LANGUAGE SPECIFICATION

LISA PASCAL

19 February 1982

Rich Page / David Casseres

[file pas:p0; date 2/19/82]

[file pas:p0; date 2/19/82]

DATE: 19 February 1982
TO: List
FROM: Rich Page & David Casseres
SUBJECT: Lisa Pascal Language Specification

Here is the new Language Specification for Lisa Pascal. Please direct your comments to Rich Page.

This Language Specification attempts to describe both the eventual definition of the language and the current implementation. Inevitably, there are a few places where the two do not match, and these points are spelled out in the text.

[file pas:p0; date 2/19/82]

[file pas:p0; date 2/19/82]

CONTENTS

1. INTRODUCTION.....1
1.1 Product Name and Number.....1
1.2 Related Documents.....1
1.3 Novel or Unusual Features.....1
1.4 Relation to Other Products.....2
1.5 Use Environment.....2
1.5.1 Hardware Environment.....2
1.5.2 Software Environment.....2

2 DEFINITIONS.....3

3 METALANGUAGE.....5

4 LEXICAL TOKENS.....7
4.1 General.....7
4.2 Special Symbols.....7
4.3 Identifiers.....8
4.4 Directives.....8
4.5 Numbers.....8
4.6 Labels.....9
4.7 Quoted String Constants.....9
4.7.1 Character Constants.....10
4.8 Comments, Spaces, and Ends of Lines.....10

5. BLOCKS, LOCALITY, AND SCOPE.....13
5.1 Block.....13
5.2 Scope.....14
5.2.1 Defining Occurrence.....14
5.2.2 Redefinition.....14
5.2.3 Position of Defining Occurrence.....15

6 CONSTANT-DEFINITIONS.....17

7 TYPE-DEFINITIONS.....19
7.1 General.....19
7.2 Simple-Types.....19
7.2.1 General.....19
7.2.2 Standard Simple-Types.....20
7.2.3 Enumerated-Types.....23
7.2.4 Subrange-Types.....23

7.3 Structured-Types....24
7.3.1 General....24
7.3.2 Array-Types....25
7.3.3 Record-Types....26
7.3.4 Set-Types....28
7.3.5 File-Types....28
7.4 Pointer-Types....29
7.5 Identical and Compatible Types....29
7.5.1 Type Identity....30
7.5.2 Compatibility of Types....31
7.5.3 Assignment-Compatibility....31
7.6 Example of a Type-Definition-Part....33

8 DECLARING AND REFERENCING VARIABLES....35
8.1 Variable-Declarations....35
8.2 Variable-References....35
8.3 Selectors and Components....36
8.3.1 General....36
8.3.2 Arrays, Strings, and Indexes....36
8.3.3 Records and Field-Designators....37
8.3.4 File-Buffers....37
8.4 Pointer-References....38

9 PROCEDURES AND FUNCTIONS....39
9.1 Procedure-Declarations....39
9.2 Function-Declarations....41
9.3 Parameters....43
9.3.1 General....43
9.3.2 Value Parameters....45
9.3.3 Variable Parameters....45
9.3.4 Procedural Parameters....45
9.3.5 Functional Parameters....48
9.3.6 Parameter List Compatibility....48

10 EXPRESSIONS....51
10.1 General....51
10.2 Operators....52
10.2.1 Syntax....52
10.2.2 Arithmetic Operators....52
10.2.3 Boolean Operators....55
10.2.4 Set Operators....55
10.2.5 Relational Operators....56
10.2.6 @-Operator....57
10.3 Function-Calls....59
10.4 Set-Constructors....60

11 STATEMENTS.....61
11.1 General.....61
11.2 Simple Statements.....61
11.2.1 General.....61
11.2.2 Assignment-Statements.....61
11.2.3 Procedure-Statements.....62
11.2.4 GOTO Statements.....62
11.3 Structured-Statements.....63
11.3.1 General.....63
11.3.2 Compound-Statements.....63
11.3.3 Conditional-Statements.....63
11.3.4 Repetitive-Statements.....65
11.3.5 WITH-Statements.....69

12 TEXTFILE INPUT AND OUTPUT.....71
12.1 General.....71
12.2 The READ Procedure.....73
12.3 The READLN Procedure.....74
12.4 The WRITE Procedure.....75
12.5 The WRITELN Procedure.....79
12.6 The EOLN Function.....79
12.7 The PAGE Procedure.....79

13 PROGRAMS.....81

14 UNITS.....83
14.1 Regular-Units.....83
14.1.1 Writing Regular-Units.....84
14.1.2 Using Regular-Units.....85
14.2 Intrinsic-Units.....86
14.2.1 Writing Intrinsic-Units.....86
14.2.2 Using Intrinsic-Units.....87
14.3 Nested Units.....87

15 THE LISA PASCAL COMPILER.....89
15.1 Compiler Options.....89
15.2 Conditional Compilation.....91
15.2.1 Compile-Time Variables.....91
15.2.2 Compile-Time Expressions.....92
15.2.3 The SETC Option.....93
15.2.4 The IFC, ELSEC, and ENDC Options.....93
15.3 Optimization of IF Statement.....94
15.4 Optimization of WHILE and REPEAT Statements.....96
15.5 Using CASE Statements for Efficiency.....96

16 STANDARD PROCEDURES AND FUNCTIONS.....97
16.1 General.....97
16.2 Basic I/O.....97
16.2.1 RESET.....97
16.2.2 REWRITE.....98
16.2.3 CLOSE.....98
16.2.4 EOF.....99
16.2.5 EOLN.....99
16.2.6 SEEK.....99
16.2.7 PUT.....100
16.2.8 GET.....100
16.2.9 Control Characters With GET and PUT.....101
16.2.10 IORESULT.....102
16.2.11 GOTOXY.....102
16.3 Untyped File I/O.....102
16.3.1 BLOCKREAD.....103
16.3.2 BLOCKWRITE.....104
16.4 Device I/O.....104
16.4.1 UNITREAD.....105
16.4.2 UNITWRITE.....105
16.4.3 Device I/O Modes.....106
16.4.4 UNITCLEAR.....107
16.4.5 UNITBUSY.....107
16.5 EXIT and HALT Procedures.....107
16.5.1 EXIT.....107
16.5.2 HALT.....108
16.6 Dynamic Allocation Procedures.....108
16.6.1 NEW.....108
16.6.2 DISPOSE.....109
16.6.3 MARK.....109
16.6.4 RELEASE.....109
16.6.5 MEMAVAIL.....110
16.7 Transfer Procedures and Functions.....110
16.7.1 TRUNC.....110
16.7.2 ROUND.....110
16.7.3 ORD4.....110
16.7.4 POINTER.....111

16.8 Arithmetic Functions.....112
16.8.1 ODD.....112
16.8.2 ABS.....112
16.8.3 SQR.....112
16.8.4 SIN.....112
16.8.5 COS.....112
16.8.6 EXP.....112
16.8.7 LN.....112
16.8.8 SQRT.....112
16.8.9 ARCTAN.....113
16.9 Ordinal Functions.....113
16.9.1 ORD.....113
16.9.2 CHR.....113
16.9.3 SUCC.....114
16.9.4 PRED.....114
16.10 String Procedures and Functions.....114
16.10.1 LENGTH.....114
16.10.2 POS.....114
16.10.3 CONCAT.....115
16.10.4 COPY.....115
16.10.5 DELETE.....115
16.10.6 INSERT.....115
16.11 Byte-Oriented Procedures and Functions.....116
16.11.1 MOVELEFT.....116
16.11.2 MOVERIGHT.....117
16.11.3 SIZEOF.....117
16.12 Miscellaneous Procedures and Functions.....117
16.12.1 SCANEQ.....117
16.12.2 SCANNE.....118
16.12.3 FILLCHAR.....118

Appendix A: LISA PASCAL AND UCSD PASCAL.....119
Appendix B: KNOWN ANOMALIES IN LISA PASCAL.....123
Appendix C: SYNTAX OF THE LANGUAGE.....127

Section 1

INTRODUCTION

1.1 Product Name and Number

Lisa Pascal, E112

1.2 Related Documents

- Pascal User Manual and Report, Jensen and Wirth, 1975. Original definition of Pascal.
- ISO Working Draft of Standard Pascal, 1979. Redefinition of Pascal, used as standard for implementation of Lisa Pascal.
- Apple Pascal Language Reference Manual, Casseres, 1980. Describes differences between Apple II (UCSD) Pascal and original definition in Jensen & Wirth.
- Apple III Pascal Programmer's Manual, Casseres, 1981. Complete description of Apple III (UCSD) Pascal.
- Lisa Pascal Development System Manual, Schottstaedt, 1982.
- Pascal Development System Internal Documentation, Schottstaedt, 1982.
- 68000 Pascal Compiler Language Specification, Glanville, 1980.

1.3 Novel or Unusual Features

In addition to providing nearly all the features of standard Pascal, as described in the Pascal User Manual and Report (Jensen and Wirth), Lisa Pascal provides a variety of extensions. These are summarized in Appendix A. They include 32 bit integers, an OTHERWISE clause in CASE statements, procedural and functional parameters with type-checked parameter lists, and the @ pointer operator. The real arithmetic conforms to single-precision aspects of the proposed IEEE standard.

1.4 Relation to Other Products

Apple will maintain only one version of Pascal for the Lisa computer. The language specified herein is reasonably compatible with the UCSD Pascal used on Apple II and Apple III. See Appendix A for a discussion of the differences between these forms of Pascal.

1.5 Use Environment

1.5.1 Hardware Environment

The compiler operates in a standard minimum Lisa configuration: 256K data/program memory and two floppy disk drives.

1.5.2 Software Environment

The compiler makes few assumptions about its software environment. It can run on either the Operating System or the Monitor. Note that the compiler requires a minimum of 128K to compile a null program.

Section 2

DEFINITIONS

For the purposes of this document the following definitions are used:

- Error - A violation by a program of the requirements of this specification such that detection normally requires execution of the program.

- Scope - The text for which the declaration or definition of an identifier or label is valid.

- Undefined - The value of a variable or function when the variable does not necessarily have a meaningful value of its type assigned to it.

- Unspecified - A value or action or effect that, although possibly well-defined, is not specified and may not be the same in all cases or for all versions or configurations of the system. Any programming construct that leads to an unspecified result or effect is not supported.

Section 3

METALANGUAGE

The metalanguage used in this document to specify the constructs is based on Backus-Naur form. The notation has been modified from the original to permit greater convenience of description and to allow for iterative productions to replace recursive ones. Table 1 lists the meanings of the various meta-symbols.

Table 1. Metalanguage Symbols

META-SYMBOL	MEANING
=	is defined to be
|	alternatively
.	end of definition
[x]	Ø or 1 instance of x
{x}	Ø or more repetitions of x
(x|y|..|z)	grouping: any one of x,y,..z
"xyz"	the terminal symbol xyz
lower-case-name	a non-terminal symbol

For increased readability, lower-case-names are hyphenated. The juxtaposition of two meta-symbols in a production implies the concatenation of the text they represent. Within Section 4 below this concatenation is direct; no characters may intervene. In all other parts of this document the concatenation is in accordance with the rules set out in Section 4.

The characters required to form Pascal programs are those implicitly required to form the symbols and separators defined in Section 4.

Section 4

LEXICAL TOKENS

4.1 General

The lexical tokens used to construct Pascal programs are classified into special symbols, identifiers, numbers, labels and quoted string constants.

NOTE: The syntax given in this sub-clause describes the formation of these tokens from characters and their separation. It therefore does not adhere to the same rules as the syntax in the rest of this document.

```
letter = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"
       | "L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"
       | "W"|"X"|"Y"|"Z"
       | "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"
       | "l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"
       | "w"|"x"|"y"|"z"

digit = "Ø"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

4.2 Special Symbols

The special symbols are tokens having a fixed meaning. They are used to specify the syntactic structures of the language.

```
special-symbol = "+"|"-"|"*"|"/"|"="
               | "<"|">"|"["|"]"|"."|"("|")"
               | ","|":"|";"|"^"|"@"|"$"
               | "<>"|"<="|">="|":="|".."| word-symbol

   word-symbol = "AND"|"ARRAY"|"BEGIN"|"CASE"|"CONST"|"DIV"
               | "DOWNTO"|"DO"|"ELSE"|"END"|"FILE"|"FOR"
               | "FUNCTION"|"GOTO"|"IF"|"IMPLEMENTATION"
               | "IN"|"INTERFACE"|"INTRINSIC"|"LABEL"|"MOD"
               | "NIL"|"NOT"|"OF"|"OR"|"OTHERWISE"|"PACKED"
               | "PROCEDURE"|"PROGRAM"|"RECORD"|"REPEAT"
               | "SET"|"STRING"|"THEN"|"TO"|"TYPE"|"UNIT"
               | "UNTIL"|"USES"|"VAR"|"WHILE"|"WITH"
```

Matching upper and lower case letters are equivalent in word-symbols.

4.3 Identifiers

Identifiers serve to denote constants, types, variables, procedures, functions, units and programs, and fields in records. Identifiers can be of any length, but only the first 8 characters are significant. Matching upper and lower case letters are equivalent in identifiers.

```
identifier = letter {(letter | digit | "_")}
```

Examples:

```
X     Rome     gcd     SUM     get_byte
```

4.4 Directives

Directives can only occur immediately after a procedure-heading or a function-heading.

```
directive = "FORWARD" | "EXTERNAL"
```

These directives are discussed in 9.1.

4.5 Numbers

The usual decimal notation is used for numbers that are constants of the data types integer, longint, and real (see Section 7.2.2). The letter E preceding the scale factor means "times ten to the power of". In addition, a hexadecimal constant can be written by using the $ character as a prefix.

```
digit-sequence = digit {digit}
hex-digit-sequence = hex-digit {hex-digit}
hex-digit = digit|"A"|"B"|"C"|"D"|"E"|"F"
unsigned-integer = digit-sequence
hex-integer = "$" hex-digit-sequence
unsigned-real = digit-sequence "." digit-sequence
                  ["E" scale-factor]
              | digit-sequence "E" scale-factor
unsigned-number = unsigned-integer
                | hex-integer
                | unsigned-real
scale-factor = signed-integer

sign = "+" | "-"
signed-integer = [sign] unsigned-integer
signed-number = [sign] unsigned-number
```

NOTE: Lower-case "e" is legal in place of upper-case "E".

Examples:

```
1     +100     -0.1     5E-3     87.35e+8     $A05D
```

4.6 Labels

Labels are unsigned integers and are distinguished by their apparent integral values, which must be in the range 0..9999.

```
label = unsigned-integer
```

If a statement is prefixed by a label, a goto statement can refer to it.

4.7 Quoted String Constants

A quoted string constant is a sequence of zero or more characters enclosed by apostrophes. Currently, the maximum number of characters is 255. A quoted string constant with nothing between the apostrophes denotes the null string.

If the quoted string constant is to contain an apostrophe, this apostrophe must be written twice.

```
quoted-string-constant = "'" {string-character} "'"
string-character = any ascii char except CR or "'"
                 | "'" "'"
```

NOTE: The CR character (ASCII 13) cannot be used in a quoted string constant.

Examples:

```
'Pascal'   'THIS IS A STRING'      'Don''t worry!'
'A'          ';'           ''''
```

All string values have a length attribute (see Section 7.2.2.6). In the case of a string constant value the length is fixed; it is equal to the actual number of characters in the string value.

4.7.1 Character Constants

A character constant is simply a string constant whose length is exactly 1. It is compatible with any char-type or string-type.

4.8 Comments, Spaces, and Ends of Lines

The constructs:

"{" any-sequence-of-symbols-not-containing-right-brace "}"

"(*" any-sequence-not-containing star-right-paren "*)"

are called comments.

A compiler option is a comment that begins with a $ character (immediately after the { or (* that begins the comment). The $ character is followed by the mnemonic of the compiler option (see Section 15).

Apart from the effects of compiler options, the substitution of a space for a comment does not alter the meaning of a program.

Comments, spaces, and ends of lines are considered token separators. An arbitrary number of separators can occur between any two consecutive tokens, or before the first token of a program text. There must be at least one separator between any consecutive pair of tokens made up of identifiers, word-symbols,

or numbers. Except for spaces in quoted string constants, no separators can occur within tokens.

A comment cannot be nested within another comment formed with the same kind of delimiters. However, a comment formed with {...} delimiters can be nested within a comment formed with (*...*) delimiters, and vice versa.

Section 5

BLOCKS, LOCALITY, AND SCOPE

5.1 Block

A block consists of the definitions, declarations, and statement-part which together form a part of a procedure-declaration, a function-declaration or a program. All identifiers and labels with a defining occurrence in a particular block are local to that block.

```
block = [ label-declaration-part ]
          [ constant-definition-part ]
            [ type-definition-part ]
              [ variable-declaration-part ]
                [ procedure-and-function-declaration-part ]
                  statement-part
```

The label-declaration-part specifies all labels that mark a statement in the corresponding statement-part. Each label must mark exactly one statement in the statement-part. The appearance of a label in a label-declaration is the defining occurrence for the block in which the declaration occurs.

```
label-declaration-part = "LABEL" label {"," label} ";"
```

The constant-definition-part contains all constant-definitions local to the block.

```
constant-definition-part = "CONST" constant-definition ";"
                           {constant-definition ";"}
```

The type-definition-part contains all type-definitions local to the block.

```
type-definition-part = "TYPE" type-definition ";"
                       {type-definition ";"}
```

The variable-declaration-part contains all variable-declarations local to the block.

```
variable-declaration-part = "VAR" variable-declaration ";"
                            {variable-declaration ";"}
```

The procedure-and-function-declaration-part contains all procedure and function declarations local to the block.

