** will accept **INTEGER** and **SHORTREAL** arguments. See "Type Compatibility" on page 50 for more information.

LOWEST Function

Purpose

Returns the lowest value in the ordinal type of the operand. The operand can be either a type identifier or a variable. If the operand is a type identifier, the value of the function is the lowest value that a variable of that type can be assigned. If the operand is a variable, the value of the function is the lowest value that the variable can be assigned.

Definition

```
FUNCTION LOWEST( s : ordinal-type)
                   : ordinal-type;
```

Where	Represents
s	an identifier declared as an ordinal type or variable.

Examples

```
TYPE
    days  = ( sun, mon, tues, wed, thu, fri, sat );
    small = 0..31;

VAR
    i : INTEGER;
    j : 0..255;
    .
    .
    k := LOWEST( days ) ;     (*   is SUN    *)
    k := LOWEST( BOOLEAN ) ; (*   is FALSE  *)
    k := LOWEST( small ) ;    (*   is 0      *)
    k := LOWEST( i ) ;        (*   is MININT *)
    k := LOWEST( j ) ;        (*   is 0      *)
```

LPAD Procedure

Purpose

Pads or truncates a string or **GSTRING** on the left.

Definition

```
PROCEDURE LPAD( VAR s : STRING;
                    l : INTEGER;
                    c : CHAR  );

PROCEDURE LPAD( VAR s : GSTRING;
                    l : INTEGER;
                    c : GCHAR  );
```

Where	Represents
s	the **STRING** or **GSTRING** to be padded
l	the final length of *s*
c	the pad character.

If **LENGTH**(*s*) is greater than *l*, **LPAD** truncates characters on the left. If **LENGTH**(*s*) is less than *l*, **LPAD** extends *s* with the character *c* on the left.

Example

```
s := 'abcdef';
k :=  LPAD( s, 10, '$' ) ; (* produces '$$$$abcdef' in s *)
s := 'abcdef' ;
k :=  LPAD( s, 5, '$' ) ;  (* produces 'bcdef' in s *)
```

LTOKEN Procedure

Purpose

Scans its *source* parameter looking for a *token* and returns the token as a **STRING** type. The **LTOKEN** procedure is declared with three parameters.

Definition

```
PROCEDURE LTOKEN( VAR   pos    : INTEGER;
                  CONST source  : STRING;
                  VAR   result  : STRING );
```

Where	Represents
pos	an **INTEGER** corresponding to the position in the source string where the search for the token begins. The value of this **INTEGER** is updated to the starting position for subsequent calls to **LTOKEN**.
source	a **STRING** expression containing the data from which a token is to be extracted.
result	the resulting token.

The starting position of the scan is the value of the first parameter in the **LTOKEN** call. In subsequent calls to **LTOKEN**, this parameter is changed to the position at which the scan is to be resumed.

When **LTOKEN** scans a string, it ignores leading blanks, multiple blanks, and trailing blanks. If no token is in the string, the value of the first parameter is set to **LENGTH**(*source*)+1 and the *result* parameter is set to one blank. If the token is longer than the result variable, an error results.

A token can be any of the following:

- An identifier consisting of any number of alphanumeric characters, dollar signs ($), or underscores (_). The first letter must be alphabetic or $.
- An unsigned number.
- The following special symbols:

```
+   -   *   /   ->   @   ¢
=   <>   <   <=   >=   >   !
(   )   [   ]   ’   ”   %
|   &   &&   ||   ~   ~=   #
:   ;   :=   .   ,   ..
{   }   (*   *)   /*   */
(.   .)   <<   >>
```

- Any other single character.

Example

In the following example, **LTOKEN** would return the same value if `i` were set to 3; that is, leading blanks are not used:

```
i := 2;
k := LTOKEN( i, ’, Token+’, result ) ; (* i is set to 8 *)
                                (* result is set to ’Token’ *)
```

LTRIM Function

Purpose

Returns the value of the specified parameter with all leading blanks removed. The function removes MBCS blanks from **GSTRING** data.

Definition

```
FUNCTION LTRIM( CONST source : STRING )
                             : STRING;

FUNCTION LTRIM( CONST source : GSTRING )
                             : GSTRING;
```

Where	**Represents**
source	the **STRING** or **GSTRING** to be trimmed.

Note: **LTRIM** works best with pure single-byte character strings or pure MBCS strings. The predefined routine **MLTRIM** is recommended for operating on mixed strings.

Example

```
k :=  LTRIM(’ a b ’) ;     (*  yields ’a b ’   *)
k :=  LTRIM(’   ’) ;       (*  yields ’’       *)
k :=  LTRIM(’BBBD’G) ;    (*   yields ’D’G    *)
```

MARK Procedure

Purpose

Establishes a subset of dynamic variables called a *subheap* in the currently active heap. It sets its parameter to a heap control value that designates the newly established subheap. The predefined procedure **NEW** allocates a dynamic variable in the subheap most recently established by **MARK** in the currently active heap. Storage for all dynamic variables in the subheap can be freed with a single call to the **RELEASE** procedure.

Definition

```
PROCEDURE MARK( VAR p : POINTER );
```

Where	Represents
p	a pointer to any type.

MARK does not allocate storage for dynamic variables. The pointer variable passed as parameter *p* is called a *subheap pointer*. To avoid unpredictable results, you should not use the returned pointer as the base of a dynamic variable.

MAX Function

Purpose

Returns the maximum value of one or more parameters.

Definition

```
FUNCTION MAX( expr,expr.. : scalar-type)
                          : scalar-type;
```

Where	Represents
expr	a scalar expression, including **REAL** and **SHORTREAL**.

The parameters for **MAX** can be a mixture of **INTEGER**, **REAL**, and **SHORTREAL** expressions. If the parameters are mixed and one of them is a **REAL**, a **REAL** value is returned. If the parameters are mixed and do not include a **REAL** but do include a **SHORTREAL**, a **SHORTREAL** value is returned.

MAXLENGTH Function

Purpose

Returns the maximum length of the specified parameter string. The value is in the range 0 to 32767. For parameters of type **GSTRING**, the value is in the range 0 to 16382.

Definition

```
FUNCTION MAXLENGTH( s : STRING )
                      : 0..32767;

FUNCTION MAXLENGTH( s : GSTRING )
                      : 0..16382;
```

Where	Represents
s	a **STRING** or **GSTRING** expression.

Example

```
VAR
    s : STRING( 8 );
    g : GSTRING( 4 );
    .
    .
    k :=  MAXLENGTH( s ) ; (* yields 8 *)
    k :=  MAXLENGTH( g ) ; (* yields 4 *)
```

MCOMPRESS Function

Purpose

Replaces multiple consecutive blanks in the specified mixed source string with single blanks.

Definition

```
FUNCTION MCOMPRESS( CONST msource : STRING)
                                  : STRING;
```

Where	Represents
source	a mixed string expression to be compressed.

Single-byte blanks are replaced with one single-byte blank, and MBCS blanks are replaced with one MBCS blank. **MCOMPRESS** manipulates mixed strings in a character-oriented manner, treating single-byte characters and MBCS characters as distinct.

Example

```
s := '   BBBB     BB '
k :=  MCOMPRESS( s ) ; (*  yields ' B B ' *)
```

Note: The **B** represents one MBCS blank.

MDELETE Function

Purpose

Returns the source mixed string with a portion removed.

Definition

```
FUNCTION MDELETE( CONST msource : STRING;
                        start   : INTEGER;
                        len     : INTEGER ) : STRING;

FUNCTION MDELETE( CONST msource : STRING;
                        start   : INTEGER ) : STRING;
```

Where	Represents
msource	a mixed string expression from which a portion will be deleted
start	an **INTEGER** expression that specifies the starting position within the source where characters are to be deleted
len	an optional **INTEGER** expression that specifies the number of characters to be deleted.

The first character of the source string is at position 1. If the length is omitted, all remaining characters are deleted. The string is truncated beginning at position *start*.

The following conditions must exist to avoid an error message at run time:

- *start* must be greater than 0
- *len* must be greater than or equal to 0
- *start*+*len*–1 must be less than or equal to the current length of the string.

If *len* is 0, the whole string is returned.

MDELETE manipulates mixed strings in a character-oriented manner, treating single-byte characters and MBCS characters as distinct.

Examples

```
k := MDELETE( 'aBDd', 1, 2 ) ; (*   yields 'Dd'  *)
k := MDELETE( 'DBD', 2, 1 ) ;  (*  yields 'DD'  *)
```

Note: The **D** represents one MBCS character, and **B** represents one MBCS blank.

MIN Function

Purpose

Returns the minimum value of one or more expressions.

Definition

```
FUNCTION MIN( expr,expr.. : scalar-type)
                          : scalar-type;
```

Where	Represents
expr	a scalar expression, including **REAL** and **SHORTREAL**.

The parameters for **MAX** can be a mixture of **INTEGER**, **REAL**, and **SHORTREAL** expressions. If the parameters are mixed and one of them is a **REAL**, a **REAL** value is returned. If the parameters are mixed and do not include a **REAL**, but do include a **SHORTREAL**, a **SHORTREAL** value is returned.

MINDEX Function

Purpose

Returns the starting index of the first instance of the second mixed string within the first mixed string. If the second mixed string does not exist in the first, **MINDEX** returns zero. If the second mixed string is null, it returns a 1.

Definition

```
FUNCTION MINDEX( CONST msource : STRING ;
                 CONST mlookup  : STRING )
                                : 0..32767;
```

Where	Represents
msource	a mixed string expression containing the data to be compared
mlookup	the data to be compared to *msource*.

MINDEX manipulates mixed strings in a character-oriented manner, treating single-byte characters and MBCS characters as distinct.

Examples

```
k := MINDEX( 'DB', 'D' ) ;  (*  yields 1               *)
k := MINDEX( 'DB', 'a' ) ;  (* yields 0 because a      *)
                           (* single-byte 'a' is not  *)
                           (* the same as an MBCS 'D'  *)
```

Note: The **D** represents one MBCS character, and **B** represents one MBCS blank.

MLENGTH Function

Purpose

Returns the number of characters in a mixed string.

Definition

```
  FUNCTION MLENGTH( expr : STRING )
                         : INTEGER;
```

Where	Represents
expr	a mixed string parameter.

The **MLENGTH** function manipulates mixed strings in a character-oriented manner, treating single-byte characters and MBCS characters as distinct.

Example

```
  k := MLENGTH( 'aDDd' ) ;   (* yields 4 *)
```

Note: The **D** represents one MBCS character.

MLTRIM Function

Purpose

Returns the value of a specified mixed string with all leading blanks removed.

Definition

```
  FUNCTION MLTRIM( CONST msource : STRING )
                                 : STRING;
```

Where	Represents
msource	the mixed string expression to be trimmed.

The **MLTRIM** function manipulates mixed strings in a character-oriented manner, treating single-byte characters and MBCS characters as distinct.

Examples

```
  k := MLTRIM( ' BB ' ) ;      (* yields '' *)
  k := MLTRIM( ' BBabD' ) ;  (* yields 'abD' *)
```

MRINDEX Function

The **MRINDEX** function returns the starting index of the last instance of the second mixed string within the first. If the second mixed string does not exist in the first mixed string, **MRINDEX** returns a zero. If the second mixed string is null, it returns **MLENGTH**(*s*)+1.

Definition

```
  FUNCTION MRINDEX( CONST msource : STRING;
                    CONST mlookup : STRING )
                                  : 0..32767;
```

Where	Represents
msource	a mixed string expression containing the data to be compared
mlookup	the data to be compared to *msource*.

MRINDEX manipulates mixed strings in a character-oriented manner, treating single-byte characters and MBCS characters as distinct.

Examples

```
k := MRINDEX( 'DBBD', 'D' ) ;    (* yields 4 *)
k := MRINDEX( 'DBBD', 'a' ) ;    (* yields 0 because a *)
                                  (* single-byte 'a' is *)
                                  (* not the same as *)
                                  (* an MBCS 'D' *)
```

Note: The **D** represents one MBCS character, and **B** represents one MBCS blank.

MSUBSTR Function

Purpose

Returns a specified portion of a mixed string.

