I. The Pascal Language

NAME

Introduction - the Pascal compiler

FUNCTION

The Pascal compiler is a set of four programs that take as input, to the first of the programs, one or more files of Pascal source code, and produce as output, from the last of the programs, assembler code that will perform the semantic intent of the source code. Output from the files may be separately compiled, then combined at link time to form an executable program; or Pascal subroutines can be compiled for later inclusion with other programs. One can also look on the compiler as a vehicle for implementing an instance of an abstract Pascal machine, a machine that executes statements in the language defined by some standard. That standard is generally accepted to be Jensen and Wirth, "Pascal User Manual and Report", Springer-Verlag 1978. Where ambiguous or simply uninformative, this standard has been supplemented in defining the current implementation by the ISO draft proposal for standardizing Pascal (ISO/TC 97/SC 5 N 595, January 1981).

This section of the present manual describes the current implementation, as succinctly and precisely as possible. It is organized into loosely coupled subsections, each covering a different aspect of the language. No serious attempt is made to be tutorial; the interested student is referred to Jensen and Wirth, then back to this manual for a review of the differences between this implementation of Pascal and the language described by the language standard.

The recommended order of reading the following subsections is:

Introduction - this is it.

Syntax - how to spell the words and punctuate the statements.

Identifiers - the naming of things, the scope of names and their important attributes.

Declarations - how to introduce identifiers and associate attributes with them.

Statements - how to specify executable code.

Expressions - the binding of operators, order of evaluation, and coercion of types.

Style - recommendations for what parts of the language to use and what to avoid; how to format code.

Differences - comparative anatomy of this implementation and the language standard.

Diagnostics - things the compiler complains about.

NAME

Syntax - syntax rules for Pascal

FUNCTION

At the lowest level, a Pascal program is represented as a text file, consisting of lines each terminated by a newline character. Any characters appearing between '{' and '}', including newlines, are treated as comments and replaced by a single space. The character sequences "(*" and "*)" also delimit comments.

Text lines are broken into tokens, strings of characters possibly separated by whitespace. Whitespace consists of one or more non-printing characters, such as space, tab, or newline; its sole effect is to delimit tokens that might otherwise be merged. Tokens take several forms:

An identifier - consists of a letter, followed by zero or more letters or digits. Uppercase letters are not distinct from lowercase letters; no more than eight characters are significant in comparing identifiers other than the keywords listed below. More severe restrictions may be placed on external identifiers by the world outside the compiler (see Differences). The following are the identifiers reserved for use as keywords:

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

A directive - consists of either of the identifiers "external" or "forward". Directives are used to indicate that the body of a function or procedure appears elsewhere.

An integer constant - consists of a decimal digit, followed by zero or more digits.

A real constant - consists of a decimal integer part, a decimal point '.', a decimal fraction part, and an exponent, where an exponent consists of an 'E' and an optionally signed ('+' or '-') decimal power of ten by which the integer part plus fraction part must be multiplied. Either the decimal point and fraction or the exponent may be omitted, but not both. Any real constant not in this form is illegal.

A character constant - consists of a single quote, followed by a one character literal, followed by a second single quote. The character literal for a single quote is written as two single quotes. A character constant is of the pre-defined type char, its value being the ASCII representation of that character.

A string constant - is just like a character constant, except that more than one character is contained inside the quote marks. More precisely, it is considered to be a packed array of type char.

Punctuation - consists of pre-defined strings of one or two characters. The complete set of punctuation for Pascal is:

```
+       <>      .
-       >=      :
*       >       ^
/       :=      (
<       ,       )
<=      ;       [ or (.
=       :       ] or .)
```

Other characters - are illegal outside of comments, character constants, or string constants; their appearance elsewhere causes an error.

NOTATION

Grammar, the rules by which the syntactic elements above are put together, permeates the remaining discussions. To avoid long-winded descriptions, some simple shorthand is used throughout this section to describe grammatical constructs.

A name enclosed in angle brackets, such as <statement>, is a "metanotion", i.e., some grammatical element defined elsewhere. Presumably any sequence of tokens that meets the grammatical rules for that metanotion can be used in its place, subject to any semantic limitations explicitly stated. Just about any other symbol stands for itself, i.e., it must appear literally, like the semicolon in

<statement> ; <statement>

Exceptions are the punctuation "[", "]", "]*", "{", "}", and "|"; these have special meanings unless made literal by being enclosed in single quotes.