```
procedure-and-function-declaration-part =
  {(procedure-declaration | function-declaration) ";"}
```

The statement-part specifies the algorithmic actions to be executed upon an activation of the block.

```
statement-part = compound-statement
```

At run time, all variables have values that are unspecified upon entry to the statement-part.

5.2 Scope

5.2.1 Defining Occurrence

Each identifier or label within a block of a Pascal program must have a defining occurrence (declaration, constant definition, procedure definition, or function definition) whose scope encloses all corresponding occurrences of the identifier or label in the program text.

This scope is the block that contains the defining occurrence, and all blocks enclosed by that block, subject to the requirements of 5.2.2.

5.2.2 Redefinition

If an identifier that has a defining occurrence for block A has a further defining occurrence for some block B enclosed by A, block B and all blocks enclosed by B are excluded from the scope of the defining occurrence for block A. (See Appendix B.)

An identifier that is a field-identifier can be used as a field-identifier within a field-designator in any block in which a variable of the corresponding record-type is accessible.

5.2.3 Position of Defining Occurrence

The defining occurrence of an identifier or label must precede all corresponding occurrences of that identifier or label in the program text with one exception: A type-identifier TYP that specifies the domain of a pointer-type ^TYP can have its defining occurrence anywhere in the type-definition-part in which ^TYP occurs. (See Appendix B.)

An identifier or label cannot have more than one defining occurrence for a particular block.

Section 6

CONSTANT-DEFINITIONS

A constant-definition introduces an identifier to denote a constant.

```
constant-definition = identifier "=" constant
constant = constant-identifier
         | signed-number
         | quoted-string-constant
constant-identifier = identifier
```

The occurrence of an identifier on the left hand side of a constant-definition is its defining occurrence as a constant-identifier for the block in which the constant-definition occurs. The scope of a constant-identifier does not include its own definition.

A constant-identifier following a sign denotes a value of type integer, longint, or real.

Section 7

TYPE-DEFINITIONS

7.1 General

A type determines the set of values which variables of that type can assume, and the operations that can be performed upon them. A type-definition associates an identifier with a type.

```
type-definition = identifier "=" type
type = simple-type
     | structured-type
     | pointer-type
```

The occurrence of an identifier on the left hand side of a type-definition is its defining occurrence as a type-identifier for the block in which the type-definition occurs. The scope of a type-identifier does not include its own definition, except for pointer-types (see 5.2.3).

To help clarify the syntax description with some semantic hints, the following terms are used to distinguish identifiers according to what they denote.

```
simple-type-identifier = type-identifier
structured-type-identifier = type-identifier
pointer-type-identifier = type-identifier
type-identifier = identifier
```

7.2 Simple-Types

7.2.1 General

All the simple types define ordered sets of values.

```
simple-type = ordinal-type
            | real-type
            | string-type
ordinal-type = enumerated-type
             | subrange-type
             | ordinal-type-identifier
ordinal-type-identifier = type-identifier
real-type = real-type-identifier
real-type-identifier = type-identifier
```

The standard ordinal-types are integer, longint, char, and boolean. Any other ordinal-type-identifier must be defined to denote an ordinal-type. Any real-type-identifier other than real must be defined to denote the standard type real. String-types are discussed in 7.2.2.6 below.

7.2.2 Standard Simple-Types

A standard type is denoted by a predefined type-identifier. The following types are standard:

7.2.2.1 Integer

The values are a subset of the whole numbers denoted as specified in 4.5. The predefined integer constant maxint is defined to be 32767. Maxint defines the subset of the integers over which the integer operations are defined. The range is the set of values:

-maxint, -maxint+1, ... -1, Ø, 1, ... maxint-1, maxint

These are 16-bit integers. Note that the type integer is not equivalent to the subrange type -maxint..maxint; the subrange type implies range-checking, while the type integer implies that values outside the range are truncated to 16 bits.

7.2.2.2 Longint

The values are a subset of the whole numbers denoted as specified in 4.5. The range is the set of values from -(2**31-1) to 2**31-1, i.e., -2147483647 to 2147483647.

These are 32-bit integers.

NOTE: Integer arithmetic is done in both 16-bit and 32-bit precision. Mixed sized operand expressions are evaluated in a manner similar to the FORTRAN single/double precision floating point arithmetic rules:

- All variables of type integer have 16-bit values. All variables of type longint have 32-bit values.

- All integer constants in the range -maxint..maxint are considered to be 16-bit values. All integer constants in the range of longint values but not in the range -maxint..maxint are considered to be 32-bit values.

- When both arguments to an operator (or the single argument to a unary operator) are 16-bit or smaller integer quantities, 16-bit operations are always performed. Smaller values are converted to 16-bit values prior to the operation.

- When one or more argument is a 32-bit value, all operands are first converted to 32-bits, and a 32-bit result is computed.

- The expression on the right of an assignment statement is evaluated independently of the size of the variable on the left. If necessary, the value of the expression is truncated or extended to match the size of the variable on the left.

Note that the ord4 function (see 16.7.3) can be used to convert a 16-bit integer value to a 32-bit value.

NOTE: There is a performance penalty for the use of longint values. The penalty is essentially a factor of 2 for operations other than division and multiplication; for division and multiplication, the penalty is much worse than a factor of 2.

7.2.2.3 Real

The values are a subset of the real numbers denoted as specified by 4.5.

7.2.2.4 Boolean

The values are truth values denoted by the identifiers false and true, such that false is less than true. The function-call ord(false) returns Ø, and ord(true) returns 1.

7.2.2.5 Char

The values are extended 8-bit ASCII, represented by ordinal values in the range Ø..255. The ordering properties of the character values are defined by the ordering of these ordinal values, i.e. the relationship between the character variables c1 and c2 is the same as the relationship between ord(c1) and ord(c2).

7.2.2.6 String

A string value is a sequence of characters that has a dynamic length attribute. The length is the actual number of characters in the sequence at any time during program execution. A string type also has a static size attribute. The size is the maximum limit on the length of the value. The current value of the length attribute is returned by the standard function LENGTH; the size attribute of a string type is determined when the string type is defined.

```
string-type = "STRING" "[" size attribute ]"
```

Do not confuse the size with the length.

where the size attribute is an unsigned number in the range 1..255. The upper limit, 255, is for the current implementation.

Note that the size attribute of a string constant is equal to the length of the string constant value, namely the number of characters actually in the string.

All string-types are implicitly packed types. Do not make any assumptions about the internal storage format of strings, as this format may not be the same in all implementations.

NOTE: Operators applicable to standard types are specified in Section 10.

7.2.3 Enumerated-Types

An enumerated-type defines an ordered set of values by enumeration of the identifiers which denote these values. The ordering of these values is determined by the sequence in which the constants are listed.

```
enumerated-type = "(" identifier-list ")"
identifier-list = identifier { "," identifier }
```

The occurrence of an identifier within the identifier-list of an enumerated-type is its defining occurrence as a constant for the block in which the enumerated-type occurs.

Examples:

```
(red,yellow,green,blue)
(club,diamond,heart,spade)
(married,divorced,widowed,single)
```

7.2.4 Subrange-Types

The syntax for a subrange-type is

```
subrange-type = constant ".." constant
```

Both constants must be of ordinal-type. Both constants must either be of the same ordinal-type, or one must be of type integer and the other of type longint. If both are of the same ordinal-type, this type is called the host-type. If one is of type integer and the other of type longint, the host-type is longint.

Examples:

```
1..100
-10..+10
red..green
```

A variable of subrange-type possesses all the properties of variables of the host type, with the restriction that its

run-time value must be in the specified closed interval.

NOTE: In the current implementation, subranges of the base-type longint will not work correctly if run-time range-checking is enabled. If range-checking is turned off, they work correctly. Range-checking is controlled by the compiler options $R+ and $R- (see Section 15).

7.3 Structured-Types

7.3.1 General

A structured-type is characterized by the type(s) of its components and by its structuring method. If the component type is itself structured, the resulting structured-type exhibits more than one level of structuring. There is no specified limit on the number of levels to which data-types can be structured.

```
structured-type = ["PACKED"] unpacked-structured-type
                |structured-type-identifier
unpacked-structured-type = array-type
                         | set-type
                         | file type
                         | record type
```

The use of the prefix PACKED in the definition of a structured-type indicates to the compiler that storage should be economized, even if this causes an access to a component of a variable of the type to be less efficient in terms of space or time.

An occurrence of the PACKED prefix only affects the representation of the level of the structured-type whose definition it precedes. If a component is itself structured, the component's representation is packed only if the PACKED prefix also occurs in the definition of its type.

See the Pascal Development System Internal Documentation for information on the implementation of packing.

7.3.2 Array-Types

An array-type is a structured-type consisting of a fixed number of components that are all of one type, called the component-type. The number of elements is determined by one or more index-types, one for each dimension of the array. There is no specified limit on the number of dimensions. In each dimension, the array can be indexed by any possible value of the corresponding index-type, so the number of elements is the product of the cardinalities of all the index-types.

```
array-type = "ARRAY" "[" index-type { "," index-type } "]"
             "OF" component-type
index-type = ordinal-type
component-type = type
```

In the current implementation, the index-type should not be longint or a subrange of longint, and arrays should not contain more than 32767 bytes. In implementations that allow arrays of more than 32767 bytes, there willl be a slight performance penalty for accessing such arrays.

Examples:

```
ARRAY[1..100] OF real
ARRAY[Boolean] OF color
```

If the component-type of an array-type is also an array-type, the result is a single multi-dimensional array. In other words, the declaration of an array whose component-type is itself an array-type is equivalent to the declaration of a multi-dimensional array, as illustrated by the following examples.

```
ARRAY[Boolean] OF
        ARRAY[1..10] OF ARRAY[size] OF real
```

is equivalent to:

```
ARRAY[Boolean,1..10,size] OF real
```

and

```
PACKED ARRAY[1..10] OF
    PACKED ARRAY[1..8] OF Boolean
```

is equivalent to:

```
PACKED ARRAY[1..10,1..8] OF Boolean
```

7.3.3 Record-Types

A record-type is a structured-type consisting of a fixed number of components called fields, possibly of different types. For each component, the record-type definition specifies the type of the field and an identifier that denotes it.

The syntax shown below for a record-type permits the specification of a variant-part. This enables different variables, although of identical record-type, to exhibit structures which differ in the number and/or types of their components.

Each variant is introduced by one or more case-constants. All the case-constants must be distinct and must be of an ordinal-type that is compatible with the tag-type (see 7.5, and note that in the current implementation tags of type longint do not work correctly).

When a record is of a type that has a variant part, all fields of all variants are accessible at all times. The variants "overlay" each other in memory.

The variant-part provides for the specification of an optional tag-field. The tag-field is an additional fixed field of the record, and its value may be used by the program to indicate which variant should be used at a given time.

NOTE: In a future edition, a more intelligible discussion of variants will be provided.

```
record-type = "RECORD" [field-list [";"] ] "END"
field-list = fixed-part [ ";" variant-part  ]
           | variant-part
fixed-part = record-section { ";" record-section }
record-section = identifier-list ":" type
variant-part = "CASE" [tag-field ":" ] tag-type "OF"
                 variant { ";" variant }
tag-field = identifier
variant = case-constant-list ":"
            "(" [ field-list [";"] ] ")"
tag-type = ordinal-type-identifier
case-constant-list = case-constant { "," case-constant }
case-constant = constant
field-identifier = identifier
```

NOTE: In the current implementation, the type longint should not be used as a tag-type as it will not work correctly.

Examples:

```
RECORD
  year : integer;
  month : 1..12;
  day : 1..31
END

RECORD
  name, firstname : string[80];
  age : 0..99;
  CASE married : Boolean OF
    true : (spousesname : string[80]);
    false : ()
END

RECORD
  x,y : real;
  area : real;
  CASE s : shape OF
    triangle : ( side : real;
                 inclination, angle1, angle2 : angle);
    rectangle : (side1, side2 : real;
                 skew, angle3 : angle);
    circle : (diameter : real);
END
```

The occurrence of an identifier as a tag-field or within the identifier-list of a record-section is its defining occurrence as a field-identifier for the record-type in which the tag-field or record-section occurs.

7.3.4 Set-Types

A set-type defines the range of values which is the powerset of its base-type.

```
set-type = "SET" "OF" base-type
base-type = ordinal-type (except longint)
```

In the present implementation, the base-type must not have more than 4Ø88 possible values. If the base-type is a subrange of integer, it must be within the limits Ø..4Ø87. In the future, this may be expanded to allow 64K possible values and to allow the base-type to be integer or any subrange of integer.

Operators applicable to sets are specified in section 1Ø.2.4. Section 1Ø.4 shows how set values are denoted in Pascal.

Sets with less than 32 possible values in the base-type can be held in a register and offer the best performance. For sets larger than this, there is a performance penalty that is essentially a linear function of the size of the base-type.

The empty set (see 1Ø.2.4) belongs to every set-type.

7.3.5 File-Types

A file-type is a structured-type consisting of a sequence of components that are all of one type (the component-type). The component-type may be any type except a file-type, or any structured-type that contains a file-type.

The component data is not in program-addressable memory but is accessed via a peripheral device (it may be in a memory area that is accessed like a peripheral device). The number of components (i.e. the length of the file) is not fixed by the file-type definition.

```
file-type = "FILE" ["OF" type]
          | "TEXT"
```

The type file (without the "OF type" construct) represents an "untyped" file for use with the BLOCKREAD and BLOCKWRITE functions (see Apple II Pascal Reference Manual or Apple III Pascal Programmer's Manual).

The standard type text denotes a "textfile". In LISA Pascal, the type text is distinct from the type file of char. Textfiles are discussed in Section 12.

The type file of char is a file whose records are of type char, containing char values that are not interpreted or converted in any way during I/O operations. In a stored file of this type, the char values are packed into bytes. Note that this type is distinct from the type text (unlike UCSD Pascal).

In Lisa Pascal, files can be passed to procedures and functions as variable parameters, as explained in 9.3.3.

7.4 Pointer-Types

A pointer-type consists of an unbounded set of values pointing to variables of a specified type called the base-type.

Pointer values are created by the standard procedure new (see 16.6.1) by the @ operator (see 10.2.6), and by the procedure pointer (see 16.7.4).

```
pointer-type = "^" type-identifier | pointer-type-identifier
```

The pointer value NIL belongs to every pointer type. NIL does not point to a variable. If you access memory via a NIL pointer reference, the results are unspecified; there may not be any error indication.

7.5 Identical and Compatible Types

As explained below, Lisa Pascal has stronger typing than UCSD Pascal (in accordance with the proposed ISO standard). In Lisa Pascal, two types may or may not be identical, and identity is required in some contexts but not in others. Even if not identical, two types may still be compatible, and this is

sufficient in contexts where identity is not required -- except for assignment, where assignment-compatibility is required.

Identical types are required only in the following contexts:

- Variable parameters (see 9.3.3).
- Result types of functional parameters (see 9.3.5).
- Value and variable parameters within parameter-lists of procedural or functional parameters (see 9.3.6).
- One-dimensional packed arrays of char being compared via a relational operator (see 10.2.5).

Compatibility is required in the majority of contexts where two or more entities are used together, e.g. in expressions.

Assignment-compatibility is required whenever one entity is assigned to another, either explicitly or implicitly (as in passing value parameters). Assignment-compatibility is "compatibility," as modified by implicit coercion of certain types and by range-checking.

7.5.1 Type Identity

Types that are defined at two or more different places in the program text are identical if the same type identifier is used at these places, or if different identifiers are used which have been defined to be equivalent to each other by type definitions of the form T1 = T2;.

Note that

```
T1 = T2; T3 = T1;
```

does not make T1 and T3 identical! Also note that

```
T4 = integer; T5 = integer;
```

does make T4 and T5 identical, since both are defined by the same type identifier. In general,

```
T6 = T7; T8 = T7;
```

does make T6 and T8 identical if T7 is a type-identifier. However,

```
T9 =^integer; T1Ø = ^integer;
```

does not make T9 and T1Ø identical since ^integer is not a type identifier but a "constructed type" consisting of the special symbol "^" and a type identifier.

Finally, note that two variables declared in the same declaration, as in

```
VAR1, VAR2: ^integer;
```

are of identical type. However, if the declarations are separate then the definitions above apply. The declarations

```
VAR1: ^integer;
VAR2: ^integer;
VAR3: real;
VAR4: real;
```

imply that VAR3 and VAR4 are identical, but VAR1 and VAR2 are not.

7.5.2 Compatibility of Types

Two types are compatible if any of the following are true:

- They are identical.
- One is a subrange of the other.
- Both are subranges of the same type.
- Both are string types (the lengths and sizes may differ).
- They are set-types of compatible base-types.

7.5.3 Assignment-Compatibility

The value of an expression EXPVAL of type EXPTYP is assignment-compatible with a variable, parameter, or

function-identifier of type VTYP if any of the seven statements which follow is true.

- VTYP and EXPTYP are identical and neither is a file-type nor structured-type with a file component.
- VTYP is a real-type and EXPTYP is integer or longint.
- VTYP and EXPTYP are compatible ordinal-types and EXPVAL is in the closed interval specified by the type VTYP.
- VTYP and EXPTYP are compatible set-types and all the members of EXPVAL are in the closed interval specified by the base-type of VTYP.
- VTYP and EXPTYP are string types and the length of EXPVAL is equal to or less than the size specified by the type VTYP.
- VTYP is a string type or a char type and EXPVAL is a one-character string constant.
- VTYP is a packed array of char with N elements and EXPVAL is a string constant containing exactly N characters. If the index-type of the packed array of char is not 1..N, results are unspecified.

At any place where the rule of assignment-compatibility is used and none of the above is true, either a compiler error or a run-time error occurs.

7.6 Example of a Type-Definition-Part

```
TYPE
  count = integer;
  range = integer;
  colour = (red, yellow, green, blue);
  sex = (male, female);
  year = 19ØØ..1999;
  shape = (triangle, rectangle, circle);
  card = ARRAY[1..8Ø] OF char;
  str = string[8Ø];
  polar = RECORD r : real; theta : angle END;
  person = ^persondetails;
  persondetails = RECORD
    name, firstname : str;
    age : integer;
    married : Boolean;
    father, child, sibling : person;
    CASE s : sex OF
      male : (enlisted, bearded : Boolean);
      female : (pregnant : Boolean)
  END;
  tape = FILE OF persondetails;
  intfile = FILE OF integer;
```

NOTE: In the above examples 'count', 'range' and 'integer' denote identical types. The type 'year' is compatible with, but not identical to, the types 'range', 'count' and 'integer'.

Section 8

DECLARING AND REFERENCING VARIABLES

8.1 Variable-Declarations

A variable declaration consists of a list of identifiers denoting the new variables, followed by their type.

variable-declaration = identifier-list ":" type

The occurrence of an identifier within the identifier-list of a variable-declaration is its defining occurrence as a variable-identifier for the block in which the declaration occurs. A variable declared in a variable-declaration exists during the entire execution process of the block in which it is declared, except as specified in 5.2.2.

Examples:

```
x,y,z: real;
i,j: integer;
k: Ø..9;
p,q,r: Boolean;
operator: (plus, minus, times);
a: ARRAY[Ø..63] OF real;
c: color;
f: FILE OF char;
hue1,hue2: SET OF color;
p1,p2: person;
m,m1,m2 : ARRAY[1..1Ø,1..1Ø] OF real;
coord : polar;
pooltape : ARRAY[1..4] OF tape;
```

8.2 Variable-References

A variable-reference denotes the value of a variable of simple-type or pointer-type, or the collection of values represented by a variable of structured-type. (Note that if the variable is of simple-type or pointer-type, it may be a component of another variable of structured-type.)