Definition

```
FUNCTION  MSUBSTR( CONST msource : STRING;
                          start   : INTEGER;
                          len     : INTEGER) : STRING;

FUNCTION  MSUBSTR( CONST msource : STRING;
                           start : INTEGER) : STRING;
```

Where	Represents
msource	a mixed string expression from which a substring is returned
start	an **INTEGER** expression that specifies the starting position within the source from which the substring is to be extracted
len	an **INTEGER** expression that determines the length of the substring.

The first character of the source string is at position 1. If the length is omitted, the substring returned is the remaining portion of the source string from position *start*.

The following conditions must exist to avoid an error message at run time:

- *start* must be greater than 0
- *len* must be greater than or equal to 0
- *start*+ *len*–1 must be less than or equal to the current length of the string.

If *len* is 0, a null string is returned.

MSUBSTR manipulates mixed strings in a character-oriented manner, treating single-byte characters and MBCS characters as distinct.

Examples

```
k := MSUBSTR('aDDd',2,3) ; (* yields 'DDd' *)
k := MSUBSTR('aDDd',1,3) ; (* yields 'aDD' *)
k := MSUBSTR('aDDd',3) ;    (* yields 'Dd' *)
k := MSUBSTR('aDDd',1) ;    (* yields 'aDDd' *)
k := MSUBSTR('aDDd',5,0) ; (* yields '' *)
k := MSUBSTR('aDDd',2,5) ; (* is an error *)
```

Note: The **D** represents one MBCS character.

MTRIM Function

Purpose

Returns the value of a specified mixed string with all trailing blanks removed.

Definition

```
  FUNCTION MTRIM( CONST msource : STRING )
                                : STRING;
```

Where	**Represents**
msource	the mixed string expression to be trimmed.

MTRIM manipulates mixed strings in a character-oriented manner, treating single-byte characters and MBCS characters as distinct.

Examples

```
  k := MTRIM(' BB ') ;       (* yields '' *)
  k := MTRIM('abD BB') ;   (* yields 'abD' *)
```

Note: The **D** represents one MBCS character, and **B** represents one MBCS blank.

NEW Procedure

Purpose

Allocates storage for a dynamic variable in the current heap and sets the pointer to point to the dynamic variable.

Definition

```
  Form 1:
   PROCEDURE  NEW( VAR p1  : POINTER );

  Form 2:
   PROCEDURE  NEW( VAR p2  : POINTER;
                   t1,t2... : ordinal-type);

  Form 3:
   PROCEDURE  NEW( VAR p3  : STRINGPTR;
                       len : INTEGER );

   PROCEDURE  NEW( VAR p3  : GSTRINGPTR;
                       len : INTEGER );
```

Where	**Represents**
p1	a pointer to any type
p2	a pointer to a **RECORD** type with variants
t1,t2	ordinal constants representing tag fields
p3	a **STRINGPTR** or a **GSTRINGPTR** (VS mode only)
l*en*	an **INTEGER** expression (VS mode only)
Form 1	allocates the amount of storage necessary to represent a value of the type to which the pointer refers. If the type of the dynamic variable is a record with a variant part, the space allocated is the amount required for the record when the largest variant is active.

Form 2 allocates a variant record when it is known which variant (and subvariants) is active. The amount of storage allocated is only what is necessary to contain the variants specified. The scalar constants are tag field values. The first one indicates a particular variant in the record that is active; subsequent tags indicate active subvariants.

Note: *This procedure does not set tag fields*. The tag list only indicates the amount of storage required; you must set the tag fields after the record is allocated.

Form 3 allocates a string whose maximum length is known only at run time. It is available only in VS mode. The amount of storage made available for the string is defined by the required second parameter. See "STRINGPTR" on page 85 and "GSTRINGPTR" on page 67 for more information.

Examples

The following example shows Form 1 of the **NEW** procedure.

```
TYPE
    linkp = @link;
    link = RECORD
              name : STRING( 30 );
              next : linkp
           END;

 VAR
    p, head : linkp;
    .
    .
 BEGIN
    .
    .
    NEW( p );
    WITH p@ DO
       BEGIN
          name := '';
          next := head ;
       END  ;
    head := p  ;
    .
    .
 END;
```

In the following example, Form 2 of the **NEW** procedure is used to allocate records with variants.

```
TYPE
    age = 0..100;
    recp = @rec;
    rec = RECORD
             name : STRING( 30 );
             CASE how_old : age OF
                0..18 :
                ( father : recp );
                19..100 :
                ( CASE married : BOOLEAN OF
                     TRUE  : ( spouse : recp ) ;
                     FALSE : ()                 )
          END ;

VAR
    p, father, spouse : recp;
    .
    .
BEGIN
    .
    .
    NEW( p, 18 );                        (* First call to NEW  *)
    WITH p@ DO
       BEGIN
          name    := 'J. B. SMITH, JR' ;
          how_old := 18 ;
          NEW( father, 54, TRUE );       (* Second call to NEW *)
          WITH father@ DO
             BEGIN
                name    := 'J. B. SMITH';
                how_old := 54;
                married := TRUE;
                NEW( spouse, 50, TRUE ); (* Third call to NEW  *)
                .
                .
             END (* with father@ *) ;
       END (* with p@ *) ;
       .
       .
END;
```

NEWHEAP Procedure

Purpose

Creates a new heap.

Definition

```
PROCEDURE NEWHEAP( VAR p : pointer );

PROCEDURE NEWHEAP( VAR p : pointer; CONST s : string);
```

Where	Represents
p	a pointer to any type
s	an optional string. This string is ignored by XL Pascal. It is used only for compatibility with VS Pascal.

NEWHEAP makes the pointer passed to **NEWHEAP** a heap-id for the new heap. The heap is initially empty. It will contain all of the dynamic variables created by **NEW** while the heap is established as the currently active heap by the **USEHEAP** procedure.

To avoid unpredictable results, you should not use a pointer that is a heap-id set by **NEWHEAP** as the base of a dynamic variable.

ODD Function

Purpose

Returns **TRUE** if the **INTEGER** value is odd, or **FALSE** if it is even.

Definition

```
FUNCTION ODD( i : INTEGER )
              : BOOLEAN;
```

Where	Represents
i	an **INTEGER** expression.

ORD Function

Purpose

Returns an integer that corresponds to an ordinal value. The **ORD** function also works with pointers.

Definition

```
FUNCTION ORD( s : ordinal-type)
              : INTEGER;

FUNCTION ORD( s : pointer)
              : INTEGER;
```

Where	Represents
s	any ordinal type. In VS mode, *s* can also be a pointer expression.

If the operand is of type **CHAR**, the value returned is the position in the ASCII character set for the character operand. See the *User's Guide for IBM AIX XL Pascal Compiler/6000* for a table of the ASCII character set and more information on the ASCII values and corresponding characters. If the operand is an enumerated scalar, **ORD** returns the position in the enumeration (beginning at zero). For example, in the following declaration:

```
COLOR = (RED, YELLOW, BLUE)
```

`ORD(RED)` is 0 and `ORD(BLUE)` is 2. If the operand is a pointer, the function returns the machine address of the dynamic variable referenced by the pointer.

Note: Although pointers can be converted to integers, XL Pascal provides no function to convert an integer to a pointer.

Ordinal Conversions

Purpose

The definition of any type identifier that specifies an ordinal type (scalars or subranges) forms an ordinal conversion function. The ordinal conversion functions convert an **INTEGER** into a specified ordinal type. Ordinal conversion is the opposite of the XL Pascal predefined function **ORD**, which converts any ordinal value into a 32-bit **INTEGER**.

Definition

```
FUNCTION  id-type( i : INTEGER )
                     : ordinal-type;
```

Where	Represents
id–type	type identifier for any scalar including reals
i	an expression with an **INTEGER** value to be converted to the type of *id–type*.

Syntax

```
── id_type ── (── expression ── )  ─┤
```

Description

An integer expression is converted to another ordinal type by enclosing the expression in parentheses and prefixing it with the type identifier of the desired ordinal type. The conversion is performed in such a way as to be the inverse of the **ORD** function. If the operand is not in the following range, a subrange error exists:

```
ORD ( LOWEST (ordinal type ) )..ORD ( HIGHEST (ordinal type ) )
```

Equivalent Expressions

The following XL Pascal expressions are equivalent by definition:

Expression	Is Equivalent to
CHAR(*x*)	**CHR**(*x*)
INTEGER(*x*)	*x*
ORD(type(*x*))	*x*
REAL(*x*)	**FLOAT**(*x*) for numbers of type **REAL** or **SHORTREAL**

Examples

```
TYPE
    week = ( sun, mon, tue, wed, thu, fri, sat );

VAR
    day : week;
    .
    .
    day := week( 6 );  (* assigns sat to day    *)
    day := week( 0 );  (* assigns sun to day    *)
    day := week( 3 );  (* assigns wed to day    *)
    day := week( 7 );  (* is a compile-time error  *)
```

PACK Procedure

Purpose

Copies elements from an unpacked source array to a packed target array. Copying starts with the specified element of the source array. The types of the elements of the two arrays must be identical. In VS mode, if the array elements are of subrange type, they need only have identical bounds.

Definition

```
PROCEDURE  PACK( CONST source : array-type;
                       index  : index-of-source;
                 VAR   target : pack-array-type);
```

Where	Represents
source	an array
index	an expression compatible with the index of *source*
target	a packed array variable

Example

The following example shows the use of the **PACK** procedure, and equivalent code that performs the same task without the predefined routine. An error results if the number of elements in array `z` is greater than the number of elements used in array `a`.

Assuming the following declarations:

```
a : ARRAY [m..n] OF t;
z : PACKED ARRAY [u..v] OF t;
```

the example

```
PACK( a, i, z );
```

is equivalent to

```
k := i;
FOR j := LBOUND( z ) TO HBOUND( z ) DO
    (* j and k are temporary variables *)
    BEGIN
       z[j] := a[k];
       IF j <> HBOUND( z )
          THEN k := SUCC( k );
    END;
```

PAGE Procedure

Purpose

Causes a skip to the top of the next page when the **TEXT** file is printed.

Definition

```
PROCEDURE  PAGE( VAR f : TEXT );
```

Where	Represents
f	an optional **TEXT** file variable. The default is the predefined file **OUTPUT**.

PARMS Function

Purpose

Returns a string associated with the initial call to the XL Pascal main program.

Definition

```
FUNCTION  PARMS : STRING;
```

PICTURE Function

Purpose

Formats a floating-point value according to a picture format.

Definition

```
FUNCTION  PICTURE( CONST p : STRING;
                         r : REAL ): STRING( 100 );
```

Where	**Represents**
p	a picture specification
r	the **REAL** number to be formatted

The **PICTURE** function returns the string representation of a **REAL** number formatted according to a picture specification. The characters that make up the picture specification are similar to those found in PL/I and COBOL. A picture specification consists of two fields: a decimal field and an exponent field. The latter is optional, but the decimal field is always required. The decimal field can consist of two subfields: the integer part and the fractional part. The latter is optional, but the integer part is always required.

Picture characters can be specified in lowercase. A picture character can be grouped into the following categories:

- Digit and decimal-point specifiers
- Zero suppression characters
- Insertion characters
- Mathematical signs and the dollar symbol
- Exponent specifiers

Digit and decimal-point specifiers:

9 specifies that the associated position in the data item is to contain a decimal digit.

V divides the decimal field into two parts: the integer part and the fractional part. This character specifies that a decimal point is to be assumed at this position in the associated data item.

Note: **V** does not specify that an actual decimal point is to be inserted. The integer and fractional parts of the assigned value are aligned on the **V** character. An assigned value can be truncated or extended with zero digits at either end. If no **V** character appears, a **V** is assumed at the right end of the decimal field.

Zero suppression characters:

Z specifies a conditional digit position in the character string value and may cause a leading zero to be replaced with a blank.

***** specifies a conditional digit position in the character string value and may cause a leading zero to be replaced with an asterisk (*).

Leading zeros occur in the leftmost digit positions of the integer part of floating-point numbers.

Insertion characters:

Insertion characters are added into corresponding positions in the output string provided that zero suppression is not taking place. If zeros are being suppressed when an insertion character is encountered, a blank or an asterisk is inserted in the corresponding place in the output string, depending on whether the zero-suppression character is a **Z** or an asterisk (*).