Brackets surround an element that may occur zero or one time. The optional occurrence of a label, for instance, is specified by:

[<label> :] <statement>

This means that the metanotion <label> may but need not appear before <statement>, and if it does must be followed by a literal colon. To specify the optional, arbitrary repetition of an element, the notation "[]*" is used. A comma separated list of <ident> metanotions, for example (i.e., one instance of <ident> followed by zero or more repetitions), would be represented by:

<ident> [, <ident>]*

Vertical bars are used to separate the elements in a list of alternatives, exactly one of which must be selected. The line:

```
boolean | char | integer | real
```

requires the specification of any one of the four keywords listed. Such a list of alternatives is enclosed in braces in order to precisely delimit its scope. For instance:

```
<ex1> { to | downto } <ex2>
```

indicates that either a literal "to" or a literal "downto" must appear between <ex1> and <ex2>.

NAME

Identifiers - naming things in Pascal

FUNCTION

Identifiers are used to give names to the objects created in a Pascal program. Syntactically, an identifier is a sequence of letters and digits, starting with a letter. In differentiating identifiers, only their first eight characters are significant, and cases are not distinguished, so that "Summation" and "summations" are the same identifier.

Each identifier acquires, from its usage in a Pascal program, a precisely defined scope, storage class, and type. Its scope is the extent of program text over which a given declaration of the identifier is known to the compiler. In this implementation of Pascal, the storage class of an identifier indicates whether or not it is declared in the current source program, and, if it is, whether it will be made available to other, separately compiled programs. The type of an identifier determines what operations can be performed on it (and in conjunction with what other operands), as well as the internal encoding of the object associated with it.

"Identifier", as used in the remainder of this subsection, implies a name associated with some object that occupies storage at run-time, that is, code or data. In Pascal, three other classes of identifiers exist: those naming constants, labels, and types. The use and meaning of these identifiers are described under Declarations; here, it suffices to note that they are scoped in the manner described below. Considerations of storage class are irrelevant to them, as they are always local to the file in which they are declared, and can never be imported from other files.

SCOPE

Central to the scoping of Pascal identifiers is the notion of a block. Roughly speaking, every Pascal source file constitutes a block, as does each subprogram declared in the file. The useful scope of an identifier extends from the point of its declaration through the end of the block in which it is declared; that is, any use of an identifier that is to refer to a given declaration of it must textually follow that declaration. However, the declaration of an identifier is ultimately known throughout the block containing it, so that any textually preceding use of the same identifier meant to refer to some previous declaration will conflict with the later declaration (and cause an error).

An instance of an identifier is automatically known in all blocks nested within the one in which it is declared, though it may be redeclared in any of the nested blocks. If it is, then the closest statically nested declaration will apply to it, subject to the glitch explained above.

Three exceptions exist to the rule that the declaration of an identifier must precede any other use of it. Members of the list of identifiers given in the program statement of a source file need not be declared within the file, though if they are, their declaration must precede any

other use. In addition, the declaration of a pointer type may precede that of the type it points to, so long as the latter occurs later in the type declarations for the same block. Finally, the directive forward may be used to declare a subprogram at a point preceding its full definition, so long as the latter appears in the same block. Refer to Declarations for further details on all of these exceptions.

STORAGE CLASSES

An identifier has one of two storage classes: either it is defined (and storage reserved for its object) within the current source file, or it is imported from another file.

An identifier may be named as externally defined (and hence to be imported) either by mentioning it in the list of variables in a program statement (about which see Declarations), or, in the case of a function or procedure name, by replacing its body with the directive "external".

Identifiers defined in a given source module are implicitly classed as either global or local. If the program statement of the source file contains no program name, then the file is taken to be a "library" module defining code or data to be accessed from other files, and the compiler makes globally known all of the variable, procedure, and function declarations occurring in the outermost block of the file. Note in this regard that the ultimate handling of identifier names communicated between modules is dependent on the external operating system, which may restrict them further than the compiler does. (See Differences for a system-specific summary.)

Except for those just mentioned, all identifiers declared in a Pascal program are local to the current source file. The objects they name are, effectively, dynamically allocated on entry to the block in which they are defined, and deallocated on exit. The value of an object declared in a block is therefore undefined upon each entry to the block, and lost between activations. In blocks other than the outermost one, multiple instances of such an object may exist simultaneously, one for each concurrent activation of the block.

TYPE

All types in Pascal are built from the so-called "simple types", which consist of ordinal types and the pre-defined type real. All simple types determine an ordered series of scalar values; ordinal types, in addition, ultimately determine a contiguous sequence of integers. Ordinal types are the pre-defined type integer, enumerated types, and subrange types.

An enumerated type consists of a series of identifiers each of which is associated with a value according to its position in the series: the first with the value zero, the second with the value one, and so on. A subrange type designates some range of values within the complete range of values contained in another ordinal type, and is specified by naming the smallest and largest value in the subrange.