```
variable-reference = variable-identifier
                   | file-buffer
                   | pointer-reference
                   | variable-reference selector
variable-identifier = identifier
```

Syntax for file-buffers, pointer-references, and selectors is given below.

8.3 Selectors and Components

8.3.1 General

A component of an array or record is denoted by a variable-reference that refers to the array or record, followed by a selector that specifies the component.

```
selector = index | field-designator
```

If the variable is an array or a string, the selector is an index. If the variable is a record, the selector is a field-designator. Components of files are referenced via file-buffers (see 8.3.4).

Note that a component of a set cannot be directly referenced.

8.3.2 Arrays, Strings, and Indexes

A component of a variable of array-type or string-type is denoted by a variable-reference that refers to the array or string, followed by an index.

```
index = "[" expression {"," expression} "]"
```

Examples:

```
m[i,j]
a[i+j]
```

Each expression in the index selects a component in the corresponding dimension of the array. The number of expressions must not exceed the number of index-types in the array declaration, and the type of each expression must be

assignment-compatible with the corresponding index-type.

In indexing a multi-dimensional array, you can use either multiple indexes or multiple expressions within an index. The two forms are completely equivalent, as shown by the following example.

```
     m[i][j]
is equivalent to
     m[i,j]
```

A string can be indexed by only one index expression, whose value must be in the range 1..n where n is the dynamic length of the string variable. For array variables, each index expression must be assignment-compatible with the corresponding index-type specified in the definition of the array-type.

8.3.3 Records and Field-Designators

A component of a variable of record-type is denoted by a variable-reference that refers to the structured record, followed by a field-designator that specifies the component.

```
field-designator = "." field-identifier
field-identifier = identifier
```

Example:

```
p2^.pregnant
coord.theta
```

8.3.4 File-Buffers

Although a file variable may have any number of components, only one component is accessible at any time. The position of the current component in the file is called the "current file position." See 16.2 for standard procedures that move the current file position. Program access to the current component is via a variable called a "file-buffer."

The file-buffer is implicitly declared when the file variable is declared. If F is a file variable with components of type T, the associated file-buffer is a variable of type T.

The file-buffer associated with a particular file variable is referenced by a variable-reference that refers to the file variable, followed by an up-arrow. Thus the file-buffer of file F is referenced by F^.

```
file-buffer = file-variable-reference up-arrow
file-variable-reference = variable-reference
up-arrow = "^"
```

Section 16.2 describes standard procedures that are used to move the current file position within the file and to transfer data between the file-buffer and the current file component.

8.4 Pointer-References

In addition to declared variables and file-buffers, Pascal provides for dynamically allocated variables. A dynamically allocated variable is created by the standard procedure new (see 16.6.1) which takes as its argument a pointer variable; this pointer variable becomes a pointer to the dynamically allocated variable.

Since a dynamically allocated variable has no identifier of its own, it is referenced by a variable-reference that refers to its pointer, followed by an up-arrow. Thus if P refers to a variable of pointer-type, the variable that P points to is referenced by P^. This is called a pointer-reference.

```
pointer-reference = pointer-variable up-arrow
pointer-variable = variable-reference
up-arrow = "^"
```

Examples:

```
p1^
p1^.sibling^
```

Section 9

PROCEDURES AND FUNCTIONS

9.1 Procedure-Declarations

A procedure-declaration associates an identifier with part of a program so that it can be activated by a procedure-statement.

```
procedure-declaration = procedure-heading ";" body
body = procedure-block
     | "FORWARD"
     | "EXTERNAL"
procedure-block = block
```

The procedure-heading specifies the identifier naming the procedure, and the formal parameters (if any). The appearance of an identifier in the procedure-heading of a procedure is its defining occurrence as a procedure-identifier for the block in which the procedure-declaration occurs.

```
procedure-heading = "PROCEDURE" identifier
                        [ formal-parameter-list ]
procedure-identifier = identifier
```

The syntax for a formal-parameter-list is given in 9.3.1.

The algorithmic actions to be executed upon activation of the procedure by a procedure-statement are specified by the statement-part of the procedure-block. The use of the procedure-identifier in a procedure-statement within the procedure-block implies recursive execution of the procedure.

An example of a procedure declaration is:

```
PROCEDURE readinteger (VAR f: text; VAR x: integer) ;
  VAR value,digitvalue: integer;
  BEGIN
    WHILE (f^ = ' ') AND NOT eof(f) DO get(f);
    value := Ø;
    WHILE (f^ IN ['Ø'..'9']) AND NOT eof(f) DO
      BEGIN digitvalue := ord(f^) - ord('Ø');
        value := 1Ø*value + digitvalue;
        get(f)
      END;
    x := value
  END;
```

A procedure-declaration that has "FORWARD" instead of a procedure-block is called a forward declaration. Somewhere after the forward declaration, the procedure is actually defined by a procedure-declaration that uses the same procedure-identifier, omits the formal-parameter-list, and includes a procedure-block. The forward declaration and the subsequent actual definition must be local to the same block, but need not be contiguous; that is, other procedures or functions can be declared between them and can call the procedure that has been declared forward. This permits mutual recursion.

The forward declaration and the subsequent procedure-declaration that actually defines the procedure constitute a defining occurrence at the place of the forward declaration.

A procedure-declaration that has "EXTERNAL" instead of a procedure-block defines the Pascal interface to a separately assembled or compiled routine (a .PROC in the case of assembly language). The external code must be linked with the compiled Pascal host program before execution; see the Lisa Pascal Development System manual for details.

An example of an external procedure definition is

```
PROCEDURE MAKESCREEN(INDEX: INTEGER);
  EXTERNAL;
```

This means that MAKESCREEN is an external procedure that will be linked to the host program before execution.

It is the programmer's responsibility to ensure that the external procedure is compatible with the EXTERNAL definition in the Pascal program; the current Linker does no checking.

This description of external procedures also applies to external functions.

Note that Lisa Pascal (unlike UCSD Pascal) does not allow a variable parameter of an external procedure or function to be declared without a type. To obtain a similar effect, use a formal-parameter of pointer-type, as in the following example:

```
TYPE BIGPAOC = PACKED ARRAY[Ø..32767] OF CHAR;
     BIGPAOCPTR = ^BIGPAOC;

...

PROCEDURE WHATEVER (BYTEARRAY: BIGPAOCPTR);
  EXTERNAL;
```

The actual-parameter can be any pointer value obtained via the @-operator (see 1Ø.2.6). For example, if DOTS is a packed array of boolean, it can be passed to WHATEVER by writing

```
WHATEVER(@DOTS)
```

9.2 Function-Declarations

A function-declaration serves to define a part of the program that computes a value of simple-type or a pointer value. A function is activated by the evaluation of a function-call (see 1Ø.3); function-calls appear as operands in expressions.

```
function-declaration = function-heading ";" body
body = function-block
     | "FORWARD"
     | "EXTERNAL"
function-block = block
```

The function-heading specifies the identifier naming the function, the formal parameters (if any), and the type of the function result. The appearance of an identifier in the function-heading of a function-declaration is its defining occurrence as a function-identifier for the block in which the function-declaration occurs.

```
function-heading = "FUNCTION" identifier
                   [formal-parameter-list] ":" result-type
function-identifier = identifier
result-type = simple-type-identifier
            | pointer-type-identifier
```

The algorithmic actions to be executed upon activation of the function by a function-call are specified by the statement-part of the function-block. The function-block should normally contain at least one assignment-statement that assigns a value to the function-identifier. The result of the function is the last value assigned. If no such assignment-statement exists, or if it exists but is not executed, the value returned by the function is unspecified.

The syntax for a formal-parameter-list is given in 9.3.1.

The use of the function-identifier in a function-call within the function-block implies recursive execution of the function.

A function-declaration that has "FORWARD" instead of a function-block is called a forward declaration. Somewhere after the forward declaration, the function is actually defined by a function-declaration that uses the same function-identifier, omits the formal-parameter-list and result-type, and includes a function-block. The forward declaration and the subsequent actual definition must be local to the same block, but need not be contiguous; that is, other procedures or functions can be declared between them and can call the function that has been declared forward. This permits mutual recursion.

The forward declaration and the subsequent function-declaration that actually defines the function constitute a defining occurrence at the place of the forward declaration.

Examples:

```
FUNCTION GCD(m,n : integer) : integer; forward;
```

```
FUNCTION max(a: vector; n: integer): real;
  VAR x: real; i: integer;
  BEGIN
    x := a[1];
    FOR i := 2 TO n DO BEGIN
      IF x < a[i] THEN x := a[i]
    END;
    max := x
  END;

FUNCTION GCD; {which has been forward declared}
  BEGIN
    IF n=Ø THEN GCD := m ELSE GCD := GCD(n,m MOD n)
  END;

FUNCTION Power(x: real;y: integer): real ; { y >= Ø}
  VAR w,z: real; i: integer;
  BEGIN
    w := x; z := 1; i := y;
    WHILE i > Ø DO BEGIN
      {z*(w**i) = x ** y }
      IF odd(i) THEN z := z*w;
      i := i div 2;
      w := sqr(w)
    END;
    {z = x**y }
    Power := z
  END;
```

A function-declaration that has "EXTERNAL" instead of a function-block defines the Pascal interface to a separately compiled or assembled external routine. See the explanation in 9.1 above.

9.3 Parameters

9.3.1 General

There are four kinds of parameters: value parameters, variable parameters, procedural parameters, and functional parameters.

```
formal-parameter-list = "(" parameter-section
                          {";" parameter-section} ")"
parameter-section = ["VAR"] parameter-group
                  | procedure-heading
                  | function-heading
parameter-group = identifier-list ":" type-identifier
parameter-identifier = identifier
```

A parameter-group preceded by "VAR" is a list of variable parameters. A parameter-group without a preceding "VAR" is a list of value parameters. A procedure-heading or function-heading denotes a procedural or functional parameter; see 9.3.4 and 9.3.5 below.

The occurrence of an identifier within the identifier-list of a parameter-group is its defining occurrence as a parameter-identifier for the formal-parameter-list in which it occurs and any corresponding procedure-block or function-block.

The occurrence of an identifier in a procedure-heading or function-heading within a parameter-section (as the name of the function or procedure) is its defining occurrence as a procedural or functional parameter for the formal-parameter-list in which it occurs and any corresponding procedure-block or function-block.

If the formal-parameter-list is part of the definition of a procedural or functional parameter, there must be no corresponding procedure-block or function-block.

Note that the types of formal-parameters are denoted by type-identifiers. In other words, only a simple identifier can be used to denote a type in a formal-parameter-list. To use a type such as ARRAY[Ø..255] OF CHAR as the type of a parameter, you must declare a type-identifier for this type:

```
TYPE CHARRAY = ARRAY[Ø..255] OF CHAR;
```

The identifier CHARRAY can then be used in a formal-parameter-list to denote the type.

NOTE: The identifier FILE (for an untyped file) is not allowed as a parameter type. To use a parameter of this type, declare some other identifier for the type FILE -- for example,

```
TYPE PHYLE = FILE;
```

The identifier PHYLE can then be used in a formal-parameter-list to denote the type FILE.

9.3.2 Value Parameters

The actual-parameter (see 1Ø.3 and 11.2.3) must be an expression, and its value must not be of file-type. The formal parameter denotes a variable local to the block. The current value of the expression is assigned to the variable upon activation of the block. The actual-parameter must be assignment-compatible with the type of the formal parameter.

9.3.3 Variable Parameters

The actual-parameter (see 1Ø.3 and 11.2.3) must be a variable-reference. The formal parameter denotes this actual variable during the entire activation of the block. Any operation within the block, involving the formal parameter, is performed on the actual-parameter. The type of the actual parameter must be identical to that of the formal parameter. If the selection of this variable involves the indexing of an array, or the de-referencing of a pointer, these actions are executed before the activation of the block.

Components of variables of any packed type (including string-types) cannot be used as actual variable parameters.

9.3.4 Procedural Parameters

When the formal parameter is a procedure-heading, the actual-parameter (see 1Ø.3 and 11.2.3) must be a procedure-identifier. The identifier given in the procedure-heading represents the actual procedure during the entire activation of the block as shown in the following example:

```
PROGRAM Pass_Proc;
  VAR i: INTEGER;

  PROCEDURE a(PROCEDURE x) {x is a formal procedural
                            parameter.}
    BEGIN
      WRITE('About to call x ');
      x                     {call procedure passed as
                             parameter}
    END;

  PROCEDURE b;
    BEGIN
      WRITE('In procedure b')
    END;

  FUNCTION c(PROCEDURE x): INTEGER;
    BEGIN
      x;                    {call procedure passed as
                             parameter}
      c:=2
    END;

  BEGIN
    a(b);                   {call a, passing b as parameter}
    i:= c(b)                {call c, passing b as parameter}
  END.
```

The actual procedure and the formal procedure must have compatible formal-parameter-lists (see 9.3.6). Regardless of the formal-parameter-list of the actual procedure, only the identifier of the actual procedure is passed as shown in the following example:

```
PROGRAM Test;
  VAR i: INTEGER;

  PROCEDURE X_as_Par(y: INTEGER);
    BEGIN
      WRITELN('y=', y)
    END;

  PROCEDURE Call_Proc(PROCEDURE x_again(z: INTEGER));
    VAR b: BOOLEAN;
    BEGIN
      b:= TRUE;
      x_again(1)
    END;

  BEGIN
    i:=2;
    Call_P o (X_as_Par)
  END.
```

If the procedural parameter, upon activation, accesses any non-local entity (by identifier, pointer reference, or label), the entity accessed must be one that was accessible to the procedure when its procedure-identifier was passed as a procedural parameter.

To see what this means, consider a procedure PP which is known to another procedure, FIRSTPASSER. Suppose that the following sequence takes place:

- FIRSTPASSER is executing.
- FIRSTPASSER calls a procedure named RCVR1, passing PP as a procedural parameter.
- RCVR1 calls RCVR2, again passing PP as a procedural parameter.
- RCVR2 calls PP (first execution of PP).
- RCVR2 calls RCVR3, again passing PP as a procedural parameter.
- RCVR3 calls FIRSTPASSER (indirect recursion), and passes PP to FIRSTPASSER as a procedural parameter.

- FIRSTPASSER (executing recursively) calls PP (second execution of PP).

Thus the procedure PP is called first from RCVR2, and then from the second (recursive) execution of FIRSTPASSER.

Suppose that PP accesses an entity named XXX, which is not local to PP; and suppose that each of the other procedures has a local entity named XXX.

Each time PP is called, which XXX does it access? The answer is that in each case, PP accesses the XXX that is local to the first execution of FIRSTPASSER -- that is, the XXX that was accessible when PP was originally passed as a procedural parameter.

9.3.5 Functional Parameters

When the formal parameter is a function-heading, the actual-parameter (see 10.3 and 11.2.3) must be a function-identifier. The identifier given in the function-heading represents the actual function during the entire activation of the block.

Functional parameters are exactly like procedural parameters, with the additional rule that corresponding formal and actual functions must have identical result-types.

9.3.6 Parameter List Compatibility

Parameter list compatibility is required of the parameter lists of corresponding formal and actual procedural or functional parameters.

Two formal-parameter-lists are compatible if they contain the same number of parameters and if the parameters in corresponding positions match. Two parameters match if:

- They are both value parameters of identical type;
- Or they are both variable parameters of identical type;
- Or they are both procedural parameters with compatible parameter lists;

- Or they are both functional parameters with compatible parameter lists and identical result-types.

Section 10

EXPRESSIONS

10.1 General

Expressions consist of operators and operands, i.e. variables, constants, set-values, and function calls. Operator precedence is as follows: The @-operator and the not-operator have the highest procedence, followed by the multiplying-operators, then the adding-operators and signs, and finally, with the lowest precedence, the relational-operators. A left-to-right rule is used to break ties between two or more operators of the same precedence.