,	causes a comma to be inserted into the associated position of the output string.
.	causes a point (.) to be inserted into the associated position of the output string. The character never causes point alignment in the number; that function is served solely by the character **V**.
B	causes a blank to be inserted into the associated position of the output string.

Mathematical signs and the dollar symbol:

Mathematical sign and the dollar symbol (**S**, **+**, **–**, **$**) can be used in either a static or a drifting manner. The static use specifies that a sign, a dollar symbol, or a blank always appears in the associated position. The drifting use specifies that leading zeros are to be suppressed.

A drifting character is specified by multiple use of that character in a picture field.

+	specifies a plus sign character (+) if the number is >=0, if the number is <0, it specifies a blank.
–	specifies a minus sign character (–) if the number is <0, if the number is >=0, it specifies a blank.
S	specifies a plus sign character (+) if the number is >=0, if the number is <0, it specifies a minus sign character (–).
$	specifies a dollar sign character ($).

Exponent specifiers:

The characters **E** and **K** delimit the exponent field of a picture specification. The exponent field must always be the last field.

E	specifies that the associated position contains the letter **E**, which indicates the start of the exponent field.
K	specifies that the exponent field appears to the right of the associated position. It does not specify a character data item.

Examples

In the following table, the first column shows the picture specification format `P`, the second column shows the real numbers to be formatted (`R`), and the third column shows the results of `PICTURE(P,R)`.

Picture (P)	**Real Number (R)**	**PICTURE (P,R)**
'99999'	123.0	'00123'
'ZZZZ9'	123.0	' 123'
'****9'	123.0	'**123'
'ZZZZ9'	0.0	' 0'
'ZZZZZ'	0.0	' '
'****9'	0.0	'****0'
'*****'	0.0	'*****'
'S9999'	123.0	'+0123'
'+9999'	123.0	'+0123'
'+9999'	-123.0	' 0123'
'999.99'	-123.456	'001.23'
'999V.99'	123.456	'123.46'
'ZZZ,ZZZ,ZZ9'	123456.0	' 123,456'
'***,***,**9'	123456.0	'****123,456'
'-ZZ,ZZZ,ZZ9'	-123456.0	'- 123,456'
'---,---,--9'	-123456.0	' -123,456'
'$**,***,**9V.99'	123456.78	'$***123,456.78'
'$$$,$$$,$$9V.99'	123456.78	' $123,456.78'
'S9V.9999ES99'	1.23456	'+1.2346E+00'
'S9V.9999KS99'	1.23456	'+1.2346+00'
'-999.999,V99'	1234.567	' 001.234,57'
'-9.999E9'	-1234.567	'-1.235E0'
'9B9B9B9B9B9'	123456.0	'1 2 3 4 5 6'
'9.9.9.9.9.9'	12345.0	'0.1.2.3.4.5'
'999999S'	-12345.0	'12345-'
'999+'	-123.45	'123 '
'999+'	+123.45	'123+'
'ZZZ.V99'	0.12	' 12'
'ZZZV.99'	0.12	' .12'
'-9V.999ES9'	1.23E4	' 1.230E+4'
'S9999VESZ9'	-123456.0	'-1235E+ 2'
'-V.999E-99'	123456.0	' .123E 06'

PRED Function

Purpose

Returns the predecessor value of the parameter expression.

Definition

```
FUNCTION PRED( s : ordinal-type )
                 : ordinal-type;
```

Where	**Represents**
s	any ordinal expression.

The **PRED** of the first element of an ordinal type is an error. `PRED(TRUE)` is **FALSE** and `PRED('B')` is `'A'`. The **PRED** of an integer is equivalent to subtracting one from the value of the integer. **PRED** of a **REAL** argument is an error.

Example

```
TYPE
    nephews = ( huey, duey, louie );

    k := PRED( duey ) ;  (*   yields huey    *)
    k := PRED( huey ) ;  (*   is an error    *)
    k := PRED( TRUE ) ;  (*   yields FALSE   *)
    k := PRED( 'b' ) ;   (*   yields 'a'     *)
    k := PRED( i ) ;     (*   yields i-1     *)
    k := PRED( 3.0 ) ;   (*   is an error    *)
```

PUT Procedure

Purpose

Writes the contents of the file pointer into the specified file, and positions a file pointer to its next element. The file must have been previously opened for output.

Definition

```
PROCEDURE PUT( VAR  f : filetype );
```

Where	Represents
f	a file variable

In VS mode, the open is implicit for **TEXT** files.

Note: The **PUT** procedure cannot write MBCS data to a **TEXT** file.

QUERYHEAP Procedure

Purpose

Gets the heap-id of the currently active heap.

Definition

```
PROCEDURE QUERYHEAP( VAR p : pointer );
```

Where	Represents
p	a pointer to any type

QUERYHEAP sets its pointer argument to the heap-id of the currently active heap. The currently active heap can be either the default heap or a heap created by **NEWHEAP** and then established as the current heap by **USEHEAP**.

To avoid unpredictable results, you should not use a pointer that is a heap-id set by **QUERYHEAP** as the base of a dynamic variable.

RANDOM Function

Purpose

Returns a pseudorandom **REAL** value in the range >0.0 and <1.0.

Definition

```
FUNCTION  RANDOM( s : INTEGER ) : REAL;
```

Where	Represents
s	an expression evaluated to an **INTEGER** value

The parameter *s* is called the *seed* of the random number and specifies the beginning of the sequence. **RANDOM** always returns the same value when called with the same nonzero seed. If you pass a seed value of 0, **RANDOM** returns the next number as generated from the previous seed. Thus, the general way to use this function is to pass it a nonzero seed on the first invocation and a zero value thereafter.

READ Procedure (for RECORD Files)

Purpose

Reads data from **RECORD** files. Each call to **READ** reads one record from the specified file into each variable in the call.

Definition

```
PROCEDURE READ( VAR f          : FILE OF t;
                VAR v1, v2... : t);
```

Where	**Represents**
f	a **RECORD** file variable
v	a list of variables whose type matches the file component type of *f*

You can specify more than one variable on each call by separating each variable with a comma. The effect is the same as multiple calls to **READ**.

Examples

The following example shows the **READ** procedure used for record files.

```
READ( f, v );
```

is equivalent to

```
v := f@;
GET( f );
```

The following example shows multiple variables on **READ**.

```
READ( f, v1, v2 );
```

is equivalent to

```
READ( f, v1 );
READ( f, v2 );
```

Related Information

See the *User's Guide for IBM AIX XL Pascal Compiler/6000* for more information on **READ** for record files.

READ and READLN Procedures (for TEXT Files)

Purpose

Read data from a **TEXT** file. The **READ** procedure reads character data from a **TEXT** file and converts character data to conform to the type of the operand. The file parameter is optional; the default file is **INPUT**.

The **READLN** procedure reads in data (if any variables are specified) the same way **READ** operates, and then moves the file pointer to the beginning of the next line. For **TEXT** files opened with the **INTERACTIVE** attribute (which is turned on by default using the **TERMIN** procedure or for the input identifier when **–qlanglvl=vs**), the file pointer is positioned after the end of the logical record, and the end-of-line condition is set to **TRUE**.

Definition

```
PROCEDURE READ( VAR  f   : TEXT;
                     v   : see below );

PROCEDURE READLN( VAR  f : TEXT;
                       v : see below );

PROCEDURE READLN( VAR  f : TEXT );
(* resets file pointer to first character of the next file row *)
```

Where	Represents
f	an optional **TEXT** file open for reading. The default is the predefined file **INPUT**.
v	a list of variables (optional for **READLN**):

CHAR (or subrange)
MBCS fixed string (**PACKED ARRAY** [1..*n*] **OF GCHAR**)
Fixed string (**PACKED ARRAY** [1..*n*] **OF CHAR**)
GCHAR
GSTRING
INTEGER (or subrange)
REAL
SHORTREAL
STRING

Because **TEXT** files not currently open for **WRITE** are implicitly opened for **READ**, a call to reset a **TEXT** file is not necessary before a call to **READ** or **READLN**.

You can specify more than one variable on each call by separating the variables with a comma, with input data read from left to right. The effect is the same as multiple calls to **READ**.

The following example shows multiple variables on **READ**.

```
READ( f, v1, v2 );
```

is equivalent to

```
READ( f, v1 );
READ( f, v2 );
```

The following example shows multiple variables on **READLN**.

```
READLN( f, v1, v2, v3 );
```

is equivalent to

```
READ( f, v1 );
READ( f, v2 );
READ( f, v3 );
READLN( f );
```

Reading Variables with a Length

You can qualify a **READ** variable with a field length expression such as

```
READ( f, v : n )
```

where `v` is the variable being read and `n` is the field length expression.

This expression denotes the number of characters in the input line to be processed for the variable `v`. If the number of characters indicated by the field length is exhausted during a read operation, the reading operation stops and a subsequent read operation begins at the first character following the field. If the reading is completed before processing all characters of the field, the rest of the field is skipped.

In the following example, field lengths in the call to **READLN** are specified for the integer and character variables.

Given the following data:

```
36 24 abcdefghiklmnopqrstuvwxyz
```

with this declaration:

```
VAR
    i,j : INTEGER;
    s   : STRING( 100 );
    ch  : CHAR;
    cc  : PACKED ARRAY [1..10] OF CHAR;
    f   : TEXT;
    .
    .
READLN( f, i : 4, j : 10, ch : j, cc, s );
```

the variables would be assigned

```
i               36
j               4
ch              'I'
cc              'nopqrstuvw'
s               'xyz'
LENGTH ( s )    3
```

Reading CHAR Data

The next character in the file is assigned to a variable of type **CHAR**.

Reading MBCS Fixed String Data

If the variable is declared as an MBCS fixed string (**PACKED ARRAY** [1..*n*] **OF GCHAR**), multibyte characters are stored into each element of the array. This is equivalent to performing a read operation for each element using a loop ranging from the lower bound of the array to the upper bound. If the end-of-line condition becomes true before the variable is filled, the rest of the variable is filled with blanks. Reading MBCS fixed string data causes the data to be converted from file code format to process code format.

Reading Fixed String Data

If the variable is declared as a fixed string (**PACKED ARRAY** [1..*n*] **OF CHAR**), characters are stored into each element of the array. This is equivalent to a loop ranging from the lower bound of the array to the upper bound performing a read operation for each element. If the end-of-line condition should become true before the variable is filled, the rest of the variable is filled with blanks.

Reading GCHAR Data

The next multibyte character in the file is assigned to a variable of type **GCHAR**. Reading **GCHAR** data converts it from file code format to process code format.

Reading GSTRING Data

Multibyte characters are read into a **GSTRING** variable until the variable has reached its maximum length, or until the end of the line is reached. Reading **GSTRING** data converts it from file code format to process code format.

Reading INTEGER Data

The **INTEGER** data is read by skipping leading blanks and end of lines, and reading an optional sign followed by one or more numeric characters until a nonnumeric character is found. If the characters read do not form a valid **INTEGER**, a runtime error occurs.

Reading REAL and SHORTREAL Data

REAL data is read by skipping leading blanks and end of lines. Characters are then read until a character is found that is not a **REAL** number. If the characters read do not form a valid **REAL** number, a runtime error occurs.

In VS mode, **SHORTREAL** data is read in the same way that **REAL** data is read.

Reading STRING Data

Characters are read into a **STRING** variable until the variable has reached its maximum length, or until the end of the line is reached.

Example

Given the following data:

```
36 24 abcdefghijklmnopqrstuvwxyz
```

with this declaration:

```
VAR
    i, j : INTEGER;
    s  : STRING( 100 );
    ch : CHAR;
    cc : PACKED ARRAY [1..10] OF CHAR;
    f  : TEXT;
    .
    .
    READLN( f, i, j, ch, cc, s );
```

the variables would be assigned

```
i           36
j           24
ch          ' '
cc          'abcdefghij'
s           'klmnopqrstuvwxyz'
LENGTH( s ) 16
```

Related Information

See the *User's Guide for IBM AIX XL Pascal Compiler/6000* for more information on **READ** and **READLN**.

READSTR Procedure

Purpose

Reads character data from a source string into one or more variables.