Pascal provides four pre-defined (or "standard") simple types:

boolean - is an enumerated type with the two values false and true.

char - is an enumerated type whose values are the codes for the ASCII character set, e.g. character constants.

integer - is the subrange of the counting numbers designated by the compiler-defined interval -maxint .. maxint. maxint is typically 32766 [sic], but may be set to a wide range of values at compile time. Its value is typically chosen to be large enough to count essentially all the bytes in the address space of the target machine (though so mundane a characteristic is irrelevant to Pascal).

real - is a floating-point number of standard precision, typically about sixteen digits. For storage economy, a six-digit representation for real may be specified at compile time.

The pre-defined types boolean and char have special meaning to the language, in that all character constants have type char, and all logical expressions have type boolean.

From the simple types may be built the structured types of array, record, set and file, each of which consists of some ordered succession of simple types or of other structured types, as explained below. Recursive application of the rules for deriving structured types can lead to a large, if not infinite, assortment.

an array - is a linear sequence of identical elements, each of which is an instance of some other type, and any of which may be accessed by an index giving its position in the sequence. An array is indexed by any ordinal type, the number of its elements being the number of distinct values that the index type may assume.

a record - consists of a fixed number of components, called fields, each of which may be an instance of any other type. A record may end with a "variant part", that is, with differing sets of fields, depending on the value assigned to a designated "variant selector", or "tag field".

a set - is a collection of boolean indicators (not present/present), one for each distinct value that may be assumed by some ordinal type, called the "base type" of the set.

In this implementation, all values in the base type of a set are presumed to lie in the inclusive range from zero to some positive maximum specifiable at compile-time. The base types of two sets are equivalent if identical, or if both are subranges of the same type; sets whose base type is a subrange are implemented as sets of the full ordinal type from which the subrange is derived, subject to the compile-time limits just mentioned. Sets of 32 or fewer elements are represented as a single integer of the requisite length; larger sets become arrays of bytes, and are stored eight elements per byte.

a file - is a linear sequence of identical elements, each of which is an instance of some other type, and which may be accessed one at a time, starting with the first. Access may be either for writing all elements (after a rewrite call), or for reading some elements (after a reset call). (See Section II for details, and Section III for extensions to these standard capabilities.)

The elements may be of any type except a file type, or a structured type involving a file type. A pre-defined type text exists that is declared to be file of char. Most operating systems distinguish between "binary" files, which faithfully record the internal representation of elements, and "text" files, which are made presentable to people at terminals and printers. In this implementation, only type text can reliably be connected to a text file.

Access to variables of type file is through an associated "buffer variable" that, for an input file, contains the value currently available for input, or, for an output file, serves as the location from which the next value output will be obtained. Three sets of routines assist the manipulation of files: the pre-defined Pascal routines described in Section II, the extra interface routines described in Section III, and the I/O routines in the portable C interface, which may be called from within a Pascal program.

In addition to the structured types, Pascal provides pointer types, each of which is associated with ("points to") some other type. The type pointed to, or "domain type", may be any of the types described above, or a second pointer type. Thus pointers may take on whatever values are needed to locate objects within the address space of the target computer. A special value nil may be assigned to any pointer to ensure that it points at no existing object.

The pre-defined function new allocates space for an object of the domain type of the pointer, assigning to its pointer argument a value that uniquely identifies the object, and is guaranteed not to equal nil. The pre-defined function dispose deallocates space previously allocated by new, and discredits the pointer identifying it. Section II has further details.

NAME

Declarations - declaring names in Pascal

FUNCTION

A Pascal program consists of a single block, preceded by the program heading described below. In general, a block contains parts declaring all the identifiers (labels, symbolic constants, types, variables, procedures, and functions) defined by it, followed by a single compound statement that specifies the code executed on entry to it. All parts of a block except for the concluding statement are optional. If no identifiers of a given class are to be defined within the block, the corresponding part is omitted.

Thus a <block> has the format:

```
[ <labelpart> ]
[ <constpart> ]
[ <typepart> ]
[ <varpart> ]
[ <functiondec> | <proceduredec> ]*
<compound-statement>
```

This subsection explains the content of each of the parts listed, except for the final <compound-statement>, which is explained in the subsection on Statements.

THE OUTERMOST BLOCK

A Pascal source file has the form:

```
program [ <ident> ] [ ( <pident> [, <pident> ]* ) ] ;
    <block> .
```

If <ident> is present, it is taken as the name of the program, and a full-fledged <block>, as defined above, is presumed to follow the program statement. If <ident> is absent, the source file is assumed to be a "library file" defining code or data for use elsewhere. In this case, all variable, procedure, and function identifiers defined at the outermost level of the <block> are made externally known, and may be imported into other source files. In a library file, the outermost <block> must not contain a <labelpart> or a <compound-statement>.