```
unsigned-constant = unsigned-number
                  | quoted-string-constant
                  | constant-identifier
                  | "NIL"
factor = [@-operator] variable
       | unsigned-constant
       | function-call
       | set-constructor
       | "(" expression ")"
       | not-operator factor
term = factor { multiplying-operator factor }
simple-expression = [ sign ] term { adding-operator term }
sign = "+" | "-"
expression = simple-expression
             [ relational-operator simple-expression ]
```

Examples are:

(a) Factors:

```
x
15
(x+y+z)
NOT p
```

(b) Terms:

```
x*y
i/(1-i)
p OR q
(x <= y) AND (y < z)
```

(c) Simple expressions:

```
x+y
-x
huel +
i*j + 1
```

(d) Expressions:

```
x = 1.5
p <= q
p = q A
(i < j)
c IN huel
```

A function-call activates a function, and denotes the value returned by the function (see 10.3). A set-constructor denotes a value of a set-type (see 10.4).

10.2 Operators

10.2.1 Syntax

```
multiplying-operator = "*" | "/" | "DIV" | "MOD" | "AND"
adding-operator = "+" | "-" | "OR"
relational-operator = "=" | "<>" | "<" | ">" | "<="
                    | ">=" | "IN"
@-operator = "@"
not-operator = "NOT"
```

The order of evaluation of the operands of a binary operator is unspecified.

10.2.2 Arithmetic Operators

The types of operands and results for binary and unary operations are as shown in tables 2 and 3 respectively.

Table 2. Binary Operations

operator	operation	type of operands	type of result
+	addition	integer, real, or longint	integer, real, or longint
-	subtraction	integer, real, or longint	integer, real, or longint
*	multiplication	integer, real, or longint	integer, real, or longint
/	division	integer, real, or longint	real
DIV	division with truncation	integer or longint	integer or longint
MOD	modulo	integer or longint	integer or longint

Table 3. Unary Operations

operator	operation	type of operand	type of result
+	identity	integer, real, or longint	same
-	sign-inversion	integer, real, or longint	same
NOT	negation	boolean	boolean
@	pointer formation	variable, procedure, or function	same as NIL

NOTE: The symbols +, - and * are also used as set operators (see 10.2.4).

Any operand whose type is SUBR, where SUBR is a subrange of some ordinal-type ORDTYP, is treated as if it were of type ORDTYP. Consequently an expression that consists of a single operand of type SUBR is itself of type ORDTYP.

If both the operands of the addition, subtraction, or multiplication operators are of the type integer or longint, the result is of the type integer or longint as described in 7.2.2.2, but if otherwise, the result is of the type real.

The result of the identity or sign-inversion operator is of the same type as the operand.

The result of the @-operator is a pointer value of the same type as NIL, i.e., it is compatible with any pointer-type.

The value of i DIV j is the mathematical quotient of i/j, rounded toward zero. An error occurs if j=Ø.

The value of i MOD j is equal to the value of

    i - (i DIV j)*j.

Note that the sign of the result of MOD is always the same as the sign of i. An error occurs if j=Ø.

The predefined constant maxint is of type integer. Its value is 32767. This value satisfies the following conditions:

- All integral values in the closed interval from -maxint to +maxint are representable in the type integer.
- Any unary operation performed on an integer value in this interval will be correctly performed according to the mathematical rules for integer arithmetic.
- Any binary integer operation on two integer values in this same interval will be correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in this interval. If the result is not in this interval, it is truncated to 16 bits.
- Any relational operation on two integer values in this same interval will be correctly performed according to the mathematical rules for integer arithmetic.

10.2.3 Boolean Operators

The types of operands and results for Boolean operations are shown in table 4.

Table 4. Boolean Operations

operator	operation	type of operands	type of result
OR	logical "or"	Boolean	Boolean
AND	logical "and"	Boolean	Boolean
NOT	logical negation	Boolean	Boolean

Whether a Boolean expression is completely or partially evaluated if its value can be determined by partial evaluation is unspecified.

10.2.4 Set Operators

The types of operands and results for set operations are shown in table 5.

Table 5. Set Operations

operator	operation	type of operands	type of result
+	union	compatible set-types	(see 10.2.4.1)
-	difference	compatible set-types	(see 10.2.4.1)
*	intersection	compatible set-types	(see 10.2.4.1)

10.2.4.1 Result Type in Set Operations

The following rules govern the type of the result of a set operation where one (or both) of the operands is a SET OF SUBR, where ORDTYP represents any ordinal-type and SUBR represents a

subrange of ORDTYP:

- If ORDTYP is not the type INTEGER, then the type of the result is SET OF ORDTYP.

- If ORDTYP is the type INTEGER, then the type of the result is SET OF Ø..4Ø87 in the current implementation (Ø..32767 in a future implementation). This rule results from the limitations on set-types (see 7.3.4).

1Ø.2.5 Relational Operators

The types of operands and results for relational operation are shown in table 6.

Table 6. Relational Operations

operator	type of operands	type of result
= <>	compatible set-, simple-, or pointer-types (see note below)	Boolean
< >	any simple-type	Boolean
<= >=	any set-type or simple-type	Boolean
IN	left operand: any ordinal-type T; right operand: SET OF T	Boolean

NOTE: In addition to the operand types shown in the table, the = and <> operators can also be used to compare a packed array[1..N] of char with a string constant containing exactly N characters, or to compare two one-dimensional packed arrays of char of identical type.

Except when applied to sets, the operators <> , <= , and >= stand for 'not equal', 'less than or equal' and 'greater than or equal' respectively. The operands of =, <>, <, >, >=, and <= must be either of compatible type, or one operand must be real and the other must be integer or longint. If u and v are set operands,

```
u <= v  denotes the inclusion of u in v
u >= v  denotes the inclusion of v in u
```

If p and q are Boolean operands, p = q denotes their equivalence and p <= q denotes the implication of q by p, because false < true. Similarly, p <> q denotes logical "exclusive-or." When the relational operators = , <> , < , > , <= , and > are used to compare strings (see 7.2.2.6), they denote lexicographic ordering according to the ordering of the ASCII character set. The string operands need not be of compatible type; any two string values may be compared.

The operator IN yields the value true if the value of the operand of ordinal-type is a member of the set; otherwise it yields the value false.

10.2.6 @-Operator

A pointer to a variable can be computed with the @-operator. @ is a unary operator taking a single variable, parameter, procedure, or function as its operand and computing the value of its pointer. The type of the value is equivalent to the type of NIL, and consequently can be assigned to any pointer variable.

10.2.6.1 @-Operator With a Variable

For an ordinary variable (not a parameter), the use of the @-operator is straightforward. For example, if we have the declarations

```
INT: integer;
TWOCHARPTR: ^packed array[0..1] of char;
```

then the statement

```
TWOCHARPTR:=@INT
```

causes PTR to point to INT. Now TWOCHARPTR^ is a reinterpretation of the bit value of INT as though it were a packed array[0..1] of char.

The operand of @ cannot be an element of a packed variable.

10.2.6.2 @-Operator With a Value Parameter

When the @-operator is applied to a formal value parameter, the result is a pointer to the stack location containing the actual value. Suppose that FOO is a formal value parameter in a procedure and FOOPTR is a pointer variable. If the procedure executes the statement

```
FOOPTR:=@FOO
```

then FOOPTR^ is a reference to the value of FOO. Note that if the actual-parameter was a variable-reference, FOOPTR^ is not a reference to the variable itself; it is a reference to the value taken from the variable and stored on the stack.

10.2.6.3 @-Operator With a Variable Parameter

When the @-operator is applied to a formal variable parameter, the result is a pointer to the actual-parameter (the pointer is taken from the stack). Suppose that FUM is a formal variable parameter of a procedure, FIE is a variable passed to the procedure as the actual-parameter for FUM, and FUMPTR is a pointer variable. If the procedure executes the statement

```
FUMPTR:=@FUM
```

then FUMPTR is a pointer to FIE. FUMPTR^ is a reference to FIE itself.

10.2.6.4 @-Operator With a Procedure or Function

It is possible to apply the @-operator to a procedure or a function, yielding a pointer to the entry-point. Note that Pascal provides no mechanism for using such a pointer. Currently the only use for a procedure pointer is to pass it to an assembly-language routine, which can then JSR to that address.

If the procedure pointed to is in the local segment, @ returns the current address of the procedure's entry point. If the procedure is in some other segment, however, the @-operator returns the address of the jump table entry for the procedure.

In logical memory mapping (see Lisa Pascal Development System Manual), the procedure pointer is always valid.

In physical memory mapping, code swapping may change a local-segment procedure address without warning, and the procedure pointer can become invalid. If the procedure is not in the local segment, the jump-table entry address will remain valid despite swapping because the jump table is not moved.

10.3 Function-Calls

A function-call specifies the activation of the function denoted by the function-identifier. If the corresponding function-declaration contains a list of formal-parameters, then the function-call must contain a corresponding list of actual-parameters. Each actual-parameter is substituted for the corresponding formal-parameter. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters must be equal to the number of formal parameters.

The order of evaluation and binding of the actual-parameters is unspecified.

```
function-call = function identifier
                  [ actual-parameter-list ]
function-identifier = identifier
actual-parameter-list = "(" actual-parameter
                          { "," actual-parameter } ")"
actual-parameter = expresssion
                 | variable-reference
                 | procedure-identifier
                 | function-identifier
```

Examples of function-calls:

```
Sum(a,63)
GCD(147,k)
sin(x+y)
eof(f)
ord(f^)
```

10.4 Set-Constructors

A set-constructor denotes a value of a set-type, and is formed by writing expressions within [brackets].

```
set-constructor = "[" [ member-group
                      { "," member-group } ] "]"
member-group = expression [ ".." expression ]
```

The notation [] denotes the empty set, which belongs to every set-type. Any member-group x..y denotes as set members the range of all values of the base-type in the closed interval x to y.

If x is greater than y, then x..y denotes no members and [x..y] denotes the empty set.

All values designated in member-groups in a particular set-constructor must be of the same ordinal-type. This ordinal-type is the base-type of the resulting set. If an integer value designated as a set member is outside the limits given in 7.3.4 (0..4087 in the current implementation), the results are unspecified.

Examples of set-constructors:

```
[red, c, green]
[1, 5, 10..K MOD 12, 23]
['A'..'Z', 'a'..'z', chr(xcode)]
```

Section 11

STATEMENTS

11.1 General

Statements denote algorithmic actions, and are executable. They can be prefixed by labels, which can be referenced by goto statements.

```
statement = [[ label ":" ] ( simple-statement
                           | structured-statement )]
label = unsigned-integer
```

An unsigned-integer used as a label must be in the range Ø..9999.

11.2 Simple Statements

11.2.1 General

A simple-statement is a statement of which no part constitutes another statement. An empty statement consists of no symbols and denotes no action.

```
simple-statement = assignment-statement
                 | procedure-statement
                 | goto-statement
```

11.2.2 Assignment-Statements

The assignment-statement serves to replace the current value of a variable by a new value specified as an expression.

```
assignment-statement = ( variable | function-identifier )
                       ":=" expression
```

The expression must be assignment-compatible with the type of the variable or function. If the selection of the variable involves the indexing of an array or the de-referencing of a pointer, the decision whether these actions precede or follow the evaluation of the expression is unspecified.

Examples:

```
      x := y+z;
      p := (1<=i) AND (i<100);
      i := sqr(k) - (i*j);
   huel := [blue,succ(c)];
```

11.2.3 Procedure-Statements

A procedure-statement serves to execute the procedure denoted by the procedure-identifier. If the procedure has formal-parameters (see 9.3.1), the procedure-statement must contain a list of actual-parameters that are substituted in place of the corresponding formal parameters. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters must be equal to the number of formal parameters. The order of evaluation and binding of the actual parameters is unspecified.

```
procedure-statement = procedure-identifier
                        [ actual-parameter-list ]
procedure-identifier = identifier
```

Examples:

```
printheading;
transpose(a,n,m);
bisect(fct,-1.0,+1.0,x);
```

11.2.4 GOTO Statements

A goto-statement serves to indicate that further processing is to continue at another part of the program text, namely at the place of the label.

```
goto-statement = "GOTO" label
```

The following restrictions govern the use of labels:

- A goto-statement cannot refer to a case-constant, since a case-constant is not a label.

- The effect of a jump into a structured statement from outside of the structured statement is unspecified.

- The effect of a jump between the THEN part and the ELSE part of an IF statement is unspecified.

- The effect of a jump between two different cases within a CASE statement is unspecified.

11.3 Structured-Statements

11.3.1 General

Structured-statements are constructs composed of other statements that must be executed either in sequence (compound-statement), conditionally (conditional-statements), repeatedly (repetitive-statements), or within an expanded scope (with-statements).

```
structured-statement = compound-statement
                     | conditional-statement
                     | repetitive-statement
                     | with-statement
```

11.3.2 Compound-Statements

The compound-statement specifies that its component statements are to be executed in the same sequence as they are written. The symbols BEGIN and END act as statement brackets.

```
compound-statement = "BEGIN" [statement { ";" statement }]
                     "END"
```

Example:

```
BEGIN z := x ; x := y; y := z END
```

11.3.3 Conditional-Statements

A conditional-statement selects for execution a single one of its component statements.

```
conditional-statement = if-statement
                      | case-statement
```

11.3.3.1 IF-Statements

```
if-statement = "IF" expression "THEN" statement
                 [ else-part ]
else-part = "ELSE" statement
```

The expression must yield a result of type Boolean. If the expression yields the value true, the statement following the THEN is executed. If the Boolean-expression yields false and the else-part is present, the statement following the ELSE is executed, otherwise an empty statement is executed.

The syntactic ambiguity arising from the construct:

```
IF e1 THEN IF e2 THEN s1 ELSE s2
```

is resolved by interpreting the construct as being equivalent to:

```
IF e1 THEN
  BEGIN
    IF e2 THEN s1 ELSE s2
  END
```

Examples:

```
IF x < 1.5 THEN z := x+y ELSE z := 1.5;
IF p1 <> NIL THEN p1 := p1^.father;
```

11.3.3.2 CASE-Statements

The case-statement contains an expression (the selector) and a list of statements. Each statement must be preceded by one or more constants. All the case-constants must be distinct and must be of an ordinal-type that is compatible with the type of the selector. The case-statement specifies execution of the statement whose case-constant is equal to the current value of the selector. If no such case-constant exists and an otherwise-part is present, the statement following the OTHERWISE is executed; else nothing is executed and control passes to the statement following the case-statement.

```
case-statement = "CASE" expression "OF"
                   case-list-element
                   {";" case-list-element }
                   [";" "OTHERWISE" statement ]
                   [";"] "END"
case-list-element = case-constant-list ":" statement
case-constant-list = constant {"," constant}
```

Examples:

```
CASE operator OF
  plus:   x := x+y;
  minus:  x := x-y;
  times:  x := x*y
END

CASE i OF
  1:         x := sin(x);
  2:         x := cos(x);
  3,4,5:     x := exp(x);
  OTHERWISE x := ln(x)
END
```

NOTE: In the current implementation, the case-statement will not work correctly if any case-constant is of type longint or the value of the selector is of type longint.

11.3.4 Repetitive-Statements

Repetitive-statements specify that certain statements are to be executed repeatedly.

```
repetitive-statement = while-statement
                     | repeat-statement
                     | for-statement
```

11.3.4.1 REPEAT-Statements

```
repeat-statement = "REPEAT" [statement { ";" statement }]
                   "UNTIL" expression
```

The expression must yield a result of type Boolean. The statements between the symbols REPEAT and UNTIL are repeatedly executed until the expression yields the value true on completion

of the sequence of statements. The sequence of statements is executed at least once, because the expression is evaluated after execution of the sequence.

Examples:

```
REPEAT
  k := i MOD j;
  i := j;
  j := k
UNTIL j = Ø

REPEAT
  process(f^);
  get(f)
UNTIL eof(f)
```

11.3.4.2 WHILE Statements

```
while-statement = "WHILE" expression "DO" statement
```

The expression must yield a result of type Boolean. The statement is repeatedly executed while the expression yields the value true. If its value is false at the beginning, the statement is not executed.

The while-statement:

```
WHILE b DO body
```

is equivalent to:

```
IF b THEN REPEAT
  body
UNTIL NOT b
```

Examples:

```
WHILE a[i] <> x DO i := i+1

WHILE i>Ø DO BEGIN
  IF odd(i) THEN z := z*x;
  i := i DIV 2;
  x := sqr(x)
END
```

```
WHILE NOT eof(f) DO BEGIN
  process(f^ );
  get(f)
END
```

11.3.4.3 FOR-Statements

The for-statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the control-variable of the for-statement.