Definition

```
PROCEDURE READSTR( CONST s : STRING;
                   VAR   v : see below );
```

Where	Represents
s	a string expression to be used for input
v	a list of one or more variables; each must be one of the following types:

CHAR (or subrange)
MBCS fixed string (**PACKED ARRAY** [1..*n*] **OF GCHAR**)
Fixed string (**PACKED ARRAY** [1..*n*] **OF CHAR**)
GCHAR
GSTRING
INTEGER (or subrange)
REAL
SHORTREAL
STRING

The source string can be a variable or a constant string value. The actions of **READSTR** are identical to that of **READ** except that the source data is extracted from a string expression instead of a text file. **READSTR** is especially useful for converting a string to a different type.

The compiler regards the source string for **READSTR** as a virtual line. When it is totally empty or when the end of the line is reached, a virtual end-of-line exists.

The same principles apply to arguments of type **STRING** and **PACKED ARRAY OF CHAR**. Strings are assigned a null string at end-of-line, and packed arrays of **CHAR** get filled with blanks at end-of-line.

Examples

As with the **READ** procedure, variables can be qualified with a field length expression. This is shown in the following example.

With this declaration:

```
VAR
    i  , j : INTEGER;
    s  : STRING( 100 );
    s1 : STRING( 100 );
    ch : CHAR;
    cc : PACKED ARRAY [1..10] OF CHAR;
    .
    .
    s := '36 245abcdefghijk';
    READSTR( s, i, j : 3, ch, cc : 5, s1 );
```

the variables would be assigned

```
i                36
j                24
ch               '5'
cc               'abcde     '
s1               'fghijk'
LENGTH( s1 )     6
```

The following example shows code that has the same effect as **READSTR**.

```
READSTR( s, v1, v2 );
```

has the same effect as

```
REWRITE( f );
WRITE( f, s );
RESET( f );
READ( f, v1, v2 );
```

In the following example, the **READSTR** procedure on the empty source string `s` assigns a blank to `c1`.

```
VAR
    c1, c2 : CHAR;
    .
    .
    s : = '';
    READSTR( s, c1 );
```

In the following example, however, a runtime error message is generated when the system attempts to read into `c2`:

```
VAR
    c1, c2 : CHAR;
    .
    .
    s : = '';
    READSTR( s, c1, c2 );
```

Related Information

The **READ** procedure is described on page 177.

RELEASE Procedure

Purpose

Frees one or more subheaps previously established by calls to **MARK**. The parameter of **RELEASE** must contain the heap control value returned in the pointer parameter of a previous call to **MARK**.

Definition

```
PROCEDURE RELEASE( VAR p : pointer );
```

Where	Represents
p	a pointer returned from a call to **MARK**

Subheaps are created and cleared in a stack-like manner within a heap. **RELEASE** frees all subheaps established in the heap since the corresponding **MARK** was processed. The subheap freed by **RELEASE** can be in the currently active heap or a different heap.

When you free a heap, all of the dynamic variables allocated in the heap are also freed. As a result, **RELEASE** is a means for disposing of many dynamic variables at one time. Pointers that reference dynamic variables of a heap are undefined when the heap is freed. Using these pointer values later may cause unpredictable results.

RELEASE sets its parameter variable *p* to **NIL** if it is a valid destination for assignment.

Example of the RELEASE and MARK Procedures

```
TYPE
    markp = @INTEGER;
    linkp = @link;
    link  = RECORD
               name : STRING( 30 );
               next : linkp
            END;

VAR
    p : markp;
    q1, q2, q3 : linkp;

BEGIN
    .
    .
    MARK( p );
    .
    .
    NEW( q1 );
    NEW( q2 );
    NEW( q3 );
    .
    .
    RELEASE( p );          (* Frees q1, q2 and q3 *)
    .
    .
END;
```

RESET Procedure

Purpose

Opens a file for input, positions the file pointer to the beginning of the file, and prepares the file to be used for input. After you call **RESET**, the file pointer points to the first data element of the file. When **RESET** cannot locate the specified file, processing stops and a runtime error message is issued.

In VS mode, a call to **RESET** a **TEXT** file is not necessary before a call to **READ** or **READLN** unless the file is open for output.

Definition

```
PROCEDURE RESET( VAR   f : filetype;
                 CONST s : STRING);
```

Where	**Represents**
f	a file variable
s	an optional string of file-dependent options to be used in opening the file (VS mode only)

Related Information

File opening options are described in the *User’s Guide for IBM AIX XL Pascal Compiler/6000*.

RETCODE Procedure

Purpose

Sets the program completion code, which is the return code passed to the caller of the XL Pascal program.

Definition

```
PROCEDURE RETCODE( retvalue : INTEGER );
```

Where	Represents
retvalue	an **INTEGER** value in the range 0..32767

The value of the operand is returned to the system when an exit is made from the main program. If this routine is called several times, only the last value specified is passed back to the system.

Note: Passing a negative value to **RETCODE** has unpredictable results.

REWRITE Procedure

Purpose

Opens a file for output, positions the file pointer to the beginning of the file, and prepares the file to receive output. **REWRITE** erases the contents of the referenced file unless you request an open option that preserves the existing data (for example, `DISP = MOD` in VS mode).

In VS mode, a call to **REWRITE** a **TEXT** file is not necessary before a **WRITE** or **WRITELN** call, unless the file is open for input.

Definition

```
PROCEDURE REWRITE( VAR   f : filetype;
                   CONST s : STRING);
```

Where	Represents
f	a file variable
s	an optional string of file-dependent options to be used in opening the file (VS mode only)

Related Information

File opening options are described in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

RINDEX Function

Purpose

Returns the starting index of the last instance of the second parameter within the first parameter. If the second parameter does not exist in the first parameter, **RINDEX** returns a zero. If the second parameter is null, **RINDEX** returns **LENGTH**(*s*)+1.

Definition

```
FUNCTION RINDEX( CONST source : STRING ;
                 CONST lookup : STRING )
                              : 0..32767;

FUNCTION RINDEX( CONST source : GSTRING ;
                 CONST lookup : GSTRING )
                              : 0..16382;
```

Where	Represents
source	a **STRING** or **GSTRING** expression to which *lookup* is compared
lookup	the **STRING** or **GSTRING** expression to be compared to *source*

Note: The **RINDEX** function works best with pure single-byte character strings or pure MBCS strings. The predefined routine **MRINDEX** is recommended for operating on mixed strings.

Example

```
VAR
    s  : STRING( 10 );
    .
    .
    s := 'abcabcabc';
    .
    .
    k := RINDEX( s, 'bc' ) ;  (* yields  8   *)
    k := RINDEX( s, 'x' ) ;   (* yields  0   *)
```

ROUND Function

Purpose

Converts a real value to an integer value by rounding the operand. XL Pascal uses the rounding algorithm defined in the ANSI–83 Pascal standard. `ROUND(r)` is equivalent to:

```
IF r > 0.0 THEN
    ROUND := TRUNC( r + 0.5 )
ELSE
    ROUND := TRUNC( r - 0.5 )
```

Definition

```
FUNCTION ROUND( r : REAL )
                  : INTEGER;

FUNCTION ROUND( s : SHORTREAL )
                  : INTEGER;
```

Where	Represents
r	a **REAL** expression
s	a **SHORTREAL** expression (VS mode only)

Note: **INTEGER** arguments are not allowed with the **ROUND** function.

Examples

```
k := ROUND( 1.0) ;    (* is 1  *)
k := ROUND( 1.1) ;    (* is 1  *)
k := ROUND( 1.9) ;    (* is 2  *)
k := ROUND( 0.0) ;    (* is 0  *)
k := ROUND(-1.0) ;    (* is -1 *)
k := ROUND(-1.1) ;    (* is -1 *)
k := ROUND(-1.9) ;    (* is -2 *)
```

RPAD Procedure

Purpose

Pads or truncates a string or **GSTRING** on the right.

Definition

```
PROCEDURE RPAD( VAR s  : STRING;
                    l  : INTEGER;
                    c  : CHAR    );

PROCEDURE RPAD( VAR s  : GSTRING;
                    l  : INTEGER;
                    c  : GCHAR    );
```

Where	Represents
s	the **STRING** or **GSTRING** to be padded
l	the final length of *s*
c	the pad character

If **LENGTH**(*s*) is greater than *l*, **RPAD** truncates characters on the right. If **LENGTH**(*s*) is less than *l*, **RPAD** extends *s* with the character *c* on the right.

Example

```
s := 'abcdef';
k := RPAD( s, 10, '$' ) ;  (* yields 'abcdef$$$$' in s *)
s := 'abcdef';
k := RPAD( s,  5, '$' ) ;     (* yields 'abcde' in s *)
g := 'DDDDDD'G;
k := RPAD ( g, 10, 'B'G ) ; (* yields 'DDDDDDBBBB'G in g *)
g := 'DDDDDD'G;
k := RPAD ( g, 5, 'B'G ) ;  (* yields 'DDDDD'G in g *)
```

SEEK Procedure

Purpose

Positions a file pointer to a specified element. **SEEK** specifies the number of the next file component to be operated on by a **GET** or **PUT** operation. File components have an origin of 1. The **SEEK** procedure is not supported for **TEXT** files. The file specified in the **SEEK** procedure must be opened by **RESET**, **REWRITE**, or **UPDATE**.

The value of the file buffer is undefined after a call to **SEEK**. The **SEEK** procedure does not perform an I/O operation.

Definition

```
PROCEDURE SEEK( VAR  f : filetype;
                     n : INTEGER);
```

Where	Represents
f	a **RECORD** file variable
n	component number of the file

SIN Function

Purpose

Computes the sine of a floating-point number representing an angle in radians.

Definition

```
FUNCTION SIN( x : REAL )
                : REAL;
```

Where	Represents
x	a **REAL** expression

REAL functions will accept **INTEGER** and **SHORTREAL** arguments. See "Type Compatibility" on page 50 for more information.

SIZEOF Function

Purpose

Returns the storage amount in bytes needed to contain a variable of the type specified.

The **SIZEOF** function is useful for block input and output, where you need to specify the number of bytes to be transferred.

Definition

```
FUNCTION SIZEOF( s : any-type)
                   : INTEGER;

FUNCTION SIZEOF( s : record-type;
         t1,t2,... : ordinal-type);
                   : INTEGER;
```

Where	Represents
s	a type or variable identifier
t1,t2	ordinal constants representing tag fields

If the parameter *s* refers to a **RECORD** with a variant part, and if no tag values are specified, the storage required for the record with its largest variant is returned.

If parameter *s* is a record variable or a type identifier of a record, it can be followed by a tag list that defines a particular variant configuration of the record. The function returns the amount of storage required for a record with that variant configuration.

SQR Function

Purpose

Computes the square of a number. The function returns the same type as the argument.

Definition

```
FUNCTION SQR( i : INTEGER )
              : INTEGER;

FUNCTION SQR( r : REAL )
              : REAL;

FUNCTION SQR( s : SHORTREAL )
              : SHORTREAL;
```

Where	Represents
i	an **INTEGER** expression
r	a **REAL** expression
s	a **SHORTREAL** expression (VS mode only)

SQRT Function

Purpose

Computes the square root of a number. If the argument is less than zero, a runtime error message is produced.

Definition

```
FUNCTION SQRT( x : REAL )
               : REAL;
```

Where	Represents
x	a **REAL** expression

REAL functions will accept **INTEGER** and **SHORTREAL** arguments. See “Type Compatibility” on page 50 for more information.

STOGSTR Function

Purpose

Converts a **STRING** to a **GSTRING**, and converts data from file code to process code.

Definition

```
FUNCTION STOGSTR( x : STRING )
                  : GSTRING;
```

Where	Represents
x	a string containing only MBCS characters

Example

```
VAR
    g : GSTRING( 4 );
    s : STRING( 10 );

BEGIN
    s := 'DB'              (* DB is stored in s *)
    g := STOGSTR( s );     (* DB is stored in g *)
    s := 'Dbc'             (* Dbc is stored in s *)
    g := STOGSTR( s );     (* Dbc is stored in g, including the *)
                      (* single-byte characters (since the ASCII *)
                      (* character set is a subset of every MBCS)*)
```

STR Function

Purpose

Converts a **CHAR**, **STRING**, or **PACKED ARRAY OF CHAR** to a **STRING**. If the parameter is a **STRING**, the function is valid but has no effect.