Each <pident> is an identifier that is to be imported into the current program from another file, except for certain file variables as described below. The standard files input and output, if used by the program, must be listed here, and must not be declared in <varpart>. Any other identifiers listed that are not declared in <varpart> are also assumed to be imported, and of type text.

In a main program (one whose program statement contains an <ident>), if a <pident> is explicitly declared to be of a file type, then no external object of that name is imported. Instead, the file is declared locally, and an external filename is associated with it at the start of each run. By

default, this name is <pident>, but any arguments given on the command line by which the program is invoked effectively overwrite the default name. If at least n arguments are given, and the nth <pident> is declared as a file variable, then the nth command line argument is taken as the name of the file to be associated with it.

Only files named in the program statement of a main program are permanent. All other files are given private names on creation and are removed when their defined scope is exited.

For example:

```
program usefiles(datin, datout, extfil, input);

    ...

var
    datin:  filetype;
    datout: filetype;

    ...
```

This excerpt from the main program usefiles would associate each of the file variables datin and datout with an external file, named by the first and second argument from the command line with which usefiles was run. It also imports the file variable extfil from some other source file, and specifies that the pre-defined file variable input will be used. Because extfil is not explicitly declared here, it is assumed to be of type text; it will be associated with the third command line filename given.

LABEL DECLARATIONS

Labels may be prefixed to any statement within the <compound-statement> that terminates each block, and may serve as the targets of unconditional goto statements, within certain constraints described in the subsection on Statements.

A <labelpart> has the following form:

label <labno> [, <labno>]* ;

A <labno> is an unsigned integer between 0 and 9999 inclusive that must be unique within the scope of the declaration.

CONSTANT DECLARATIONS

Pascal allows the user to parameterize manifest constants by associating each with an identifier that is subsequently usable in place of the constant.

The <constpart> naming these associations has the following form:

```
const <ident> = <constant> ;
    [ <ident> = <constant> ; ]*
```

Each <ident> present may be any identifier, while a <constant> must be either an identifier that is a component of an enumerated type, or a numeric, string or character constant, as defined in Syntax. Within the scope of the declaration, all occurrences of each <ident> will be replaced by the corresponding <constant>.

TYPE DECLARATIONS

Any block may define new types based on previously declared ones. The form of the <typepart> used to do so is:

```
type <ident> = <ty> ;
    [ <ident> = <ty> ; ]*
```

<ident> is made to be a synonym for <ty>, which is some type already known in the current block, or a new type constructed from one or more of the following components:

boolean | char | integer | real - specify one of the pre-defined simple types.

(<eident> [, <eident>]*) - specifies the components of an enumerated type. For example:

```
feel = (flaccid, soft, bouncy, firm, petrous);
```

<minval> .. <maxval> - specifies a subrange of values drawn from some previously-defined ordinal type, <minval> naming the value in the new subrange with the lowest ordinal position, and <maxval> the value with the highest. The subrange must include at least one value. A few examples:

```
salable = soft .. firm;
boffset = -128 .. 127;
```

[packed] array '[' <range> ']' of <ty> - where '[' and ']' represent actual brackets, specifies an array each element of which is of type <ty>. <range> is an ordinal type specifying the range of indices that may used to access the array. A multidimensional array may appear either as a series of discrete array declarations, or as a single declaration in which <range> is a series of ordinal types separated by commas, the series enclosed in square brackets. Hence the following declarations are equivalent:

```
sales = array [feel, size, salesman] of integer;
sales = array [feel] of array [size]
    of array [salesman] of integer;
```

[packed] file of <ty> - specifies a file consisting of values each having type <ty>. <ty> must not contain any element of a file type.

[packed] set of <range> - specifies a set of boolean membership indicators, one for each of the values in <range>, which may be any ordinal type.

In addition to the components given above, <ty> may also be constructed from one or more record definitions, of the format:

```
[ packed ] record [ <field> ; ]*
    [ <field> | <variant> ]
end
```

where a <field> is defined as:

```
<fident> [, <fident> ]* : <ty>
```

Each <fident> given specifies a single field of type <ty>, to be contained in each instance of the record; if two consecutive fields have the same type, the name of the first may precede that of the second, the two separated by a comma, both sharing the same type specifier.