```
for-statement = "FOR" control-variable ":=" initial-value
                ( "TO | "DOWNTO" ) final-value
                "DO" statement
control-variable = variable-identifier
initial-value = expression
final-value = expression
```

The control-variable must be a variable-identifier (without subscript or field-identifier). It must be local to the innermost block containing the for-statement, and must not be a variable parameter of that block. The control-variable must be of ordinal-type, and the initial and final values must be of a type compatible with this type. If the value of the control-variable is altered by the repeated statement or by any procedure or function activated by the repeated statement, the effect is unspecified. After a for-statement is executed, the value of the control-variable is unspecified, unless the for-statement was exited by a goto. Apart from these restrictions, the for-statement:

```
FOR v := e1 TO e2 DO body
```

is equivalent to:

```
BEGIN
  temp1 := e1;
  temp2 := e2;
  IF temp1 <= temp2 THEN BEGIN
    v := temp1;
    body;
    WHILE v <> temp2 DO BEGIN
      v := succ(v);
      body
    END
  END
END
```

and the for-statement:

```
FOR v := e1 DOWNTO e2 DO body
```

is equivalent to:

```
BEGIN
  temp1 := e1;
  temp2 := e2;
  IF temp1 >= temp2 THEN BEGIN
    v := temp1;
    body;
    WHILE v <> temp2 DO BEGIN
      v := pred(v);
      body
    END
  END
END
```

where temp1 and temp2 are auxiliary variables of the host type of the variable v that do not occur elsewhere in the program.

Examples of for-statements are:

```
FOR i := 2 TO 63 DO IF a[i] > max THEN max := a[i]
```

```
FOR i := 1 TO n DO FOR j := 1 TO n DO BEGIN
  x := Ø;
  FOR k := 1 TO n DO
    x := x + m1[i,k]*m2[k,j];
  m[i,j] := x
END

FOR c := red TO blue DO q(c)
```

11.3.5 WITH-Statements

```
with-statement = "WITH" record-variable-list "DO"
                    statement
record-variable-list = record-variable
                          { "," record-variable }
record-variable = variable-reference
```

The occurrence of a record-variable in the record-variable-list is a defining occurrence of its field-identifiers as variable-identifiers for the with-statement in which the record-variable-list occurs.

The statement:

```
WITH v1,v2, ....vn DO s
```

is equivalent to:

```
WITH v1 DO
  WITH v2 DO
    ....
      WITH vn DO s
```

Example:

```
WITH date DO IF month = 12 THEN BEGIN
  month := 1;
  year := year + 1
END
ELSE month := month + 1
```

is equivalent to:

```
IF date.month = 12 THEN BEGIN
  date.month := 1;
  date.year := date.year+1
END
ELSE date.month := date.month+1
```

If the selection of a variable in the record-variable-list involves the indexing of an array or the de-referencing of a pointer, these actions are executed before the component statement is executed.

WARNING: If a variable in the record-variable-list is a pointer-reference, the value of the pointer must not be altered within the WITH statement. If the value of the pointer is altered, the results are unspecified. Example:

```
WITH PPP^ DO BEGIN
  ...
  NEW(PPP);  {Don't do this ...}
  ...
  PPP:=XXX;  {... or this}
  ...
END
```

Section 12

TEXTFILE INPUT AND OUTPUT

This section describes input and output with files of the standard type TEXT. Note that in LISA Pascal, the type TEXT is distinct from FILE OF CHAR (see 7.3.5).

12.1 General

Textfiles are a special kind of file. A textfile represents a sequence of characters, usually formatted into lines by CR characters (ASCII 13). The console keyboard and screen appear to a Pascal program to be built-in textfiles named INPUT and OUTPUT respectively. Other interactive devices can also be represented in Pascal programs as files of type TEXT; when an interactive device is accessed, a "lazy evaluation" method is used for gets, puts, and references to the file buffer as described in 16.2.

When a textfile is stored (e.g. on diskette), it contains information other than the actual sequence of characters represented.The stored file is a sequence of 1Ø24-byte "pages," each containing some number of complete lines of text and padded with null characters. Two 512-byte "header blocks" may also be present at the beginning of the file. Also, a sequence of spaces in the text may be compressed into a two-byte code, namely a DLE character (ASCII 1Ø) followed by a byte containing 32 + the number of spaces represented. All of this special formatting is invisible to a Pascal program if the file is accessed via a file variable of type TEXT (but visible via a file variable of type FILE OF CHAR).

NOTE: The header blocks at the beginning of textfiles will be abolished at some time in the future.

Certain things that can be done with a FILE OF CHAR are illegal with a file variable of type TEXT:

- The SEEK procedure is illegal on a textfile.
- A textfile opened with RESET cannot be used for output, and a textfile opened with REWRITE cannot be used for input. Results are unspecified if either of these operations is attempted.

In place of these capabilities, textfiles provide the following:

- Automatic conversion of each input CR character into a space.
- The EOLN function to detect when the end of an input line has been reached.
- The READ procedure, which can read not only char values but also string values and numeric values (from textual representations).
- The WRITE procedure, which can write not only char values but also string values, numeric values, and boolean values (as textual representations).
- Line-oriented reading and writing via the READLN and WRITELN procedures.
- The PAGE procedure, which inserts a form-feed character into an output textfile.
- Automatic conversion of input DLE-codes to the sequences of spaces that they represent. Note that output sequences of spaces are not converted to DLE-codes.
- Automatic skipping of header blocks and null characters during input.
- Automatic generation of header blocks, and automatic padding of text "pages" with null characters on output.

The GET and PUT procedures can also be used with textfiles, although it is generally more convenient to use READ, READLN, WRITE, and WRITELN. GET and PUT with a textfile perform input and output of single characters, but they deal with the character sequence represented by the formatted textfile, not necessarily with the characters actually stored.

12.2 The READ Procedure

The syntax of the parameter list of READ is:

```
     READ-parameter-list = "("[file-variable ","]
variable-reference
                                {"," variable-reference}")"
     file-variable = variable-reference
```

If the file-variable is omitted, the procedure is applied to the standard file input.

In the following, f denotes a file of type TEXT and v1..vn denote variables of the types char (or a subrange of char), integer, subrange of integer, longint, string, or real:

a) READ(f,v1,...,vn) is equivalent to:

```
BEGIN READ(f,v1); ... ; READ(f,vn) END
```

b) If v is a variable of type char or subrange of char READ(f,v) is equivalent to:

```
BEGIN v := f^; get(f) END
```

c) If v is a variable of type integer, subrange of integer, or longint, READ(f,v) implies the reading from f of a sequence of characters that form a signed-integer according to the syntax of 4.5. The value read is assigned to v, if the value is assignment-compatible with the type of v. Preceding spaces and line-markers are skipped. Reading ceases as soon as the file buffer variable f^ contains a character that does not form part of a signed-integer. An error occurs if the sequence of characters does not form a signed-integer as specified in 4.5.

d) If v is a variable of type real, READ(f,v) implies the reading from f of a sequence of characters that form a signed-number according to the syntax of 4.5. The value of the number is assigned to the variable v. Preceding spaces and line-markers are skipped. Reading ceases as soon as the file buffer variable f^ contains a character that does not form part of a signed-number.

An error occurs if the sequence of characters does not form a signed-number as specified in 4.5.

e) If v is a variable of type string, READ(f,v) implies the reading from f of a sequence of characters up to but not including the line separator. The resulting character-string is assigned to v. An error occurs if the number of characters read exceeds the size attribute of v.

READ can also be used to read from a file f that is not a textfile. In this case READ(f,x) is equivalent to:

```
BEGIN x := f^; get(f) END
```

12.3 The READLN Procedure

The syntax of the parameter list of READLN is:

```
READLN-parameter-list = ["(" (file-variable | variable-reference)
                              {"," variable-reference} ")"]
file-variable = variable-reference
```

If the file-variable is omitted, the procedure is applied to the standard file input.

If the list of variables (other than the file-variable) is omitted, READLN causes a skip to the beginning of the next line (if there is one, else to the end-of-file). Thus READLN(f) is equivalent to:

```
BEGIN
  WHILE NOT eoln(f) DO get(f);
  IF NOT eof(f) THEN get(f)
END
```

READLN(f,v1,...,vn) is equivalent to:

```
BEGIN READ(f,v1,...,vn); READLN(f) END
```

12.4 The WRITE Procedure

The syntax of the parameter list of WRITE is:

```
     write-parameter-list = "("[file-variable ","]
write-parameter
                                {"," write-parameter}")"
     file-variable = variable-reference
     write-parameter = expression [":" expression
                          [":" expression ] ]
```

In the following intolerable discussion, which will someday be replaced by something a human being can comprehend, f denotes a textfile, p1,...,pn denote write-parameters, E denotes an expression, M and N denote expressions of type integer or longint:

a) WRITE(f,p1,...,pn) is equivalent to:

```
BEGIN WRITE(f,p1); ... ; WRITE(f,pn) END
```

b) The write-parameters p have the following forms:

```
E
E:M
E:M:N
```

where E is an expression whose value is to be written on the file f. E can be numeric (integer, longint, or real), char, Boolean, or a string. M and N are expressions whose integer values are the field-width parameters. Their values should be greater than zero; if they are not, the results are unspecified. Exactly M characters are written (with an appropriate number of spaces to the left of the representation of E), except when E is a numeric value that requires more than M characters for its representation; in such cases the number of characters written is as small as is consistent with the representation of E (see requirements d and e).

WRITE(f,E) is equivalent to WRITE(f,E:M), using a default value for M that depends on the type of E (see below).

WRITE(f,E:M:N) is applicable only if E is of type real (see requirement e).

c) If E is of type char, the default value for M is one.

d) If E is of type integer or longint, the decimal representation of the number E is written on the file f. The default value for M is 8. If pten is the positive integer defined by:

```
IF E = Ø
  THEN pten := 1
  ELSE determine pten such that 1Ø**(pten-1)
    <= abs(E) < 1Ø**pten
```

the representation consists of:

(1) if M >= pten + 1
(M-pten-1) spaces,
the sign character ("-" if E<Ø, otherwise a space),
pten digits.

(2) If M<pten+1, pten characters are written if E>=Ø, (pten+1) if E<Ø.

e) If E is of type real, a decimal representation of the number E, rounded to the specified number of significant figures or decimal places, is written on the file f. The default value for M is 12.

WRITE(f,E:M) causes a floating-point representation of E to be written. If edig is the number of digit characters written in the exponent, and the non-negative number er and the integer pten are defined by:

```
IF E = Ø.Ø
  THEN BEGIN er := Ø.Ø; pten := 1 END
  ELSE
  BEGIN
    er := abs(E);
    determine pten such that
      1Ø**(pten-1) <= er < 1Ø**pten;
    er := er + Ø.5 * (1Ø**(pten-M+edig+4));
    er is truncated to (M-edig-4) significant
      decimal figures
  END
```

the representation consists of:

(1) if M >= edig + 6:
the sign character
("-" if E<Ø and er<>Ø, otherwise a space),
the leading digit of er,
the character ".",
the next (M-edig-5) digits of er,
the character "E",
the sign of (pten - 1) ("+" or "-"),
edig digits for (pten - 1)
(with leading zeros if necessary).

(2) If M<edig+6, (edig+6) characters are written,
including one digit after the decimal point.

WRITE(f, E:M:N) causes a fixed-point representation of E to be written. If the non-negative number er and the positive integer pten are defined by:

```
IF E = Ø.Ø
  THEN er := Ø.Ø
  ELSE
  BEGIN
    er := abs(E);
    er := er + Ø.5 * 1Ø**(-N);
    er is truncated to N decimal places
  END;
IF trunc(er) = Ø
  THEN pten := 1
  ELSE determine pten such that 1Ø**(pten-1)
    <= trunc(er) < 1Ø**pten
```

the representation consists of:

(1) if M >= pten+N+2:
(M-pten-N-2) spaces,
the sign character
("-" if E<Ø and er<>Ø, otherwise a space),
the first pten digits of er,
the character ".",
the next N digits of er.

(2) If M<pten+N+2, (pten+N+2) characters are written.

f) If E is of type Boolean, a representation of the word true or the word false (as appropriate) is written on the file f. This is equivalent to:

WRITE(f,' TRUE':M) or WRITE(f,'FALSE':M)

as appropriate. The default value of M is 5 (see "g" below).

g) If E is of string type with length L, the default value for M is L. The value of E is written on the file f preceded by (M-L) spaces if M>=L. If M<L characters one through M of the string are written.

WRITE can also be used to write out a packed array[1..L] of char. The effect is the same as writing a string whose length is L.

WRITE can also be used to write onto a file which is not a textfile. In this case WRITE(f,x) is equivalent to:

BEGIN f^ := x; put(f) END

If the file-variable is omitted, the procedure is applied to the standard file output.

The value of the buffer variable f^ is unspecified immediately after WRITE(f).

12.5 The WRITELN Procedure

The syntax of the parameter list of WRITELN is:

```
writeln-parameter-list = ["(" (file-variable
                                | write-parameter)
                              {"," write-parameter}")"]
file-variable = variable-reference
write-parameter = expression [":" expression
                    [":" expression ] ]
```

WRITELN(f,p1,...,pn) is equivalent to:

```
BEGIN WRITE(f,p1,...,pn); WRITELN(f) END
```

WRITELN(f) appends a CR character to the file f.

If the file-variable or the parameter-list is omitted, the procedure is applied to the standard file output.

12.6 The EOLN Function

If a CR character is read from a textfile, the value of the buffer variable f^ becomes a space, and eoln(f) yields true. If a READ or READLN is executed while eoln(f) is true, the next character in the file is read, if there is a next character. If there are no more characters in the file, then eof(f) becomes true, eoln(f) remains true, and the value of f^ is undefined.

Also, eoln(f) will return true whenever eof(f) is true.

12.7 The PAGE Procedure

Page(f) inserts text into the textfile f that will cause skipping to the top of a new page when f is printed. The actual-parameter f cannot be omitted.

Section 13

PROGRAMS

A Pascal program has the form of a procedure declaration except for its heading and an optional uses-clause.

```
program = program-heading ";" [ uses-clause ";" ] block "."
program-heading = "PROGRAM" identifier
                    [ "(" program-parameters ")" ]
program-parameters = identifier-list
uses-clause = "USES" identifier-list
identifier-list = identifier {, identifier}
```

Currently, any program-parameters are purely decorative and are totally ignored by the compiler. The uses-clause, if present, identifies units that the program uses (see Section 14).

The code of a program's main body is always placed in a run-time segment whose name is a string of blanks (the "blank segment"). Any other block can be placed in a different segment by using the $S compiler option (see Appendix A). If no $S option is used in the program, all code is placed in the blank segment. Code from a program can be placed in the same segment with code from a regular-unit, but it cannot be mixed with code from an intrinsic-unit (see Section 14).

Section 14

UNITS

A unit is a separately compiled, non-executable object file which can be linked with other object files to produce complete programs. There are two kinds of units, called regular-units and intrinsic-units.

Each unit used by a program (or another unit) must be compiled, and its object file must be accessible to the compiler, before the program (or unit) can be compiled.

The syntax for a unit is:

```
unit = unit-heading ";"
          [intrinsic-clause ";"]
          interface-part
          implementation-part
          "END" "."
unit-heading = "UNIT" identifier
intrinsic-clause = "INTRINSIC" ["SHARED"]
interface-part = "INTERFACE"
                 [ uses-clause ]
                 [ constant-definition-part ]
                 [ type-definition-part ]
                 [ variable-declaration-part ]
                 [ procedure-and-function-declaration-part ]
implementation-part = "IMPLEMENTATION"
                 [ constant-definition-part ]
                 [ type-defintion-part ]
                 [ variable-declaration-part ]
                 [ procedure-and-function-declaration-part ]
```

If an intrinsic-clause appears after the unit-heading, the unit is an intrinsic-unit; if there is no intrinsic-clause, the unit is a regular-unit.

14.1 Regular-Units

When a program uses a regular-unit, the Linker inserts compiled code from the regular-unit into the host program's object file. Regular-units can be used as a means of modularizing large programs. A regular-unit can also be used as a means of making code available for incorporation in various programs, without

making the source available.

By default, the code of a regular-unit is placed in the "blank segment" (see Section 13). The code of the entire unit, or of blocks within the unit, can be placed in one or more different segments by using the $S compiler option (see Appendix A). Code from a regular-unit can be placed in the same segment with code from a program, but it cannot be mixed with code from an intrinsic-unit.

14.1.1 Writing Regular-Units

The INTERFACE section declares constants, types, variables, procedures, and functions that are public. The host program can access these entities just as if they had been declared in the host program. Procedures and functions declared in the INTERFACE section are abbreviated to nothing but the procedure or function name, parameter specifications, and function result-type.

Note that the INTERFACE section may contain a uses-clause. Thus a unit can use another unit (see 14.3).

The IMPLEMENTATION section, which follows the last declaration in the INTERFACE section, begins by declaring any local (private) constants, types, variables, procedures, or functions.

The public procedures and functions are declared again in the IMPLEMENTATION. The parameters and function result types are omitted from these definitions, since they were declared in the INTERFACE section; and the procedure and function blocks, omitted in the INTERFACE, are included in the IMPLEMENTATION.

NOTE: There is no INITIALIZATION section in Lisa Pascal units (unlike UCSD). Also note that global labels cannot be declared in a unit.

A short example of a unit is:

```
UNIT Simple;
  INTERFACE             { public objects declared }
    CONST FirstValue=1;
    PROCEDURE AddOne(VAR Incr:INTEGER);
    FUNCTION Add1(Incr:INTEGER):INTEGER;
  IMPLEMENTATION
    PROCEDURE AddOne;   { note lack of parameters... }
      BEGIN
        Incr:=Incr+1
      END;
    FUNCTION Add1;      { ...and lack of function result
                          type }
      BEGIN
        Add1:=Incr+1
      END;
END.
```

14.1.2 Using Regular-Units

The syntax for a uses-clause is given above in Section 13. Note that in a host program, the uses-clause (if any) must immediately follow the program-heading. In a host unit, the uses-clause (if any) immediately follows the symbol INTERFACE. Only one uses-clause may appear in any host program or unit; it declares all units used by the host program or unit.

See 14.3 for the case where a host uses a unit that uses another unit.

It is necessary to specify the file to be searched for regular units. The $U compiler option specifies this file. See Appendix A for more details.

Assume that the example unit Simple (see above) is compiled to an object file named APPL:SIMPLE.OBJ. The following is a short program that uses Simple. It also uses another unit named Other, which is in file APPL:OTHER.OBJ.