Definition

```
FUNCTION STR( x : CHAR )
                : STRING;

FUNCTION STR( x : PACKED ARRAY [1..n] OF CHAR )
                : STRING;

FUNCTION STR( x : STRING )
                : STRING;
```

Where	**Represents**
x	a **CHAR**, **PACKED ARRAY**[1..*n*] **OF CHAR**, or **STRING**

XL Pascal implicitly converts a literal string to a **CHAR** or **PACKED ARRAY OF CHAR** and a **STRING** type to a **PACKED ARRAY OF CHAR** on assignment. All other conversions require you to explicitly state the conversion.

Given a declaration like this:

```
VAR
    aoc : PACKED ARRAY [1..5] OF CHAR
    ch  : CHAR;
    ...
```

you can assign a **CHAR** to a **PACKED ARRAY OF CHAR** by either of the following:

```
1) aoc := STR( ch );
2) aoc := ' ';
   aoc[1] := ch;
```

Examples

```
VAR
    sc  :  CHAR;
    sa  :  PACKED ARRAY [1..4] OF CHAR;
    s4  :  STRING;

BEGIN
    sc  := 'a';                     (* 'a' is stored in sc  *)
    s4  := STR( sc );               (* 'a' is stored in s4  *)
    sa  := 'ab';                    (* 'ab' is stored in sa *)
    s4  := STR( sa );               (* 'ab' is stored in s4 *)
END;
```

SUBSTR Function

Purpose

Returns a substring from the specified source string.

Definition

```
FUNCTION  SUBSTR( CONST source : STRING;
                        start  : INTEGER;
                        len    : INTEGER) : STRING;

FUNCTION  SUBSTR( CONST source : GSTRING;
                        start  : INTEGER;
                        len    : INTEGER) : GSTRING;
```

Where	**Represents**
source	a **STRING** or **GSTRING** expression from which a substring is returned
start	an **INTEGER** expression that specifies the starting position within the source of the substring
len	an optional **INTEGER** expression that determines the length of the substring

The first character of the source string is at position 1. If the length is omitted, the substring returned will be the remaining portion of the source string from position *start*. The returned string includes the character in the source string at position *start*.

The following conditions must exist to avoid an error message at run time:

- *start* must be greater than 0
- *len* must be greater than or equal to 0
- *start*+ *len*–1 must be less than or equal to the current length of the string.

If *len* is 0, a null string is returned.

Note: **SUBSTR** works best with pure single-byte character strings or pure MBCS strings. The predefined routine **MSUBSTR** is recommended for operating on mixed strings.

Examples

```
k := SUBSTR('abcde',2,3) ;    (* yields 'bcd'   *)
k := SUBSTR('abcde',1,3) ;    (* yields 'abc'   *)
k := SUBSTR('abcde',4) ;      (* yields 'de'    *)
k := SUBSTR('abcde',1) ;      (* yields 'abcde' *)
k := SUBSTR('abcde',2,5) ;    (* is an error    *)
k := SUBSTR('abcde',6,0) ;    (* returns ''     *)
```

SUCC Function

Purpose

Returns the successor value of the parameter expression.

Definition

```
FUNCTION SUCC( s : ordinal-type )
                 : ordinal-type;
```

Where	Represents
s	an ordinal expression

The **SUCC** of the last element of an enumerated scalar is an error. The **SUCC** of an **INTEGER** is equivalent to adding one to the value of the **INTEGER**. Using a **REAL** argument with **SUCC** is an error.

Examples

```
TYPE
    nephews = ( huey, duey, louie );

    k := SUCC( duey ) ;    (* yields louie *)
    k := SUCC( louie ) ;   (* is an error  *)
    k := SUCC( FALSE ) ;   (* yields TRUE  *)
    k := SUCC( 'b' ) ;     (* yields 'c'   *)
    k := SUCC( i ) ;       (* yields i+1   *)
    k := SUCC( 3.0 ) ;     (* is an error  *)
```

TERMIN Procedure

Purpose

Opens a designated file for input from your terminal.

Definition

```
PROCEDURE TERMIN( VAR   f : TEXT;
                  CONST s : STRING);
```

Where	Represents
f	a **TEXT** file variable
s	an optional string of file-dependent options to be used in opening the file

Related Information

File opening options are described in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

TERMOUT Procedure

Purpose

Opens a designated file for output to the terminal.

Definition

```
  PROCEDURE TERMOUT( VAR   f : TEXT;
                     CONST s : STRING );
```

Where	Represents
f	a **TEXT** file variable
s	an optional string of file-dependent options to be used in opening the file

Related Information

File opening options are described in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

TOKEN Procedure

Purpose

Scans its source parameter looking for a token and returns the token in an **ALPHA** array. The **TOKEN** procedure is defined with three parameters.

Definition

```
  PROCEDURE TOKEN ( VAR   pos    : INTEGER;
                    CONST source : STRING;
                    VAR   result : ALPHA   );
```

Where	Represents
pos	an **INTEGER** corresponding to the position in the source string where the search for the token begins. The value of this **INTEGER** is updated to the starting position for subsequent calls to **TOKEN**.
source	a **STRING** expression containing the data from which a token is to be extracted.
result	the resulting token.

The starting position of the scan is the value of the first parameter in the **TOKEN** call. In subsequent calls to **TOKEN**, this parameter is changed to the position at which the scan is to be resumed. When **TOKEN** scans a string, it ignores leading blanks, multiple blanks, and trailing blanks. If no token is in the string, the value of the first parameter is set to **LENGTH**(*source*)+1 and the *result* parameter is set to all blanks.

If the token is longer than **ALPHALEN**, only the first **ALPHALEN** characters are returned but *pos* is updated to point past the entire token and any trailing blanks.

A token can be any of the following:

- An identifier consisting of any number of alphanumeric characters, dollar signs ($), or underscores (_). The first character must be a letter.
- An unsigned number.

- The following special symbols:

```
+     –     *     /     –>    @     ¢
=     <>    <     <=    >=    >     !
(     )     [     ]     ’     ”     %
|     &     &&    ||    ~     ~=    #
:     ;     :=    .     ,     ..
{     }     (*    *)    /*    */
(.    .)    <<    >>
```

- Any other character.

Example

In the following example, **TOKEN** would return the same value if `i` were set to 3; that is, leading blanks are ignored.

```
i := 2;
k := TOKEN( i, ’, Token+’, result ) ;         (* i is set to 8  *)
                                 (* result is set to ’Token        ’ *)
```

TRACE Procedure

Purpose

Displays the list of procedures and functions currently on the invocation stack. Each line of the output shows the following:

- Name of the routine
- Statement number where the call took place
- Return address in hexadecimal format
- Name of the unit with the calling procedure

Definition

```
PROCEDURE TRACE( VAR f : TEXT );
```

Where	**Represents**
f	the **TEXT** file to receive the trace listing

TRIM Function

Purpose

Returns the value of a specified parameter with all trailing blanks removed. It removes trailing MBCS blanks from **GSTRING** data.

Definition

```
FUNCTION TRIM( CONST source : STRING )
                            : STRING;
FUNCTION TRIM( CONST source : GSTRING )
                            : GSTRING;
```

Where	**Represents**
source	the **STRING** or **GSTRING** to be trimmed

Note: The **TRIM** function works best with pure single-byte character strings or pure MBCS strings. The predefined routine **MTRIM** is recommended for operating on mixed strings.

Examples

```
k := TRIM( ’ a b  ’ ) ;  (* yields  ’ a b’  *)
k := TRIM( ’      ’ ) ;  (* yields  ’’      *)
```

TRUNC Function

Purpose

Converts a real expression to an integer expression by rounding the operand toward zero.

Definition

```
FUNCTION TRUNC( r : REAL )
                  : INTEGER;

FUNCTION TRUNC( s : SHORTREAL )
                  : INTEGER;
```

Where	Represents
r	a **REAL** expression
s	a **SHORTREAL** expression (VS mode only)

Examples

```
k := TRUNC( 1.0 ) ;   (* is  1 *)
k := TRUNC( 1.1 ) ;   (* is  1 *)
k := TRUNC( 1.9 ) ;   (* is  1 *)
k := TRUNC( 0.0 ) ;   (* is  0 *)
k := TRUNC( -1.0 ) ;  (* is  -1 *)
k := TRUNC( -1.1 ) ;  (* is  -1 *)
k := TRUNC( -1.9 ) ;  (* is  -1 *)
```

UNPACK Procedure

Purpose

Copies elements from a packed source array to an unpacked target array, starting with the specified element of the target array. The types of the elements of the two arrays must be identical.

In VS mode, if the array elements are of subrange type, they need only have identical bounds.

Definition

```
PROCEDURE UNPACK( CONST source : pack-array-type;
                  VAR   target : array-type;
                        index  : index-of-target );
```

Where	Represents
source	a packed array
target	an array variable
index	an expression compatible with the index of *target*

Example

The following example shows the **UNPACK** procedure and equivalent code that performs the same task without the predefined routine. An error results if the number of elements in `z` is greater than the number of elements used in `a`.

Given the following declarations:

```
a : ARRAY [m..n] OF t;
z : PACKED ARRAY [u..v] OF t;
```

the example

```
UNPACK( z, a, i );
```

is equivalent to

```
k := i;
FOR j := LBOUND( z ) TO HBOUND( z ) DO
     (*  j and k are temporary variables *)
     BEGIN
        a[k] := z[j];
        IF j <> HBOUND( z ) THEN
           k:= SUCC( k );
     END;
```

UPDATE Procedure

Purpose

Opens a designated record for both input and output updating. A **PUT** operation replaces a file component obtained from a preceding **GET** operation. Running **UPDATE** causes an implicit **GET** of the first file component (as in **RESET**). You cannot update a file of type **TEXT**.

Definition

```
PROCEDURE UPDATE( VAR   f : filetype;
                  CONST s : STRING );
```

Where	Represents
f	a **RECORD** file variable
s	an optional string of file-dependent options to be used in opening the file

Example

```
VAR
    filevar : FILE OF RECORD
                 cnt : INTEGER;
                 ...
              END;
    ...

    UPDATE( filevar );                       (* open and get *)
    WHILE NOT EOF( filevar ) DO
       BEGIN
          filevar@.cnt := filevar@.cnt + 1;
          PUT( filevar );             (* update last element *)
          GET ( filevar );            (* get next element    *)
       END;
```

Related Information

File opening options are described in the *User's Guide for IBM AIX XL Pascal Compiler/6000*.

USEHEAP Procedure

Purpose

Establishes a heap as the currently active heap.

Definition

```
PROCEDURE USEHEAP( VAR p : pointer );
```

Where	**Represents**
p	a pointer to any type. Its value must be a heap-id set by a call to **NEWHEAP** or **QUERYHEAP**

The pointer argument *p* must be a heap-id previously set by **NEWHEAP** or **QUERYHEAP**. **USEHEAP** establishes the corresponding heap as the currently active heap. All dynamic variables created by **NEW** are put in the currently active heap, and all mark values or subheap pointers established by **MARK** are in the currently active heap. If **USEHEAP** has not established a current heap, the default heap is active.

WRITE Procedure (for RECORD Files)

Purpose

Writes data to **RECORD** files. Each call to **WRITE** writes the value of each expression in the call to a new record in the specified file.

Definition

```
PROCEDURE WRITE( VAR  f : FILE OF t;
                      e : t );
```

Where	**Represents**
f	a **RECORD** file variable
e	a list of expressions whose types match the file component type of *f*

You can write more than one expression on each call by separating each expression with a comma. The effect is the same as multiple calls to **WRITE**.

Examples

The following example shows the **WRITE** procedure used for record files.

```
WRITE( f, e );
```

is equivalent to

```
f@ := e;
PUT( f );
```

Related Information

See the *User's Guide for IBM AIX XL Pascal Compiler/6000* for more information on **WRITE** for record files.

WRITE and WRITELN Procedures (for TEXT Files)

Purpose

Writes data to a file.

Definition

```
PROCEDURE WRITE( VAR f : TEXT;
                     e : see below );
PROCEDURE WRITELN( VAR f : TEXT;
                       e : see below );
PROCEDURE WRITELN( VAR f : TEXT );
```

Where	Represents
f	an optional **TEXT** file variable. The default is the predefined file **OUTPUT**.
e	a list of expressions (optional for **WRITELN**):

BOOLEAN
CHAR (or subrange)
MBCS fixed string (**PACKED ARRAY** [1..*n*] **OF GCHAR**)
Fixed string (**PACKED ARRAY** [1..*n*] **OF CHAR**)
GCHAR
GSTRING
INTEGER (or subrange)
REAL
SHORTREAL
STRING

The **WRITE** procedure writes character data to a **TEXT** file. The data is obtained by converting the expression to the appropriate output form. The file parameter is optional; if not specified, the default file **OUTPUT** is used. The **WRITELN** procedure writes data (if any expressions are specified) the same way as **WRITE**, and then positions the file to the beginning of the next line.

Because a **TEXT** file is implicitly opened if the file is closed, a call to **REWRITE** a **TEXT** file is not necessary before a call to **WRITE** or **WRITELN**.

You can produce more than one expression on each call by separating each expression with a comma. The effect is the same as multiple calls to **WRITE**. The following example shows multiple expressions on **WRITE**.

```
WRITE( f, e1, e2 );
```

is equivalent to

```
WRITE( f, e1 );
WRITE( f, e2 );
```

The following example shows multiple expressions on **WRITELN**.

```
WRITELN( f, e1, e2, e3 );
```

is equivalent to

```
WRITE( f, e1 );
WRITE( f, e2 );
WRITE( f, e3 );
WRITELN( f );
```

Writing Expressions with a Length

You can control the length of the resulting output to **TEXT** files by specifying actual parameters on **WRITE** and **WRITELN**. Each expression in the **WRITE** procedure call can be represented in one of three forms:

- *e*
- *e : TotalWidth*
- *e : TotalWidth : FracDigits* (used only for **REAL** and **SHORTREAL**)

The expression *e* represents the data to be placed in the file. The data is converted to character representations from its internal form.

The expressions *TotalWidth* and *FracDigits* must be evaluated to an **INTEGER** value. In Standard Pascal, *TotalWidth* and *FracDigits* must be greater than 0.

In VS mode, you can use any integer for *TotalWidth* and *FracDigits*.

Form 1

If *TotalWidth* is unspecified, a default value is used according to these criteria:

Type of Expression e	Default Value of TotalWidth
BOOLEAN	10
CHAR	1
MBCS fixed string	length of array
Fixed string	length of array
GCHAR	1
GSTRING	**LENGTH**(expression)
INTEGER	12
REAL	20 (scientific notation)
SHORTREAL	20 (scientific notation)
STRING	**LENGTH**(expression)

Form 2

The expression *TotalWidth* supplies the length of the field into which the data is written. If *TotalWidth* specifies a positive value, the data is right justified in the field.

In VS mode, if *TotalWidth* specifies a negative value, the data is justified to the left within a field whose length is **ABS**(*TotalWidth*). If *TotalWidth* is 0, the format of the results is unpredictable.

Form 3

You can specify the *FracDigits* expression only if *e* is an expression of type **REAL** or **SHORTREAL**. *FracDigits* controls the number of decimal places that appears in the output.

Writing BOOLEAN Data

The expression *TotalWidth* indicates the length of the field where the Boolean data is to be placed. The number of characters in the field is the value of *TotalWidth*. Boolean data is written exactly the same as the character strings True or False would be (depending on the value of the expression). The data is placed in the field and justified according to the rules stated in “Writing Expressions with a Length” on page 198.

Examples:

Call	Result
`WRITE(TRUE:10)`	`'bbbbbbTRUE'`
`WRITE(TRUE:-10)`	`'TRUEbbbbbb'`
`WRITE(FALSE:2)`	`'FA'`

Note: The `b` represents a blank.

Writing CHAR Data

The value of *TotalWidth* indicates the length of the field where the character is to be placed. If *TotalWidth* is not specified, a field length of 1 is assumed. If *TotalWidth* is greater than 1, the character is padded on the left with blanks. If *TotalWidth* is zero, no data is written.

In VS mode, if *TotalWidth* is negative, and **ABS**(TotalWidth)>1, the character is padded on the right with blanks.

Examples:

Call	**Result**
`WRITE('A':6)`	`'bbbbbA'`
`WRITE('A':-6)`	`'Abbbbb'`

Note: The `b` represents a blank.

Writing MBCS Fixed String Data

If **ABS**(*TotalWidth*) is too small to hold the data, the MBCS string is truncated on the right. If *TotalWidth* is zero, no characters are written.

Examples:

Given the following data:

```
  VAR  a : PACKED ARRAY[ 1..4 ] OF GCHAR ;
       ...
       a := 'DDDD'G;
       ...
```

Call	**Result**
`WRITE(a:6)`	'**BBDDDD**'
`WRITE(a:-6)`	'**DDDDBB**'
`WRITE(a:2)`	'**DD**'
`WRITE(a)`	'**DDDD**'

Note: The **D** represents one MBCS character, and **B** represents one MBCS blank.

Writing Fixed String Data

The expression *TotalWidth* indicates the length of the field where the array is to be placed. The data is placed in the field and justified according to the rules stated in "Writing Expressions with a Length" on page 198. If *TotalWidth* is zero, no data is written. If **ABS**(*TotalWidth*) is too small to hold the data, the string is truncated on the right.

Examples:

Given the following data:

```
  VAR  a : PACKED ARRAY [1..4] OF CHAR;
       ...
       a := 'abcd';
       ...
```

Call	**Result**
`WRITE(a:6)`	`'bbabcd'`
`WRITE(a:-6)`	`'abcdbb'`
`WRITE(a:2)`	`'ab'`
`WRITE(a)`	`'abcd'`

Note: The `b` represents a blank.

Writing GCHAR Data

For **GCHAR** data, the field width is the number of characters written to the file. If the field width is zero, no characters are written.

Examples:

Call	Result
`WRITE('`**D**`'G:6)`	'**BBBBBD**'
`WRITE('`**D**`'G:-6)`	'**DBBBBB**'

Note: The **D** represents one MBCS character, and **B** represents one MBCS blank.

Writing GSTRING Data

For **GSTRING** data, the field width is the number of characters written to the file. If the field width is zero, no characters are written.

Examples:

Call	Result
`WRITE('`**DDDD**`'G:6)`	'**BBDDDD**'
`WRITE('`**DDDD**`'G:-6)`	'**DDDDBB**'
`WRITE('`**DDDD**`'G:2)`	'**DD**'
`WRITE('`**DDDD**`'G)`	'**DDDD**'

Note: The **D** represents one MBCS character, and **B** represents one MBCS blank.

Writing INTEGER Data

The expression *TotalWidth* represents the minimum length of the field where the integer is to be placed. The value is converted to character format and placed in a field of the specified length. If the field is shorter than the size required to represent the value, the length of the field is extended.

Examples:

Call	Result
`WRITE(1234:6)`	`'bb1234'`
`WRITE(1234:-6)`	`'1234bb'`
`WRITE(1234:1)`	`'1234'`
`WRITE(1234)`	`'bbbbbbbb1234'`
`WRITE(1234:-3)`	`'1234'`

Note: The `b` represents a blank.

Writing REAL and SHORTREAL Data

REAL and **SHORTREAL** expressions can be written with any one of the three operand formats, as shown in “Writing Expressions with a Length” on page 198.

Form 1 If *TotalWidth* is not specified, the result is written in scientific notation in a 20 character field.

Form 2 If *TotalWidth* is specified and *FracDigits* is not, the result is written in scientific notation, but the number of characters in the field is the value of *TotalWidth*. One decimal place is always generated and the result is rounded to the last displayed decimal place.

In VS mode, when *TotalWidth* is zero; then the result is unpredictable.

Form 3 If both *TotalWidth* and *FracDigits* are specified, the data is written in fixed point notation in a field with length *TotalWidth*, and *FracDigits* specifies the number of digits that appears to the right of the decimal point. The **REAL** or **SHORTREAL** expression is always rounded to the last digit to be printed.

In VS mode, if *FracDigits* equals zero, a decimal point is written, but no decimal place is written. If *FracDigits* is negative, the number is written using the scientific notation, as if *FracDigits* were not specified. If *TotalWidth* is not large enough to fully represent the number, it is extended appropriately.

Examples:

Call	Result
`WRITE(3.14159)`	`'b3.141590000000E+000'`
`WRITE(3.14159:10)`	`'b3.14E+000'`
`WRITE(3.14159:1)`	`'b3.1E+000'`
`WRITE(3.14159:0)`	`unpredictable`
`WRITE(3.14159:10:4)`	`'bbbb3.1416'`
`WRITE(3.14159:-10:4)`	`'3.1416bbbb'`
`WRITE(3.14159:10:0)`	`'bbbbbbbb3.'`
`WRITE(3.14159:10:-1)`	`'b3.14E+000'`

Note: The `b` represents a blank.

Writing STRING Data

The expression *TotalWidth* indicates the length of the field where the string is to be placed. The data is placed in the field and justified according to the rules stated in "Writing Expressions with a Length" on page 198. If *TotalWidth* is zero, no data is written. If **ABS**(*TotalWidth*) is too small to hold the data, the string is truncated on the right.

Examples:

Call	Result
`WRITE('ABCD':6)`	`'bbABCD'`
`WRITE('ABCD':-6)`	`'ACBDbb'`
`WRITE('ACBD':2)`	`'AB'`
`WRITE('ACBD')`	`'ACBD'`

Note: The `b` represents a blank.

Related Information

See the *User's Guide for IBM AIX XL Pascal Compiler/6000* for more information on **WRITE** and **WRITELN**.

WRITESTR Procedure

Purpose

Converts expressions into character data and stores the data in a **STRING** variable. **WRITESTR** is especially useful for converting data into strings.

Definition

```
PROCEDURE  WRITESTR ( VAR s : STRING;
                          e : see below );
```

Where	Represents
s	a **STRING** variable
e	a list of one or more expressions; each must be one of the following types: **BOOLEAN** **CHAR** (or subrange) MBCS fixed string (**PACKED ARRAY** [1..*n*] **OF GCHAR**) Fixed string (**PACKED ARRAY** [1..*n*] **OF CHAR**) **GCHAR** **GSTRING** **INTEGER** (or subrange) **REAL** **SHORTREAL** **STRING**

The actions of **WRITESTR** are identical to **WRITE**, except that the target of the data is a **STRING** rather than a **TEXT** file. If the *s* variable appears in the *e* expression list of **WRITESTR**, the value of *s* is unpredictable.

As with **WRITE**, the expressions being converted can be qualified with a field length expression.

Examples

With this declaration:

```
VAR
    i, j : INTEGER;
    s    : STRING( 100 );
    r    : REAL;
    ch   : CHAR;
    .
    .

i  := 10;
j  := -123;
r  := 3.14159;
ch := '*';
WRITESTR( s, i : 3, j : 5, 'abc', ch, r : 5 : 2 );
```

the variable s would be assigned as

```
    ' 10 -123abc* 3.14'
```

The following example shows code with the same effect as **WRITESTR**. An error results if all variables are filled before all expressions are written.

```
WRITESTR( s, e1, e2 );
```

has the same effect as

```
REWRITE ( f )
WRITELN( f, e1, e2 );
RESET ( f )
READ( f, s );
```

Related Information

The **WRITE** and **WRITELN** procedures are described on page 197.

xl__trap Procedure

Purpose

Produces a traceback for a runtime trap.

Definition

```
PROCEDURE  xl__trap
```

Note: The name **xl__trap** contains two underscore characters.

Related Information

The *User's Guide for IBM AIX XL Pascal Compiler/6000* shows how to use **xl__trap** to diagnose runtime errors.

Chapter 11. Compiler Directives

The compiler directives of XL Pascal control compiler options and features. The compiler recognizes these directives by the **%** symbol, and does not use the text between the directive and the end of line. The compiler directives are available in VS mode only.