```
PROGRAM CallSimple;           { call the unit given above }
USES {$U APPL:SIMPLE.OBJ}     { file to search for units }
     Simple,                  { use unit Simple }
     {$U APPL:OTHER.OBJ}      { file to search for units }
     Other;                   { use unit Other }
VAR i:INTEGER;
BEGIN
  i:=FirstValue;              { FirstValue is from Simple }
  WRITE('i+1 is ',Add1(i));   { Add1 is defined in Simple }
  WRITE(xyz(i))               { xyz is defined in Other }
END.
```

14.2 Intrinsic-Units

Intrinsic-units provide a mechanism for Pascal programs to share common code, with only one copy of the code in memory. A single copy of the code is kept on disk, and when loaded into memory this code can be executed by any program that declares the intrinsic-unit (via a uses-clause, the same as for regular-units). In addition, a SHARED intrinsic-unit provides for the sharing of common data

The code of the entire unit, or of blocks within the unit, must be placed in one or more named segments by using the $S compiler option (see Appendix A). Code from a intrinsic-unit cannot be placed in the same segment with code from a program or a regular-unit.

14.2.1 Writing Intrinsic-Units

An intrinsic-unit has the same syntax as a regular-unit, except that it has an intrinsic-clause immediately after the heading:

(Note: for syntactic compatibility with UCSD Pascal, the keywords "CODE" and "DATA" may appear in the unit-heading of an intrinsic-unit, together with integer constants. These keywords and constants are accepted but are ignored.)

If the keyword SHARED appears in the intrinsic-clause, the system will contain only a single data area for the unit; the data is shared among all programs that use this unit. If SHARED does not appear in the intrinsic-clause, each program that uses the unit has its own data area for the unit.

If an intrinsic-unit contains a uses-clause, it can only use other intrinsic-units; an intrinsic-unit cannot use a regular-unit.

Each unit used by a program (or another unit) must be compiled, and its object file must be accessible to the compiler, before the program (or unit) can be compiled.

A single copy of the code of an intrinsic-unit is available to all programs in the system; therefore, intrinsic-units must be coordinated as part of system generation and system maintenance activities. Specifically, all intrinsic-units that have code in the same run-time code segment must be linked together into an intrinsic segment file, and the intrinsic segment file must be referenced in the system intrinsics library, INTRINSIC.LIB. For more information, see the Pascal Development System Manual.

14.2.2 Using Intrinsic-Units

The syntax for a uses-clause is given above in Section 13. Note that in a host program, the uses-clause (if any) must immediately follow the program-heading. In a host unit, the uses-clause (if any) immediately follows the symbol INTERFACE. Only one uses-clause may appear in any host program or unit; it declares all units used by the host program or unit.

See 14.3 for the case where a host uses a unit that uses another unit.

By default, the system looks up all units in the system intrinsics library file, *INTRINSIC.LIB. All intrinsic units are referenced in this library, so the $U compiler option is not needed with intrinsic-units.

14.3 Nested Units

When a host program (or host unit) uses a unit which uses another unit (which perhaps uses still another unit, etc.), the units are said to be nested. As explained above, the uses-clause in the host must name all units that are used by the host. Here "used" means that the host directly references something in the INTERFACE of the unit. Note that an intrinsic-unit cannot use a regular-unit.

In some cases, the uses-clause must also name a nested unit that is not actually used by the host. This is required when a unit (say UNITA) that is used by the host uses another unit (say UNITB), and the INTERFACE of UNITA contains a reference to something in the INTERFACE of UNITB. This occurs when some public entity in UNITA is defined in terms of a public entity in UNITB. The compiler must access both units; therefore, the uses-clause in the host must name both UNITB and UNITA -- in that order, the deepest nested unit first.

Section 15

THE LISA PASCAL COMPILER

15.1 Compiler Options

The following compiler options are available:

Control of Code Generation

$C+ or $C-	Turns code generation on (+) and off (-). This is done on a procedure by procedure basis. These options are intended to be written between procedures, not inside them. Default is $C+.
$D+ or $D-	Turn the generation of procedure names in object code (for debugging) on (+) and off (-). These options are intended to be written between procedures, not inside them. Default is $D+.
$R+ or $R-	Turn range checking on (+) and off (-). At present, range checking is done in assignment statements and array indexes and for string value parameters. Default is $R+.
$S segname	Start putting code modules into segment 'segname'. The default segment name is ' ', in which the main program and all built-in support code are always linked. All other code can be placed into any segment.
$X+ or $X-	Turn automatic stack expansion on (+) and off (-). Default is $X+.

Input File Control

$I filename	Include the file 'filename'. Filename cannot begin with a "+" or a "-".

$U filename	Search the file 'filename' for any subsequent units.

Conditional Compilation

$DECL list	(see 15.2 below).
$ELSEC	(see 15.2 below).
$ENDC	(see 15.2 below).
$IFC	(see 15.2 below).
$SETC	(see 15.2 below).

Listing Control

$E filename	Starts making a listing of compiler errors as they are encountered. Analogous to $L.
$L filename	Start making a listing of the compilation on file 'filename'. If a listing is being made already, that file is closed and saved prior to opening the new file.
$L+ or $L-	The first + or - following the "$L" turns the source listing on (+) or off (-) without changing the list file. You must specify the listing file before using $L+.
$L++ or $L+-	The second + or - following the "$L" turns the listing of object code offsets on (+) or off (-) without changing the list file. You must specify the listing file before using $L++ or $L+-.
$L+++ or $L++-	The third + or - following the "$L" turns the interlisting of machine code on (+) and off (-) without changing the list file. You must specify the listing file before using $L++ or $L+-.

Miscellaneous

$%+ or $%-	Allow the use of percent signs in identifiers. Don't use this in application programs. The default is $%-.

15.2 Conditional Compilation

Conditional compilation is controlled by the IFC, ELSEC, and ENDC options, which are used to bracket sections of source text. The DECL option is used to declare "compile-time variables," and the SETC option is is used to assign values to them.

15.2.1 Compile-Time Variables

IFC makes a decision based on the boolean value of a compile-time expression (see below). The expression can contain compile-time variables. These variables are completely independent of program variables; even if a compile-time variable and a program variable have the same identifier , they can never be confused by the compiler.

A compile-time variable is declared when it appears in the identifier list of a DECL option; for example, the option

```
{$DECL LIBVERSION, PROGVERSION}
```

declares LIBVERSION and PROGVERSION as compile-time variables. Notice that no types are specified; compile-time variables have no types, although their values do (see below).

The option

```
{$SETC LIBVERSION := 5}
```

assigns the value 5 to the compile-time variable LIBVERSION. Since 5 is an INTEGER value, LIBVERSION is now a variable of type INTEGER (the SETC option is explained in detail below). Now suppose that later in the compilation, the compiler finds

```
...
{$IFC PROGVERSION >= LIBVERSION}
K := KVAL1(DATA+INDAT);
{$ELSEC}
K := KVAL2(DATA+CPINDAT^);
{$ENDC}
WRITELN(K);
...
```

If the value of PROGVERSION is greater than or equal to 5 (the value of LIBVERSION), then the statement K := KVAL1(DATA+INDAT) is compiled, and the statement K := KVAL2(DATA+CPINDAT^) is skipped.

But if the value of PROGVERSION is not greater than the value of LIBVERSION, then the first statement is skipped, and the second statement is compiled.

In either case, the WRITELN(K) statement is compiled because the conditional construction ends with the {$ENDC} option.

Note the following points about compile-time variables:

- All compile-time variables must be declared before the end of the declarations section of the main program. In other words a DECL option that declares a new compile-time variable must precede the main program's procedure and function definitions (if any). The new compile-time variable is then known throughout the remainder of the compilation.

- At any point in the program, a compile-time variable can have a new value assigned to it by a SETC option.

- The type of a compile-time variable is that of the most recent value assigned to it in a SETC option. The only possible types are INTEGER, BOOLEAN, and CHAR.

15.2.2 Compile-Time Expressions

Compile-time expressions appear in the SETC option and in the IFC option. The only operands allowed in a compile-time expression are compile-time variables and constants of the types INTEGER, BOOLEAN, and CHAR.

All Pascal operators except IN and @ are allowed; the / operator is automatically replaced by DIV. A compile-time expression is evaluated by the compiler as soon as it is encountered in the text.

15.2.3 The SETC Option

The keyword SETC cannot be abbreviated. The SETC option has the form

```
{$SETC ID := EXPR}
```

where ID is the identifier of a compile-time variable and EXPR is a compile-time expression. EXPR is evaluated immediately. The value EXPR is assigned to ID.

15.2.4 The IFC, ELSEC, and ENDC Options

The keywords IFC, ELSEC, and ENDC cannot be abbreviated. The ELSEC and ENDC options take no arguments. The IFC option has the form

```
{$IFC EXPR}
```

where EXPR is a compile-time expression with a boolean value.

These three options form constructions similar to the Pascal IF statement, except that the ENDC option is always needed at the end of the IFC construction. ELSE is optional.

IFC constructions can be nested within each other to 10 levels. Every IFC must have a matching ENDC.

When the compiler is skipping, all options are ignored except the following:

```
ELSEC
ENDC
IFC (so that ENDC's can be matched properly)
INCLUDE (text is scanned even if it being skipped,
         in case it contains ELSEC, ENDC, or IFC
         options).
```

All program text is ignored during skipping. If a listing is produced, each source line that is skipped is marked with the letter S as its "lex level."

15.3 Optimization of IF Statement

When the compiler finds an IF statement controlled by a boolean constant, it may be unnecessary to compile the THEN part or the ELSE part. For example, given the declarations

```
CONST ALWAYS = TRUE;
      NEVER = FALSE;
```

then the statement

```
IF NEVER THEN <statement>
```

doesn't need to be compiled at all. In the statement

```
IF NEVER THEN <statement1> ELSE <statement2>
```

the THEN part is not compiled; only <statement2> is compiled. Similarly, in the statement

```
IF ALWAYS THEN <statement1> ELSE <statement2>
```

only <statement1> is compiled.

The interaction between this optimization and conditional compilation can be seen from the following program:

```
PROGRAM FOO;

{$SETC FLAG := FALSE}

CONST PI = 3.1415926;
      SIZE = 512;
      {$IFC FLAG}
      DEBUG = FALSE;
      {$ENDC}

VAR I,J,K,L,M,N: INTEGER;
    {$IFC NOT FLAG}
    DEBUG: BOOLEAN;
    {$ENDC}

...

{$IFC NOT FLAG}
PROCEDURE WHATMODE;
  BEGIN
    {interactive procedure to set
    global boolean variable DEBUG}
  END;
{$ENDC}

...

BEGIN {main}
  {$IFC NOT FLAG}
  WHATMODE;
  {$ENDC}
  ...