The following directives are accepted by XL Pascal to make migrating programs from VS Pascal easier, but are not used by the compiler. They are printed in the listing when they are found in the source code but otherwise have no effect. See the *VS Pascal Language Reference* for details about their use.

- **%CPAGE**
- **%PAGE**
- **%PRINT**
- **%SKIP**
- **%SPACE**
- **%TITLE**

This chapter describes the following compiler directives:

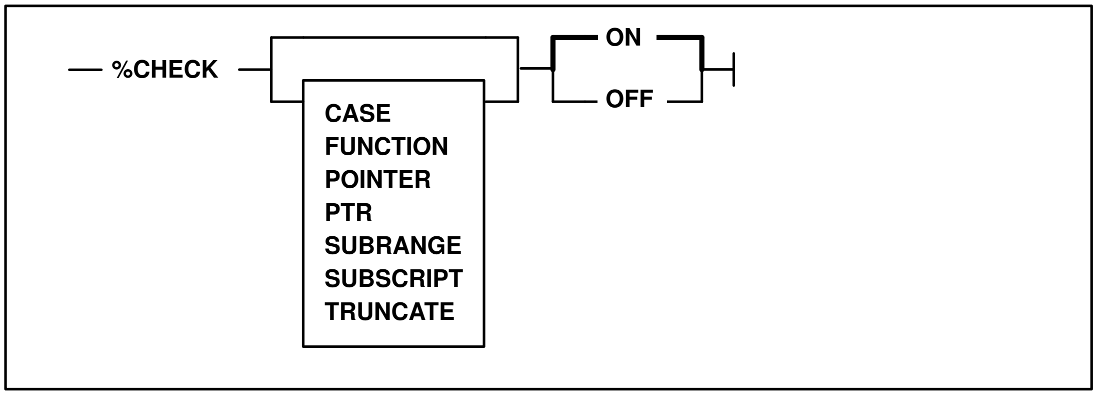
- **%CHECK**
- **%INCLUDE**
- **%LIST**
- **%MARGINS**
- **%WRITE**

%CHECK

Purpose

Controls the runtime checking features of XL Pascal. The checking can be enabled for part or all of the program.

Syntax



Parameters

CASE	checks whether the value of a **CASE** statement selector is not within any of the **CASE** ranges.
FUNCTION	checks the lack of an assignment of a value to a function before exiting from the function.
POINTER	checks the use of a pointer whose value is **NIL**.
PTR	checks the use of a pointer whose value is **NIL**. This parameter is synonymous with **POINTER**.
SUBRANGE	checks values assigned to subrange variables for proper range. This parameter also checks the range of values passed to subrange formal parameters.
SUBSCRIPT	checks the use of a subscript that is out of range for the array.
TRUNCATE	checks whether the value of a string fits into the target string on an assignment.
ON	turns checking on. If no checks are listed before **ON**, all checks are turned on. **ON** is the default.
OFF	turns checking off. If no checks are listed before **OFF**, all checks are turned off.

Description

If any of the checks are satisfied, a runtime trap occurs. If the **CHECK** option is missing, all of the checks apply. For example, **%CHECK ON** activates all of the checks.

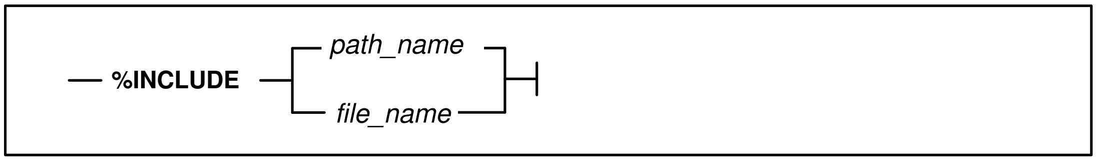
The **%CHECK** directive, like the other directives in this section, is a direction to the compiler. Its effect depends on where it appears in the text and is not subject to any structuring established by the program.

%INCLUDE

Purpose

Directs the compiler to insert source code from a file into the input stream immediately after the current line. More precisely, the compiler is directed to begin reading its input from a file. When it reaches the end of the file, the compiler resumes reading from the previous source.

Syntax



Parameters

path_name	is a fully qualified or relative path name of the file to be included.
file_name	is the name of the file to be included.

Description

If the file name is not a fully qualified or relative path name, the system searches the directories specified by all **–I** compiler options in the order in which those options were supplied. If the system cannot find the file, an error message is issued by the compiler and compilation continues.

The **%INCLUDE** directive is case sensitive. The name of the file to be included must be in the same case as the actual file name.

Example

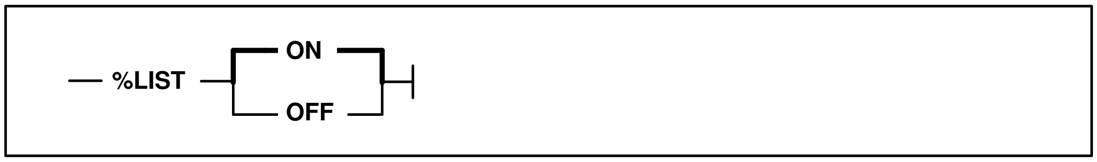
```
PROGRAM abc;

CONST
 %INCLUDE CONSTS

TYPE
 %INCLUDE TYPES

VAR
 %INCLUDE VARS

BEGIN
 ...
END.
```

%LIST

Purpose

Enables or disables the pseudo-assembler listing of the compiler. This option only takes effect if the **–qlist** compiler option is enabled.

Syntax



Parameters

ON	lists pseudoassembler code. The default is **ON**.
OFF	stops listing the pseudoassembler code.

The pseudoassembler listing for each procedure is controlled by the most recent **%LIST ON** or **%LIST OFF** before the procedure header.

%MARGINS

Purpose

Specifies the character positions of compiler input lines that can contain source code. The compiler skips all characters that lie outside the specified margins. The default is **%MARGINS** 1 256.

Syntax

```
—— %MARGINS —— integer1 —— integer2 ——┤
```

Parameters

integer1	is an unsigned **INTEGER** to indicate the new left margin.
integer2	is an unsigned **INTEGER** to indicate the new right margin.

Description

If the **%MARGINS** directive appears in a file to be included by the **%INCLUDE** directive, the new margins have effect only for the duration of that file. When the end of the file is reached and the previous source is resumed, the margin settings revert to their previous values.

%WRITE

Purpose

Allows you to write a message to the terminal during compilation at a specified location in the program. The **–qwrite** option must be specified for the **%WRITE** directive.

Syntax

```
—— %WRITE —— character_string ——┤
```

Parameter

character_string is any character string.

Appendix A. XL Pascal Language Modes

The **LANGLVL** compiler option determines the language mode that XL Pascal uses to compile your program. **DIALECT** and **IBMSET** are synonyms for **LANGLVL**.

Because it is possible to mix modes when you create a program, you may find it useful to know the features of XL Pascal that are common to both standard mode and VS mode. These features are summarized in the following tables. This appendix also briefly describes the suboptions that specify the two language modes.

Standard Mode XL Pascal

To use the features described throughout this manual as belonging to XL Pascal standard mode, specify **LANGLVL=STANDARD**. Standard mode Pascal is defined in the American National Standards Institute Pascal (X3.97–1983), more commonly known as ANSI–83. The language mode suboptions **ANSI–83** and **STD** are synonyms for **STANDARD**.

VS Mode XL Pascal

If you prefer to use features beyond the ANSI–83 language level, you should specify **LANGLVL=VS**, which is referred to throughout this manual as VS mode. It allows you to use all of the functions of ANSI–83 Standard Pascal and the selected facilities from IBM VS Pascal licensed program, implemented on the AIX RISC System/6000 computer. **LANGLVL=VS** is the default. The language mode suboption **IBM** is a synonym for suboption **VS**.

Comparison of Standard Mode and VS Mode Pascal

The following topics compare the features of the two language modes you can use under XL Pascal. The first column of the tables lists all the available features of Pascal. An x is placed under the appropriate column of the table if the particular item is available under that language mode. The column heading STD Mode refers to XL Pascal standard mode.

In the tables, the column headed VS Mode shows features available in XL Pascal when you select **LANGLVL=VS** or **LANGLVL=IBM**. If a feature in the ANSI–83 standard is modified or enhanced in VS mode, a description of the change is provided in the last column. If the feature belongs only to VS mode, no further description is added.

XL Pascal Program Structure

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
Declarations in any order		x	
Predefinition of INPUT and OUTPUT		x	
Redefinition of INPUT and OUTPUT		x	
Segment Unit		x	

XL Pascal Compiler Directives

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
%CHECK		x	
%CPAGE		x	
%INCLUDE		x	
%LIST		x	
%MARGINS		x	
%OPT		x	
%OPTION		x	
%PAGE		x	
%PRINT		x	
%PROCESS		x	
%SKIP		x	
%SPACE		x	
%TITLE		x	
%WRITE		x	

XL Pascal Constants

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
Predefined Constants	x	x	
Structured Constants		x	

XL Pascal Data Types

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
ALFA		x	
ALPHA		x	
ARRAY	x	x	
BOOLEAN	x	x	
CHAR	x	x	– MAXCHAR
GCHAR		x	
Enumerated Scalar	x	x	
FILE	x	x	
INTEGER	x	x	– MININT
POINTER		x	
Pointer Type	x	x	
REAL	x	x	– MINREAL, MAXREAL, and EPSREAL

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
RECORD	x	x	- Tag fields on the variant part can be back referenced. - Constant-expr can be used in addition to constant. - Record fields can be offset-qualified.
SET	x	x	
SHORTREAL		x	
SPACE		x	
STRING		x	
GSTRING		x	
STRINGPTR		x	
GSTRINGPTR		x	
Subrange Scalar	x	x	- PACKED or RANGE can define subrange. - Constant-expr can be used in addition to constant in certain instances.
TEXT	x	x	- INPUT and OUTPUT files need not be included in the program header.

XL Pascal Declarations

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
CONST	x	x	- Constant-expr can be used in addition to constant.
DEF		x	
FUNCTION	x	x	- Can return any type except a file, or any type containing a file.
Internal	x	x	
LABEL	x	x	- id can be used in addition to unsigned integer.
PROCEDURE	x	x	
REF		x	
STATIC		x	
TYPE	x	x	
VALUE		x	
VAR	x	x	- VAR declarations at the outermost level in programs or segments refer to global automatic data

XL Pascal Operators

Feature	**STD Mode**	**VS Mode**	**Comments About VS Mode Extensions**
NOT (~)	x	x	– ~ can be used in addition to NOT.
*	x	x	– SHORTREAL
/	x	x	– SHORTREAL
DIV	x	x	
MOD	x	x	
AND (&)	x	x	– & can be used in addition to AND.
||		x	
Shift operators (>> and <<)_		x	
+	x	x	– + can be used for string concatenation.
-	x	x	
OR (|)	x	x	– | can be used in addition to OR.
XOR (&&)		x	– && can be used in addition to XOR.
=	x	x	
<>	x	x	
~=		x	
<	x	x	
<=	x	x	
>	x	x	
>=	x	x	
IN	x	x	

XL Pascal Statements

Feature	**STD Mode**	**VS Mode**	**Comments About VS Mode Extensions**
ASSERT		x	
Assignment	x	x	
CASE	x	x	– The OTHERWISE statement can be specified. – Constant-expr can be used in addition to constant.
Compound	x	x	
CONTINUE		x	
Empty	x	x	
FOR	x	x	
GOTO	x	x	
IF	x	x	
LEAVE		x	

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
Procedure Call	x	x	- Components of packed objects can be passed by VAR. - Empty parentheses for user-defined functions with no parameters.
REPEAT	x	x	
RETURN		x	
WHILE	x	x	
WITH	x	x	

XL Pascal Routine Parameters

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
Formal routine	x	x	
Pass-by-conformant-string (VAR or CONST)		x	
Pass-by-read-only-reference (CONST)		x	
Pass-by-read/write-reference(VAR)	x	x	
Pass-by-value	x	x	

XL Pascal Routine Directives

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
EXTERNAL		x	
FORTRAN	x	x	- Equivalent to EXTERNAL, but has no effect on an XL Pascal program.
FORWARD	x	x	
MAIN	x	x	- Equivalent to EXTERNAL, but has no effect on an XL Pascal program.
NONPASCAL	x	x	- A special case of EXTERNAL required by the parameter passing conventions of calls to other XL languages.
REENTRANT	x	x	- Equivalent to EXTERNAL, but has no effect on an XL Pascal program.

XL Pascal Conversion Routines

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
CHR	x	x	
FLOAT		x	
ITOHS		x	
ORD	x	x	- Pointer to integer conversions.

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
ROUND	x	x	– SHORTREAL
Ordinal Conversion		x	
STR		x	
TRUNC	x	x	

XL Pascal Data Access Routines

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
ADDR		x	
HBOUND		x	
HIGHEST		x	
LBOUND		x	
LOWEST		x	
SIZEOF		x	

XL Pascal Data Movement Routines

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
PACK	x	x	
UNPACK	x	x	

XL Pascal General Routines

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
HALT		x	
TRACE		x	
xl__trap		x	

XL Pascal I/O Routines

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
CLOSE		x	
COLS		x	
EOF	x	x	
EOLN	x	x	
File Open Options		x	
GET	x	x	
PAGE	x	x	
PUT	x	x	
READ	x	x	
READLN	x	x	

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
RESET	x	x	– STRING
REWRITE	x	x	– STRING
SEEK		x	
TERMIN		x	
TERMOUT		x	
UPDATE		x	
WRITE	x	x	
WRITELN	x	x	

XL Pascal Mathematical Routines

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
ABS	x	x	– SHORTREAL
ARCTAN	x	x	
COS	x	x	
EXP	x	x	
LN	x	x	
MAX		x	
MIN		x	
ODD	x	x	
PRED	x	x	
RANDOM		x	
SIN	x	x	
SQR	x	x	– SHORTREAL
SQRT	x	x	– SHORTREAL
SUCC	x	x	

XL Pascal Mixed String Routines

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
MCOMPRESS		x	
MDELETE		x	
MINDEX		x	
MLENGTH		x	
MLTRIM		x	
MRINDEX		x	
MSUBSTR		x	
MTRIM		x	

XL Pascal Storage Management Routines

Feature	**STD Mode**	**VS Mode**	**Comments About VS Mode Extensions**
DISPOSE	x	x	– form 3
DISPOSEHEAP		x	
MARK		x	
NEW	x	x	– form 3
NEWHEAP		x	
QUERYHEAP		x	
RELEASE		x	
USEHEAP		x	

XL Pascal String Routines

Feature	**STD Mode**	**VS Mode**	**Comments About VS Mode Extensions**
COMPRESS		x	
DELETE		x	
INDEX		x	
LENGTH		x	
GSTR		x	
GTOSTR		x	
LPAD		x	
LTOKEN		x	
LTRIM		x	
MAXLENGTH		x	
PICTURE		x	
READSTR		x	
RINDEX		x	
STOGSTR		x	
RPAD		x	
SUBSTR		x	
TOKEN		x	
TRIM		x	
WRITESTR		x	

XL Pascal System Interface Routines

Feature	**STD Mode**	**VS Mode**	**Comments About VS Mode Extensions**
CLOCK		x	
DATETIME		x	
PARMS		x	
RETCODE		x	

XL Pascal Variables

Feature	STD Mode	VS Mode	Comments About VS Mode Extensions
Automatic	x	x	
Dynamic	x	x	
Static		x	
External (DEF and REF)		x	
Parameter	x	x	
Predefined	x	x	– STDERR.
Subscripted	x	x	– STRING and GSTRING.
Field References	x	x	
Pointer References	x	x	
File References	x	x	
SPACE References		x	
String References	x	x	

Appendix B. Predefined Identifiers in XL Pascal

The following tables list all of the predefined identifiers of XL Pascal in alphabetical order, with a brief description of each.

The names of predefined constants, types, variables, and routines in XL Pascal are declared for you in every unit before the start of your program. You can redefine these names if you want; however, it is better to use them according to their predefined meanings.

Standard Mode

Identifier	**Form**	**Description**
ABS	Function	Computes the absolute value of a number.
ARCTAN	Function	Returns the arctangent of the argument.
BOOLEAN	Type	Enumerated data type whose values are (FALSE,TRUE).
CHAR	Type	Character data type.
CHR	Function	Converts an integer to a character value.
COS	Function	Returns the cosine of the argument.
DISPOSE	Procedure	Deallocates a dynamic variable.
EOF	Function	Tests the file for end-of-file condition.
EOLN	Function	Tests the file for end-of-line condition.
EXP	Function	Returns base of natural log (e) raised to the power of the argument.
FALSE	Constant	Constant of type BOOLEAN, FALSE < TRUE.
GET	Procedure	Advances the file pointer to the next element of an input file.
INPUT	Variable	Default input file, TEXT data type.
INTEGER	Type	Integer data type.
LN	Function	Returns the natural logarithm of the argument.
MAXINT	Constant	Maximum value of type INTEGER.
NEW	Procedure	Allocates a dynamic variable in the most recent heap.
ODD	Function	Returns TRUE if integer argument is odd.
ORD	Function	Converts an ordinal value to an integer.
OUTPUT	Variable	Default output file, TEXT data type.
PACK	Procedure	Copies an array to a packed array.
PAGE	Procedure	Skips to the top of the next page.
PRED	Function	Obtains the predecessor of an ordinal type.

Identifier	Form	Description
PUT	Procedure	Advances file pointer to next element of output file.
READ	Procedure	Reads data from a file.
READLN	Procedure	Reads one line of a TEXT file.
REAL	Type	Floating point represented in long floating-point format.
RESET	Procedure	Opens a file for input.
REWRITE	Procedure	Opens a file for output.
ROUND	Function	Converts a floating-point number to an integer by rounding.
SIN	Function	Returns the sine of the argument.
SQR	Function	Returns the square of the argument.
SQRT	Function	Returns the square root of the argument.
SUCC	Function	Obtains the successor of an ordinal type.
TEXT	Type	File of character lines.
TRUE	Constant	Constant of type BOOLEAN, TRUE > FALSE.
TRUNC	Function	Converts a floating-point number to an integer by truncating.
UNPACK	Procedure	Copies a packed array to an array.
WRITE	Procedure	Writes data to a file.
WRITELN	Procedure	Writes one line to a TEXT file.

VS Mode

Identifier	Form	Description
ADDR	Function	Returns the offset of a variable.
ALFA	Type	Array of 8 characters, indexed 1..ALFALEN.
ALFALEN	Constant	HBOUND of type ALFA, value is 8.
ALPHA	Type	Array of 16 characters, indexed 1..ALPHALEN.
ALPHALEN	Constant	HBOUND of type ALPHA, value is 16.
CLOCK	Function	Returns the number of microseconds of run time.
CLOSE	Procedure	Closes a file.
COLS	Function	Returns the current column on the output line.
COMPRESS	Function	Replaces multiple blanks in a string with one blank.
DATETIME	Procedure	Returns the current date and time of day.
DELETE	Function	Returns a string with a portion removed.

Identifier	**Form**	**Description**
DISPOSEHEAP	Procedure	Frees dynamic variables allocated in a heap.
EPSREAL	Constant	Smallest REAL magnitude that, when added to 1, is detectable.
EPSSREAL	Constant	Smallest SHORTREAL magnitude that, when added to 1, is detectable.
FLOAT	Function	Converts an integer to a floating-point value.
GCHAR	Type	Holds a single character from the **AIX** Extended (Multibyte) Character Set.
GSTR	Function	Converts an array of GCHAR to a GSTRING.
GSTRING	Type	Packed array of multibyte characters whose length varies during run time up to a maximum length.
GSTRINGPTR	Type	A type for dynamically allocated multibyte character strings of a length determined at run time.
GTOSTR	Function	Converts a GSTRING to a STRING.
HALT	Procedure	Halts running of the program.
HBOUND	Function	Returns the upper bound of an array.
HIGHEST	Function	Returns the maximum value of an ordinal type.
INDEX	Function	Finds the first occurrence of one string in another.
LBOUND	Function	Returns the lower bound of an array.
LENGTH	Function	Returns the current length of a string.
LOWEST	Function	Returns the minimum value of an ordinal type.
LPAD	Procedure	Pads strings on the left.
LTOKEN	Procedure	Extracts tokens from a string.
LTRIM	Function	Returns a string with leading blanks removed.
MARK	Procedure	Creates a new subheap.
MAX	Function	Returns the maximum value of a list of scalars.
MAXCHAR	Constant	Maximum value of type CHAR: 'FF'XC.
MAXLENGTH	Function	Returns the maximum length of a string.
MAXREAL	Constant	Maximum value of type REAL.
MAXSREAL	Constant	Maximum value of type SHORTREAL.
MCOMPRESS	Function	Replaces multiple blanks in a mixed string with one blank.
MDELETE	Function	Returns a mixed string with a portion removed.
MIN	Function	Returns the minimum value of a list of scalars.

Identifier	**Form**	**Description**
MINDEX	Function	Finds the first occurrence of one mixed string in another.
MININT	Constant	Minimum value of type INTEGER.
MINREAL	Constant	Minimum value of type REAL (smallest nonzero floating-point number).
MINSREAL	Constant	Minimum positive value of type SHORTREAL.
MLENGTH	Function	Returns the current length of a mixed string.
MLTRIM	Function	Returns a mixed string with leading blanks removed.
MRINDEX	Function	Finds the last occurrence of one mixed string in another.
MSUBSTR	Function	Returns a specific portion of a mixed string.
MTRIM	Function	Returns a mixed string with trailing blanks removed.
NEWHEAP	Procedure	Creates a heap.
PARMS	Function	Returns the system-dependent invocation parameters.
POINTER	Type	Conforms to any actual parameter pointer type.
QUERYHEAP	Procedure	Obtains heap-id of the currently active heap.
RANDOM	Function	Returns a pseudo random number.
READSTR	Procedure	Converts a string to values assigned to variables.
RELEASE	Procedure	Frees storage in one or more subheaps.
RETCODE	Procedure	Sets the system-dependent return code.
RINDEX	Function	Finds the last occurrence of one string in another.
RPAD	Procedure	Pads strings on the right.
SEEK	Procedure	Positions an opened file at a specific record.
SHORTREAL	Type	Floating point represented in short floating-point format.
SIZEOF	Function	Returns the memory size of a variable or type.
STDERR	Variable	Default standard error file, TEXT data type.
STOGSTR	Function	Converts a STRING to a GSTRING.
STR	Function	Converts an array of characters to a STRING.
STRING	Type	Packed array of characters whose length varies during run time up to a maximum length.

Identifier	**Form**	**Description**
STRINGPTR	Type	A type for dynamically allocated strings of a length determined at run time.
SUBSTR	Function	Returns a specific portion of a string.
TERMIN	Procedure	Opens a file for input from the terminal.
TERMOUT	Procedure	Opens a file for output to the terminal.
TOKEN	Procedure	Extracts tokens from a string.
TRACE	Procedure	Writes the routine return stack.
TRIM	Function	Returns a string with trailing blanks removed.
UPDATE	Procedure	Opens a file for both input and output.
USEHEAP	Procedure	Establishes a new currently active heap.
WRITESTR	Procedure	Converts a series of expressions into a string.
xl__trap	Procedure	Produces traceback if a trap occurs.

Index

Communicating Your Comments to IBM

IBM AIX XL Pascal Compiler/6000
Language Reference

Version 2.1

Publication number SC09–1757–00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage–paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - United States and Canada: 416–448–6161
  - Other countries: (+1)–416–448–6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
  - Internet: **torrcf@vnet.ibm.com**
  - IBMLINK: **toribm(torrcf)**
  - IBM/PROFS: **torolab4(torrcf)**
  - IBMMAIL: **ibmmail(caibmwt9)**

Readers' Comments – We'd Like to Hear from You

IBM AIX XL Pascal Compiler/6000
Language Reference

Version 2.1

Publication Number SC09–1757–00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall Satisfaction	☐	☐	☐	☐	☐

How satisfied were you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone Number	

Readers' Comments—We'd Like to Hear from You

SC09–1757–00

Fold and Tape **Please Do Not Staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IIBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Cut or Fold Along Line

Fold and Tape **Please Do Not Staple** Fold and Tape

SC09–1757–00