  IF DEBUG THEN BEGIN <statement1> END
           ELSE BEGIN <statement2> END

  ...
END.
```

The way this is compiled depends on the compile-time variable FLAG. If FLAG is FALSE, then DEBUG is a boolean variable and the WHATMODE procedure is compiled and called at the beginning of the main program. The IF DEBUG statement is controlled by a boolean variable and all of it is compiled, in the usual manner.

But if the value of FLAG is changed to TRUE, then DEBUG is a constant with the value FALSE, and WHATMODE is neither compiled nor called. The IF DEBUG statement is controlled by a constant, so only its ELSE part, <statement2>, is compiled.

15.4 Optimization of WHILE and REPEAT Statements

A WHILE or REPEAT statement controlled by a Boolean constant does not generate any conditional branches.

15.5 Using CASE Statements for Efficiency

A sparse or small CASE statement will generate better code than the corresponding sequence of IF/THEN/ELSE constructions.

Section 16

STANDARD PROCEDURES AND FUNCTIONS

16.1 General

Standard procedures and functions are pre-declared. Since all pre-declared entities are declared in a range surrounding the program, no conflict arises from a declaration redefining the same identifier within the program block.

Standard procedures and functions cannot be used as actual procedural and functional parameters.

16.2 Basic I/O

This section covers the I/O procedures for all file types except the type TEXT and "untyped" files. Files of type TEXT are covered in Section 12.

"Untyped" files are variables of type FILE, used in conjunction with the BLOCKREAD and BLOCKWRITE functions.

16.2.1 RESET(f,title)

The parameter f is a file variable, and the parameter title is a string containing the filename.

Reset opens an existing external file and performs an implicit get(f). Note that both parameters are required (unlike UCSD Pascal).

Eof(f) becomes false if f is not empty; otherwise the value of f^ is unspecified, and eof(f) is true.

If the file is an interactive device, eof(f) is always false and the implicit get(f) operates as described above.

If there is no existing external file with the specified filename, an error occurs.

NOTE: This is a necessary initializing operation prior to accessing an existing file with the specified filename.

16.2.2 REWRITE(f,title)

The parameter f is a file variable, and the parameter title is a string containing the filename.

Rewrite creates and opens a new external file. Eof(f) becomes true and the value of f^ becomes unspecified. Title is a string containing the filename. If this is the filename of an existing external file, the existing file is discarded when the new file is closed with the LOCK option (see 16.2.3).

NOTE: This is a necessary initializing operation prior to creating a new external file with the specified filename.

Unspecified effects are caused if the current file position of a file f is altered while the buffer variable f^ is an actual variable parameter, or an element of the record-variable-list of a with-statement, or both.

NOTE: The standard procedures read, write, readln, writeln and page are described in Section 12.

16.2.3 CLOSE(f,[option])

This procedure closes a file f which was previously opened with RESET or REWRITE. The option identifier, if used, may be any one of the following predefined identifiers. If no option identifier is used, the effect is the same as using the NORMAL option.

> NORMAL -- A normal close is done. If the file was opened using REWRITE and the external file is a disk file, it is deleted from the directory.
>
> LOCK -- If the external file is a disk file and was opened with REWRITE, it is made permanent in the directory; otherwise a NORMAL close is done. If the file was opened with a REWRITE and the pathname matches an existing disk file, the old file is deleted.

PURGE -- If the external file is a disk file, it is deleted from the directory (unless write-protected). In the special case of a disk file that already exists and is opened with REWRITE, the original file remains in the directory, unchanged. If the external file is not a disk file, the associated unit will go off-line.

CRUNCH -- This is like LOCK except that it locks the end-of-file to the point of last access; i.e., everything after the last record accessed is thrown away.

All CLOSEs regardless of the option will mark the file closed and will make the file buffer variable undefined. CLOSE on a closed file causes no action.

If a program terminates with a file open (i.e., if CLOSE is omitted), the system automatically closes the file with the NORMAL option.

NOTE: If you open an existing file with RESET and modify the file with any write operation, the contents are immediately changed no matter what CLOSE option you specify.

16.2.4 EOF(f)

The parameter f is a file variable.

This indicates whether the associated buffer variable f^ is positioned at the end of the file f. If the actual parameter is omitted, the function is applied to the standard file input.

16.2.5 EOLN(f)

The parameter f is a file variable.

This indicates whether the associated buffer variable f^ is positioned at the end of a line in the textfile f (see 12.1). If the actual parameter is omitted, the function is applied to the standard file input.

16.2.6 SEEK(f,n)

The SEEK procedure allows the program to access any specified record in a file. The parameter f is the identifier of a file and n is an

expression with an integer value that specifies a record number in the file. Note that records in files are numbered from Ø.

SEEK affects the action of the next GET or PUT from/to the file, forcing it to access the specified file record instead of the "next" record. SEEK does not affect the file's buffer variable.

The file should be a file on a disk or other block-structured device. It should not be a character device, nor should it be declared as a file of type TEXT.

A GET or PUT must be executed between SEEK calls since two SEEKs in a row may cause unpredictable results. Immediately after a SEEK, EOF will return false; a following GET or PUT will cause EOF to return the appropriate value.

NOTE: The record number specified in a SEEK call is not checked for validity. If the number is not the number of a record in the file and the program tries to GET the specified record, the value of the buffer variable becomes undefined and EOF becomes true.

NOTE: If the file is a character device or is of the type TEXT, SEEK does nothing.

16.2.7 PUT(f)

The parameter f is a file variable.

If eof(f) is false, the value of the buffer variable f^ is written to the file f at the current file position and then the file position is advanced to the next component. If the resulting file position is beyond the end of the file, eof(f) becomes true, and the value of f^ becomes unspecified.

If eof(f) is true, the value of the buffer variable f^ is appended to the end of the file f and eof(f) remains true.

If the file is an interactive device, eof(f) is always false, there is no "current file position," and the value of f^ is sent to the device.

16.2.8 GET(f)

The parameter f is a file variable.

If eof(f) is false, the current file position is advanced to the next component, and the value of this component is assigned to the buffer variable f^. If no next component exists, then eof(f) becomes true, and the value of f^ becomes unspecified.

If eof(f) is true when get(f) is called, then eof(f) remains true, and the value of f^ becomes unspecified.

If the file is an interactive device, eof(f) is always false, there is no "current file position," and there may or may not be a value ready for input. In this case, a "lazy evaluation" method is used: get(f) merely sets an internal flag to indicate that "a get is pending." A subsequent reference to f^ causes a value to be input from the device and assigned to f^ before the program retrieves it from f^; if no value is ready for input, the system waits for a value.

16.2.9 Control Characters With GET and PUT

Files of type TEXT are usually accessed with the text I/O procedures described in Section 12. When GET and PUT are used with files of type TEXT or with character devices, there are some special considerations for three special control characters, namely

CR, ASCII 13
DLE, ASCII 16
ETX, ASCII 3

The CR character is used in files of characters to mark the end of a line. The special handling for CR is as follows:

- GET converts CR to a space (ASCII 32).
- PUT to a disk file: no special handling.
- PUT to a character device: an LF (ASCII 1Ø) is written immediately after the CR.

The DLE character may occur in textfiles as the first character of a two-character code that represents a sequence of spaces anywhere on a line. The special handling for DLE is as follows:

- GET from any file except the console: the DLE and the following character are converted to a sequence of spaces (the spaces are returned to the program by successive GETs).

- GET from the console: no special handling.
- PUT to a disk file: no special handling.
- PUT to a character device: the DLE and the following character are converted into a sequence of spaces.

The ETX character is used with character devices as an "end-of-text" indicator. The special handling is as follows:

- GET from a disk file: no special handling.
- GET from a character device: ETX is converted to a space. EOF and EOLN will return TRUE.
- PUT: no special action.

16.2.10 IORESULT

This built-in function takes no parameters and returns an integer value which reflects the status of the last completed I/O operation. The codes are given in the Lisa Pascal Development System Manual.

16.2.11 GOTOXY(x,y)

This procedure sends the screen cursor to the specified coordinates.

16.3 Untyped File I/O

Untyped file I/O treats a file variable as a sequence of 512-byte "blocks"; the bytes are not type-checked but considered as raw data. This can be useful for applications where the data need not be interpreted at all during I/O operations.

To use untyped file I/O, the file is declared with type FILE and no component-type, and the BLOCKREAD and BLOCKWRITE functions are used for input and output.

An untyped file has no buffer variable, and it cannot be used with GET, PUT, or any of the text I/O procedures. It can only be used with RESET, REWRITE, CLOSE, EOF, and the BLOCKREAD and BLOCKWRITE functions described below.

16.3.1 BLOCKREAD(fname,destvar,count[,startblock])

This function is used to transfer one or more 512-byte blocks of data from an untyped file fname to a program variable destvar. The count parameter specifies the number of blocks to be transferred, the startblock parameter specifies the starting block number in the file, and the integer value returned by blockread is the number of blocks actually transferred. The parameters are as follows:

- fname must be the identifier of an open untyped file.

- destvar must be a variable-reference. It refers to the variable into which the blocks of data will be read. The size and type of this variable are not checked; if it is not large enough to hold the data, other program data may be overwritten and the results are unpredictable.

- count must be an integer value. It specifies the number of blocks of data to be transferred. BLOCKREAD will read as many blocks as it can, up to this limit; if the end of the file is reached before the specified number of blocks are read, then EOF will be true and the value returned by BLOCKREAD indicates how many blocks were actually read.

- startblock must be an integer value. It specifies the starting block number in the file (see below).

The blocks in a file are considered to be numbered sequentially starting with Ø. The system keeps track of the "current" block number in each open untyped file; this is block Ø immediately after the file is opened. Each time a block is read, the current block number is incremented. If the blocknumber expression is omitted in a call to BLOCKREAD, the transfer begins with the current block. Thus the transfers are sequential if the blocknumber expression is never used; if a blocknumber expression is used, it provides random access to blocks.

After BLOCKREAD, EOF is true if the last block in the file was read.

16.3.2 BLOCKWRITE(fname,destvar,count[,startblock])

This function is used to transfer one or more 512-byte blocks of data from a program variable destvar to an untyped file fname. The count parameter specifies the number of blocks to be transferred, the startblock parameter specifies the starting block number in the file, and the integer value returned by blockread is the number of blocks actually transferred. The parameters are as follows:

- fname must be the identifier of an open untyped file.

- destvar must be a variable-reference. It refers to the variable from which the blocks of data will be read. The size and type of this variable are not checked.

- count must be an integer value. It specifies the number of blocks of data to be transferred. BLOCKWRITE will write as many blocks as it can, up to this limit; if disk space runs out before the specified number of blocks are read, then EOF will be true and the value returned by BLOCKWRITE indicates how many blocks were actually written.

- startblock must be an integer value. It specifies the starting block number in the file (see explanation under BLOCKREAD above).

16.4 Device I/O

The system identifies each peripheral device by a unit number and a unit name. The procedures described below can be used for direct communication with any peripheral device.

Note that UNITREAD and UNITWRITE are not controlled in any way by the format in which files are stored on a disk; they treat an entire disk as one sequence of blocks.

16.4.1 UNITREAD(unitnum,destvar,count[,[startblock],[mode]])

The unitread procedure transfers one or more bytes from a specified device to a program variable. The parameters are as follows:

unitnum must be an integer an integer value that is the unit number of an I/O device.

destvar must be a variable-reference. It refers to the variable into which the bytes of data will be read. The size and type of this variable are not checked; if it is not large enough to hold the data, other program data may be overwritten and the results are unpredictable.

count must be an integer value. It specifies the number of bytes of data to be transferred.

startblock must be an integer value. It is meaningful only when using a disk drive and is the absolute block number at which the transfer will start. If startblock is omitted and the unit is a disk drive, the transfer will start at block Ø.

mode must be an integer value. If it is omitted, the default is Ø. It controls options which are described below in 16.4.3.

16.4.2 UNITWRITE(unitnum,destvar,count[,[startblock],[mode]])

The unitwrite procedure transfers one or more bytes from a program variable to a specified device. The parameters are as follows:

unitnum must be an integer an integer value that is the unit number of an I/O device.

destvar must be a variable-reference. It refers to the variable from which the bytes of data will be read. The size and type of this variable are not checked.

count must be an integer value. It specifies the number of bytes of data to be transferred.

startblock must be an integer value. It is meaningful on when using a disk drive and is the absolute block number which the transfer will start. If startblock is omitted the unit is a disk drive, the transfer will start at block Ø.

mode must be an integer value. If it is omitted, the default is Ø. It controls options which are described be in 16.4.3.

16.4.3 Device I/O Modes

The device I/O options are enabled by default. They are designed to handle the special coding found in files of type text, and are convenient when text is read from such a file with UNITREAD and then output via UNITWRITE to a character device (see 16.2.9 for more explanation).

For UNITWRITE, the options apply only to character devices, not to disks or other block-structured devices.

- Conversion of DLE-blank codes to output sequences of spaces is disabled by a "mode" value that has a one in Bit 2 (see below).

- Automatic linefeeds after every CR are disabled by a "mode" value that has a one in Bit 3 (see below).

For UNITREAD, the only option is ETX (CTRL-C) recognition (enabled by default). When the ETX character (ASCII 3) is recognized by UNITREAD, the effect is to terminate the input. Any bytes in the destination variable that remain unused at this point are filled with Ø's.

Only Bit 2 and Bit 3 of the mode value have any current significance. The other bits are reserved for future use.

The following values can be used to control the options:

- Mode = Ø (the default value) enables all options.

- Mode = 4 disables DLE conversion and ETX recognition, and enables automatic linefeeds.

- Mode = 8 disables automatic linefeeds and enables DLE conversion and ETX recognition.

- Mode = 12 disables all options.

16.4.4 UNITCLEAR(unitnum)

This procedure cancels all I/O operations to the specified unit and resets the hardware to its power-up state. The unitnum parameter must be an integer value that is the unit number of an I/O device.

IORESULT is set to a non-zero value if the specified unit is not present (you can use this to test whether or not a given unit is present in the system). Note that UNITCLEAR (1) flushes the type-ahead buffer for the console and resets certain keyboard and screen parameters.

16.4.5 UNITBUSY(unitnum)

This function returns true if the specified device is busy. The unitnum parameter must be an integer value that is the unit number of an I/O device.

16.5 EXIT and HALT Procedures

16.5.1 EXIT(identifier)

The EXIT procedure exits from a specified procedure or function, or from the main program. Currently, the parameter must be the identifier of a procedure or function (possibly the identifier of the main program). If the parameter is an identifier defined in the program, it must be in the scope of the EXIT call. Note that this is more restricted than UCSD Pascal.

Eventually, the language will allow the parameter to be the predefined identifier PROGRAM. This will cause an exit from the main program, without the need to know the main program's identifier. This will make it possible to exit the main program from within a unit. For now, use HALT to exit the main program from a unit.

EXIT essentially causes a jump to the end of the named procedure.

16.5.2 HALT

The halt procedure takes no parameters. It causes immediate exit from the main program, with an error message.

16.6 Dynamic Allocation Procedures

These procedures are used to manage the heap, a memory area that is unallocated when the program starts running. As areas within the heap area are allocated via the procedure new (see below), a free-list is maintained which identifies unallocated areas within the heap.

One mechanism, the procedure new, is used for all allocation of heap space by the program. Two distinct mechanisms are used to deallocate heap space: one is the dispose procedure, which puts a particular area on the free-list, and the other is the mark and release procedures used together. Do not mix the two methods unless you are sure of what you are doing.

16.6.1 NEW(p)

This allocates a new variable v of the type that the pointer parameter p is bound to, and assigns its pointer to p. If the type of v is a record type with variants, the form

```
new(p,t1,...tn)
```

allocates a variable with space for the variants specified by the tag values t1,...,tn (instead of enough space for the largest variants). The tag values must be constants. They must be listed contiguously and in the order of their declaration. Trailing tag values can be omitted. The tag values are not assigned to the tag-fields by this procedure.

WARNING: When a record variable is dynamically allocated with explicit tag values as shown above, you should not make assignments to any fields that are not selected by the tag values. Also, you should not assign an entire record to this record. If you do either of these things, other data can be overwritten without any error being detected at compile time.

If there is no marked position in the heap (see 16.6.3 below), then new allocates space by searching the free-list until it finds a large enough area; space allocated by successive calls to new is not necessarily contiguous in this case.

If there is a marked position in the heap, then successive calls to new allocate contiguous areas beginning at the marked position.

16.6.2 DISPOSE(p)

The parameter p is of type pointer.

This indicates that the heap area occupied by the variable p^ is no longer needed; it is placed on the free-list. The value of p is set to NIL, and if another pointer pp was previously set equal to p, then the value of pp^ becomes unspecified. If the second form of new was used to allocate the variable, the following form of dispose must be used with identical tag value constants:

```
dispose(p,t1,...,tn)
```

An error is caused if the value of p is NIL when dispose is called.

Unspecified effects are caused if a variable that is currently either an actual variable parameter, or an element of the record-variable-list of a with-statement, or both, is referred to by the pointer parameter of dispose.

16.6.3 MARK(p)

The pointer parameter p is a variable parameter. The mark procedure causes the pointer p (which may be of any pointer-type) to point to the highest free area in the heap. The pointer p is also placed on a stack-like list for subsequent use with the release procedure (see below). If there is at least one pointer on this list, the new procedure always allocates areas contiguously, starting at the location pointed to by the most recent pointer on the list.

16.6.4 RELEASE(p)

The pointer p must be on the list created by the mark procedure; otherwise an error occurs. The effect of release is to remove pointers from the list, back to and including the pointer p. The

corresponding heap areas are deallocated and placed on the free-list.

16.6.5 MEMAVAIL

The MEMAVAIL function takes no parameters. It returns the number of words (not bytes) of memory available (heap and stack).

16.7 Transfer Procedures and Functions

The procedures pack and unpack, described by Jensen and Wirth, are not supported.

16.7.1 TRUNC(x)

From the real parameter x, this function returns an integer or longint result that is the integral part of x (rounding toward Ø). For example:

```
trunc(3.7) yields 3
trunc(-3.7) yields -3
```

16.7.2 ROUND(x)

From the real parameter x, this function returns an integer or longint result that is the value of x rounded to the nearest whole number. If x is positive or zero, round(x) is equivalent to trunc(x+Ø.5). For example:

```
round(3.5) yields 4
round(2.5) yields 3
round(3.7) yields 4 If x is negative, round(x) is equivalent to
```
trunc(x-Ø.5). For example:

```
round(-3.5) yields -4
round(-2.5) yields -3
round(-3.7) yields -4
```

16.7.3 ORD4(x)

The parameter x must be an expression of ordinal-type or pointer-type. The result is always of type longint. If the parameter is of type

longint, the result is the same as the parameter.

If the parameter is of pointer-type, the result is the corresponding physical address, of type longint.

If the parameter is of type integer, the result is the same numerical value represented by the parameter, but of type longint. This is useful in arithmetic expressions; for example consider the expression

```
abc*xyz
```

where both abc and xyz are of type integer. By the rules given in 7.2.2.2, the result of this multiplication is of type integer (16 bits). If the mathematical product of abc and xyz cannot be represented in 16 bits, the result is truncated. To avoid this, the expression can be written as

```
ord4(abc)*xyz
```

If the parameter is of an ordinal-type other than integer or longint, the numerical value of the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative integers starting at zero.

16.7.4 POINTER(x)

From the integer or longint parameter x, this function returns a pointer value that points to the physical address x. This pointer is of the same type as NIL and is assignment-compatible with any pointer-type.

16.8 Arithmetic Functions

For all of the arithmetic functions (except odd), the type of the actual-parameter x may be real, integer, or longint. For the abs function the type of the result is the same as the type of the parameter. For the sqr function, the result is real for a real parameter, longint for a longint parameter, and either integer or longint for an integer parameter. For the remaining arithmetic functions, the type of the result is always real:

16.8.1 ODD(x)

This yields true if the integer or longint expression x is odd; otherwise it yields false.

16.8.2 ABS(x)

This computes the absolute value of x.

16.8.3 SQR(x)

This computes the square of x.

16.8.4 SIN(x)

This computes the sine of x, where x is in radians.

16.8.5 COS(x)

This computes the cosine of x, where x is in radians.

16.8.6 EXP(x)

This computes the value of the base of natural logarithms raised to the power x.

16.8.7 LN(x)

This computes the natural logarithm of x, if x is greater than zero. If x is not greater than zero an error occurs.

16.8.8 SQRT(x)

This computes the positive square root of x, if x is not negative. If x is negative an error occurs.

16.8.9 ARCTAN(x)

This computes the principal value, in radians, of the arctangent of x.

16.9 Ordinal Functions

16.9.1 ORD(x)

The parameter x must be an expression of ordinal-type or pointer-type. The result is of type integer or longint. If the parameter is of type integer or longint, the result is the value of the parameter (with the same type).

If the parameter is of pointer-type, the result is the corresponding physical address, of type longint.

If the parameter is of another ordinal-type, the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative whole numbers starting at zero.

Note that for a parameter of type char, the result is the corresponding ASCII code. For booleans,

```
ord(false) yields 0
ord(true) yields 1
```

16.9.2 CHR(x)

This yields the character value whose ordinal number (i.e., its ASCII code) is equal to the value of the integer or longint expression x, if such a character value exists. If x is not in the range 0..255, the value returned is not within the range of the type char, and any attempt to assign it to a variable of type char will cause an error.

NOTE: For any character value, ch, the following is true:

```
chr(ord(ch)) = ch
```

16.9.3 SUCC(x)

The parameter x must be an expression of ordinal-type. The result will be of a type identical to that of the expression (see 10.1), unless the value of x has the highest ordinality in the type of x. The function yields a value whose ordinality is one greater than that of x, if such a value exists. If such a value does not exist, the value returned is not within the range of the type of x, and any attempt to assign it to a variable of this type will cause an error.

16.9.4 PRED(x)

The parameter x must be an expression of ordinal-type. The result will be of a type identical to that of the expression (see 10.1), unless the value of x has the lowest ordinality in the type of x. The function yields a value whose ordinality is one less than that of x, if such a value exists. If such a value does not exist, the value returned is not within the range of the type of x, and any attempt to assign it to a variable of this type will cause an error.

16.10 String Procedures and Functions

In the following descriptions, a "string value" means a string variable, a string constant, or any function or expression whose value is a string. Unless otherwise stated all parameters are value parameters.

NOTE: the string procedures and functions do not accept packed array of char parameters, and they do not accept indexed string parameters.

16.10.1 LENGTH(str)

The LENGTH function returns the (dynamic) length of its parameter, which must be a string value.

16.10.2 POS(substr,str)

The POS function searches for a specified substring within a specified string, and returns an integer value which is the index of the first character of the substring within the string. Both parameters must be

string values.

If the substring is not found, POS returns zero.

16.1Ø.3 CONCAT(list)

The CONCAT function concatenates strings and returns a string value. CONCAT can take any practical number of actual parameters each of which is a string value; the parameters are separated by commas.

CONCAT returns a string which is the concatenation of all the strings passed to it, in the order in which they appear in the actual-parameter-list.

16.1Ø.4 COPY(source,index,count)

The COPY function returns a string value. The source parameter must be a string value, and the index and count parameters must be integers.

COPY returns a string containing count characters copied from source starting at the index-th position in STRG.

16.1Ø.5 DELETE(dest,index,count)

The DELETE procedure modifies the value of a string variable. The dest parameter is a variable parameter of a string-type, and the index and count parameters are integers. DELETE removes count characters from dest starting at the index specified.

16.1Ø.6 INSERT(source,dest,index)

The INSERT procedure modifies the value of a string variable, inserting a substring into it. The source parameter must be a string value, the dest parameter must be a variable parameter of string-type, and the index parameter must be an integer.

INSERT inserts source into dest at the index-th position in dest.

16.11 Byte-Oriented Procedures and Functions

These features allow a program to treat a program variable as a sequence of bytes, without regard to data types. The SIZEOF function can be used to determine the number of bytes in a variable.

These procedures do no type checking on their "source" or "dest" actual-parameters. However, since these are variable parameters they cannot be subscripted if they are packed or if they are of a string-type. If an unpacked "byte array" is desired, then the type

```
array [lo..hi] of -128..127
```

should be used for "source" or "dest". The elements in an array of this type are stored in contiguous bytes, and since it is unpacked, an array of this type can be used with an index as an actual-parameter for these routines.

Note that currently an array with elements of the type Ø..255 or the type char has its elements stored in words, not bytes.

16.11.1 MOVELEFT(source,dest,length)

This procedure does a mass move of a specified number of bytes. The parameters are as follows:

- source must be a variable reference. It may refer to a variable of any type except a file-type, or a structured-type that contains a file-type. The first byte of this variable (lowest address) is the beginning of the range of bytes whose values are copied.

- dest must be a variable reference. It may refer to a variable of any type except a file type, or a structured-type that contains a file-type. The first byte of this variable (lowest address) is the beginning of the range of bytes that the values are copied into.

- length must be an integer. It specifies the number of bytes to be moved.

APPENDIX A

LISA PASCAL AND UCSD PASCAL

Introduction

This appendix contains a list of the major differences between the Lisa Pascal language and the UCSD Pascal described in the Apple Pascal Language Reference Manual and the Apple III Pascal Programmer's Manual.

Extensions

The following features have been added to UCSD Pascal:

- @ Operator -- returns the pointer to its operand (see 1Ø.2.6)
- Hexadecimal constants (see 4.5)
- Case OTHERWISE clause (same as Apple III Pascal; see 11.3.3.2)
- DISPOSE procedure (see 16.6.2)
- Global GOTOs(see 11.2.4)
- A "FILE OF CHAR" type that is distinct from the "TEXT" type (see 7.3.5 and 12.1).
- Numerous compiler options (see 15.1)
- Procedural and functional parameters (see 9.3.1, 9.3.4, and 9.3.5)
- Stronger type-checking (see 7.5 and 9.3.6)
- POINTER and ORD4 functions (see 16.7.3 and 16.7.4)
- SHARED keyword in Intrinsic units, to allow shared data segments (see 14.2)

Deletions

The following features of UCSD Pascal have not been included:

- Initialization block in a unit
- Declaration of CODE and DATA segment numbers in an Intrinsic unit. The syntax allows these declarations but the compiler ignores them. Segments are assigned automatically at run time.
- INTERACTIVE file type ("lazy evaluation" is used instead when input is from an interactive device)
- RESET procedure -- the syntax "RESET(f)", where f is a file variable identifier, is not allowed. The required syntax is "RESET(f, title)", where title is a string containing the external filename.
- KEYBOARD file -- use UNITREAD of unit 2 to get keyboard input without echoing
- PWROFTEN, TREESEARCH, BYTESTREAM, WORDSTREAM, UNITSTATUS
- EXIT(PROGRAM) -- The EXIT(identifier) form works, and the identifier can be the program-identifier. EXIT(PROGRAM) will be allowed eventually.
- Extended comparisons (including comparison of a STRING with a PACKED ARRAY OF CHAR)
- Bit-wise boolean operations
- SEGMENT keyword for procedures and functions (use the $S option)
- Compiler options:
    - $I (No automatic I/O checking; program must use IORESULT function)
    - $G ($G+ is the assumption of LISA Pascal)

- $N and $R (for resident code segments)
- $P
- $Q
- $S+ and $S++ for swapping
- $U+ and $U- (for User Program)
- $V

Replacements

The following UCSD features have been replaced with similar features:

- Long Integers -- LONGINT type (see 7.2.2.2)
- SCAN function -- SCANEQ and SCANNE (see 16.12.1 and 16.12.2)
- TURTLEGRAPHICS and APPLESTUFF -- LISAGRAF
- KEYPRESS -- use NOT UNITBUSY(2)

Other Differences

The following aspects of Lisa Pascal are not those of UCSD Pascal:

- STRING length must be explicitly declared (see 7.2.2.6).
- In UCSD Pascal, MOD and DIV are completely inconsistent. Lisa Pascal truncates toward Ø (see 1Ø.2.2).
- UCSD Pascal ignores underscores; Lisa Pascal does not (they are legal characters in identifiers; see 4.3).
- A GOTO statement cannot refer to a CASE constant in Lisa Pascal (see 11.2.4).
- A program must begin with the word PROGRAM in Lisa Pascal (see Section 13).
- TRUNC does not convert a long integer to integer type (this is not necessary in Lisa Pascal; long integers are allowed

in most places where integers are allowed).

- NEW, MARK, and RELEASE are subtly different (see 16.6).

- WRITE(b) where b is a boolean will write either ' TRUE' or 'FALSE' in Lisa Pascal (see 12.4).

- Whether a file is a textfile does not depend on whether its name ends with ".TEXT" when it is created. Instead, any file opened with a file variable of type TEXT is treated as a textfile, while a file opened with a file variable of type FILE OF CHAR is not; it is treated as a "datafile" or "asciifile", i.e. a straight file of records which are of type char (see 7.3.5 and 12.1).

- Lisa Pascal does not let you pass an element of a packed variable as a VAR parameter(see 9.3.3, 16.11, and 16.12).

- Limits on sets are different (see 7.3.4)

- The control variable of a FOR statement must be a local variable (see 11.3.4.3)

- In a WRITE or WRITELN call, the default field lengths for integers and reals are 8 and 12 respectively (see 12.4)

- HALT causes an orderly exit from the main program (currently via "Fatal Error #2"). It can be used to exit the main program from within a unit.

- EXIT is more restricted than in UCSD Pascal (see 16.5.1)

APPENDIX B

KNOWN ANOMALIES IN LISA PASCAL

Introduction

This appendix describes the known anomalies in the current implementation of the Lisa Pascal language.

Scope of Declared Constants

Consider the following program:

```
program cscope1;

const ten=1Ø;

procedure p;
  const ten=ten;  {THIS SHOULD BE AN ERROR}
  begin
    writeln(ten)
  end;

begin
  p
end.
```

The constant declaration in procedure p should cause a compiler error, because it is illegal to use an identifier within its own definition (except for pointer identifiers). However, the error is not detected by the compiler. The effect is that the value of the global constant ten is used in defining the local constant ten, and the writeln statement writes "1Ø".

A more serious anomaly of the same kind is illustrated by the following program:

```
program cscope2;

const red=1;
      violet=2;

procedure p;
  type a=array[red..violet] of integer;
       color=(violet,blue,green,yellow,orange,red);
  var v:a;
      c:color;
  begin
    v[1]:=1;
    c:=red;
    writeln(ord(c))
  end;

begin
  p
end.
```

Within the procedure p, the global constants red and violet are used to define an array index type; the effect of "array[red..violet]" is equivalent to "array[1..2]". In the definition of the type color, the constants red and violet are locally redefined; they are no longer equal to 1 and 2 respectively -- instead they are constants of type color with ordinalities 5 and Ø respectively. The writeln statement writes "5".

The use of "red" in the definition of the type color should cause a compiler error but does not.

Note the statement

```
v[1]:=1;
```

If this statement is replaced by

```
v[red]:=1;
```

a compiler error will result, as "red" is now an illegal index value for v -- even though v is of type a and a is defined by "array[red..violet]".

To avoid this kind of situation, avoid redefinition of constant-identifiers in enumerated scalar types.

Scope of Base-Types for Pointers

Consider the following program:

```
program pscope1;

type s=Ø..7;

procedure makecurrent;
  type sptr=^s;
       s=record
           ch:char;
           bool:boolean
         end;
  var current:s;
      ptrs:sptr;
  begin
    new(ptrs);
    ptrs^:=current
  end;

begin
  makecurrent
end.
```

Here we have a global type s which is a subrange of integer, and a local type s which is a record type. Within the procedure makecurrent, the type sptr is defined as a pointer to a variable of type s. The intention is that this should refer to the local type s, defined on the next line of the program; unfortunately, however, the compiler does not yet know about the local type s and uses the global type s. Thus ptrs becomes a pointer to a variable of type Ø..7 instead of a pointer to a record. Consequently the statement

```
ptrs^:=current
```

causes a compiler error since ptrs^ and current are of incompatible types.

To avoid this kind of situation, avoid redefinition of identifiers that are used as base-types for pointer-types.

APPENDIX C

SYNTAX OF THE LANGUAGE

Introduction

This appendix collects the BNF syntax specifications found in the main sections of this manual. See Section 3 for a description of the metalanguage.

Lexical Tokens (see Section 4)

```
letter = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"
       | "L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"
       | "W"|"X"|"Y"|"Z"
       | "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"
       | "l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"
       | "w"|"x"|"y"|"z"

digit = "Ø"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

special-symbol = "+"|"-"|"*"|"/"|"="
               | "<"|">"|"["|"]"|"."|"("|")"
               | ","|":"|";"|"^"|"@"|"$"
               | "<>"|"<="|">="|":="|".."| word-symbol

word-symbol = "AND"|"ARRAY"|"BEGIN"|"CASE"|"CONST"|"DIV"
            | "DOWNTO"|"DO"|"ELSE"|"END"|"FILE"|"FOR"
            | "FUNCTION"|"GOTO"|"IF"|"IMPLEMENTATION"
            | "IN"|"INTERFACE"|"INTRINSIC"|"LABEL"|"MOD"
            | "NIL"|"NOT"|"OF"|"OR"|"OTHERWISE"|"PACKED"
            | "PROCEDURE"|"PROGRAM"|"RECORD"|"REPEAT"
            | "SET"|"STRING"|"THEN"|"TO"|"TYPE"|"UNIT"
            | "UNTIL"|"USES"|"VAR"|"WHILE"|"WITH"

directive = "FORWARD" | "EXTERNAL"
```

```
digit-sequence = digit {digit}
hex-digit-sequence = hex-digit {hex-digit}
hex-digit = digit|"A"|"B"|"C"|"D"|"E"|"F"
unsigned-integer = digit-sequence
hex-integer = "$" hex-digit-sequence
unsigned-real = digit-sequence "E" scale-factor
              | digit-sequence "." digit-sequence
                  ["E" scale-factor]
unsigned-number = unsigned-integer
                | hex-integer
                | unsigned-real
scale-factor = signed-integer
sign = "+" | "-"
signed-integer = [sign] unsigned-integer
signed-number = [sign] unsigned-number

label = unsigned-integer

quoted-string-constant = "'" {string-character} "'"
string-character = any ascii char except CR or "'"
                 | "'" "'"
```

Blocks (see Section 5)

```
block = [ label-declaration-part ]
        [ constant-definition-part ]
          [ type-definition-part ]
            [ variable-declaration-part ]
              [ procedure-and-function-declaration-part ]
                statement-part

label-declaration-part = "LABEL" label {"," label} ";"
constant-definition-part = "CONST" constant-definition ";"
                           {constant-definition ";"}
type-definition-part = "TYPE" type-definition ";"
                       {type-definition ";"}
variable-declaration-part = "VAR" variable-declaration ";"
                            {variable-declaration ";"}
procedure-and-function-declaration-part =
  {(procedure-declaration | function-declaration) ";"}
statement-part = compound-statement
```

Constant-Definitions (see Section 6)

```
constant-definition = identifier "=" constant
constant = constant-identifier
         | signed-number
         | quoted-string-constant
constant-identifier = identifier
```

Type-Definitions (see Section 7)

```
type-definition = identifier "=" type
type = simple-type
     | structured-type
     | pointer-type

simple-type-identifier = type-identifier
structured-type-identifier = type-identifier
pointer-type-identifier = type-identifier
type-identifier = identifier

simple-type = ordinal-type
            | real-type
            | string-type
ordinal-type = enumerated-type
             | subrange-type
             | ordinal-type-identifier
ordinal-type-identifier = type-identifier
real-type = real-type-identifier
real-type-identifier = type-identifier

string-type = "STRING" "[" size-attribute "]"
size-attribute = unsigned-integer

enumerated-type = "(" identifier-list ")"
identifier-list = identifier { "," identifier }

subrange-type = constant ".." constant
```

```
structured-type = ["PACKED"] unpacked-structured-type
                | structured-type-identifier
unpacked-structured-type = array-type
                         | set-type
                         | file type
                         | record type

array-type = "ARRAY" "[" index-type { "," index-type } "]"
             "OF" component-type
index-type = ordinal-type
component-type = type

record-type = "RECORD" [field-list [";"] ] "END"
field-list = fixed-part [ ";" variant-part  ]
           | variant-part
fixed-part = record-section { ";" record-section }
record-section = identifier-list ":" type
identifier-list = identifier { "," identifier }
variant-part = "CASE" [tag-field ":" ] tag-type "OF"
               variant { ";" variant }
tag-field = identifier
variant = case-constant-list ":"
          "(" [ field-list [";"] ] ")"
tag-type = ordinal-type-identifier
case-constant-list = case-constant { "," case-constant }
case-constant = constant
field-identifier = identifier

set-type = "SET" "OF" base-type
base-type = ordinal-type (except longint)

file-type = "FILE" ["OF" type]
          | "TEXT"

pointer-type = "^" type-identifier
             | pointer-type-identifier
```

Variable-Declarations and Variable-References (see Section 8)

```
variable-declaration = identifier-list ":" type
identifier-list = identifier { "," identifier }

variable-reference = variable-identifier
                   | file-buffer
                   | pointer-reference
                   | variable-reference selector
variable-identifier = identifier
selector = index
         | field-designator
index = "[" expression {"," expression} "]"
field-designator = "." field-identifier
field-identifier = identifier
file-buffer = file-variable-reference up-arrow
file-variable-reference = variable-reference
up-arrow = "^"
pointer-reference = pointer-variable up-arrow
pointer-variable = variable-reference
```

Procedure and Function Declarations (see Section 9)

```
procedure-declaration = procedure-heading ";" body
body = procedure-block
     | "FORWARD"
     | "EXTERNAL"
procedure-block = block

procedure-heading = "PROCEDURE" identifier
                    [ formal-parameter-list ]
procedure-identifier = identifier

function-declaration = function-heading ";" body
body = function-block
     | "FORWARD"
     | "EXTERNAL"
function-block = block
```

```
function-heading = "FUNCTION" identifier
                     [formal-parameter-list] ":" result-type
function-identifier = identifier
result-type = simple-type-identifier
            | pointer-type-identifier

formal-parameter-list = "(" parameter-section
                          {";" parameter-section} ")"
parameter-section = ["VAR"] parameter-group
                  | procedure-heading
                  | function-heading
parameter-group = identifier-list ":" type-identifier
identifier-list = identifier { "," identifier }
parameter-identifier = identifier
```

Expressions (see Section 10)

```
unsigned-constant = unsigned-number
                  | quoted-string-constant
                  | constant-identifier
                  | "NIL"
factor = [@-operator] variable
       | unsigned-constant
       | function-call
       | set-constructor
       | "(" expression ")"
       | not-operator factor
term = factor { multiplying-operator factor }
simple-expression = [ sign ] term { adding-operator term }
sign = "+" | "-"
expression = simple-expression
               [ relational-operator simple-expression ]

multiplying-operator = "*" | "/" | "DIV" | "MOD" | "AND"
adding-operator = "+" | "-" | "OR"
relational-operator = "=" | "<>" | "<" | ">" | "<="
                    | ">=" | "IN"
@-operator = "@"
not-operator = "NOT"
```

```
function-call = function identifier
                  [ actual-parameter-list ]
function-identifier = identifier
actual-parameter-list = "(" actual-parameter
                          { "," actual-parameter } ")"
actual-parameter = expresssion
                 | variable-reference
                 | procedure-identifier
                 | function-identifier

set-constructor = "[" [ member-group
                    { "," member-group } ] "]"
member-group = expression [ ".." expression ]
```

Statements (see Section 11)

```
statement = [[ label ":" ] ( simple-statement
                           | structured-statement )]
label = unsigned-integer

simple-statement = assignment-statement
                 | procedure-statement
                 | goto-statement
assignment-statement = ( variable | function-identifier )
                         ":=" expression
procedure-statement = procedure-identifier
                        [ actual-parameter-list ]
procedure-identifier = identifier
goto-statement = "GOTO" label

structured-statement = compound-statement
                     | conditional-statement
                     | repetitive-statement
                     | with-statement

compound-statement = "BEGIN" [statement { ";" statement }]
                       "END"

conditional-statement = if-statement
                      | case-statement

if-statement = "IF" expression "THEN" statement
                 [ else-part ]
else-part = "ELSE" statement
```

```
case-statement = "CASE" expression "OF"
                   case-list-element
                   {";" case-list-element }
                   [";" "OTHERWISE" statement ]
                   [";"] "END"
case-list-element = case-constant-list ":" statement
case-constant-list = constant {"," constant}

repetitive-statement = repeat-statement
                     | while-statement
                     | for-statement

repeat-statement = "REPEAT" [statement { ";" statement }]
                     "UNTIL" expression

while-statement = "WHILE" expression "DO" statement

for-statement = "FOR" control-variable ":=" initial-value
                  ( "TO | "DOWNTO" ) final-value
                  "DO" statement
control-variable = variable-identifier
initial-value = expression
final-value = expression

with-statement = "WITH" record-variable-list "DO"
                   statement
record-variable-list = record-variable
                         { "," record-variable }
record-variable = variable-reference
```

Textfile I/O (see Section 12)

```
READ-parameter-list = "("[file-variable ","] variable-reference
                        {"," variable-reference}")"
file-variable = variable-reference

READLN-parameter-list = ["(" (file-variable | variable-reference)
                          {"," variable-reference} ")"]

write-parameter-list = "("[file-variable ","] write-parameter
                         {"," write-parameter}")"
write-parameter = expression [":" expression
                    [":" expression ] ]
```

```
writeln-parameter-list = ["(" (file-variable
                                | write-parameter)
                              {"," write-parameter}")"]
```

Programs (see Section 13)

```
program = program-heading ";" [ uses-clause ";" ] block "."
program-heading = "PROGRAM" identifier
                    [ "(" program-parameters ")" ]
program-parameters = identifier-list
uses-clause = "USES" identifier-list
identifier-list = identifier {, identifier}
```

Units (see Section 14)

```
unit = unit-heading ";"
         [intrinsic-clause ";"]
         interface-part
         implementation-part
         "END" "."
unit-heading = "UNIT" identifier
intrinsic-clause = "INTRINSIC" ["SHARED"]
interface-part = "INTERFACE"
                 [ uses-clause ]
                 [ constant-definition-part ]
                 [ type-definition-part ]
                 [ variable-declaration-part ]
                 [ procedure-and-function-declaration-part ]
implementation-part = "IMPLEMENTATION"
                 [ constant-definition-part ]
                 [ type-defintion-part ]
                 [ variable-declaration-part ]
                 [ procedure-and-function-declaration-part ]
```

[file pas:appc; date 2/19/82]