The final field of a record may be replaced with a <variant>, that is, with a set of fields that varies according to the value of a previous "tag field". Such a <variant> has the format:

```
case [ <tident> : ] <ttype> of
  [ <tval> [, <tval> ]* : ( [ <field> ; ]* [ <field> ] ) ; ]*
    <tval> [, <tval> ]* : ( [ <field> ; ]* [ <field> ] )
```

where a <field> is as defined above. <tident>, if given, is defined as a tag field of the ordinal type <ttype>. The value assigned to it selects the <field> list in the variant having a matching <tval>, which then becomes the list of fields declared for the variant. The field list selected changes dynamically with the value assigned to <tident>. If <tident> is not given, then a field from any of the field lists in the variant may be used to access the storage that the variant occupies, with machine-dependent but often useful results. Whether or not <tident> appears, however, all of the values of <tval> must be unique.

Note that the keyword "packed" is largely ignored by this implementation; it causes no change in the allocation of data storage, but its presence or absence is checked for in those constructs where the language standard requires it.

Finally, a type declaration may involve a pointer definition, of the format:

```
^<tyident>
```

where <tyident> is a type identifier that is declared elsewhere in the same <typepart> [sic]. This rule permits cross-referential types, but at the cost of such useful constructs as:

```
type
    pc = ^integer;
```

A compile-time option permits the relaxation of this rule.

VARIABLE DECLARATIONS

A variable declaration consists of one or more identifiers followed by an instance of a type, and declares each identifier to represent a data object of the type given.

The section containing variable declarations takes the form:

```
var   <ident> [, <ident> ]* : <ty> ;
    [ <ident> [, <ident> ]* : <ty> ; ]*
```

FUNCTION AND PROCEDURE DECLARATIONS

The declaration of a function or a procedure begins with a heading defining its name and formal parameters, and, in the case of a function, the type of its return value. This heading is followed either by a <block>, as defined above, giving the body of the subprogram, or by one of the directives external or forward. The directive external indicates that the body of the subprogram is defined in some other source file; the directive forward indicates that the body is defined at some later point in the current block.

Thus a <proceduredec> has the form:

```
procedure <ident> [ <formallist> ] ;
    { <block> | external | forward } ;
```

while a <functiondec> looks like:

```
function <ident> [ <formallist> ] : <tyident> ;
    { <block> | external | forward } ;
```

where <tyident> names the return type of the function, which must be a type already known in the current block. Note that functions may return values only of a simple or pointer type. If the directive forward appears, then at some later point in the same block the body of the subprogram must be given, preceded by a second procedure or function heading giving only the name of the subprogram, and omitting any <formallist> or return type.

In either kind of declaration, a <formallist> has the format:

```
( <formaldec> [ ; <formaldec> ]* )
```

where each <formaldec> has one of the forms:

```
{ <ident> [, <ident> ]* : <tyident> |
  var <ident> [, <ident> ]* : <tyident> |
  procedure <ident> [ <formallist> ] |
  function <ident> [ <formallist> ] : <tyident> }
```

In the first two forms, <ident> denotes a formal parameter that generally may used within the subprogram as if it were a variable declared in the subprogram body. Each <tyident> must be an identifier naming a type already known in the current block. If two consecutive parameters have the same type, the name of the first may precede that of the second, the two separated by a comma, both sharing the same type specifier. If the keyword var is present, then the parameters in that <formaldec> are passed by reference, i.e., changing the contents of the formal parameter causes the actual parameter variable to change. Otherwise, the parameters are passed by value. Note that parameters of a file type may be passed only by reference.

The last two forms of <formaldec> permit either procedures or functions to be passed to a subprogram as actual parameters. A reference to the formal parameter within the subprogram body then causes a call to the the actual procedure or function passed to the subprogram. The environment surrounding the passed subprogram is also secretly memorized at the time of such a call, so that any reference in the called subprogram to an object declared in one of its dynamic ancestors will access the most recent dynamically declared instance, as of the point at which the called subprogram was passed as a parameter.

NAME

Statements - the executable code

FUNCTION

Every block contained in a Pascal source file, except for the outermost block of a library file (see Declarations), normally ends with a compound statement specifying the action to be taken on entry to the block.

A compound statement specifies a series of statements to be executed sequentially, and has the format:

```
begin
    [ <statement>
    [; <statement> ]* ]
end
```

where a <statement> is one of the following simple statements or another compound statement. Wherever the syntactic descriptions below call for a <statement> to be given, a simple or a compound statement is equally acceptable.

; - is taken as a null statement, and has no effect.

<labno> : <statement> - <labno> is interpreted as a label identifying the beginning of the statement, and may serve as the target of a goto statement, within the constraints explained below.

<var> := <expr> - assigns the value of <expr> to the object indicated by <var>. <var> may denote a variable of any simple type, a pointer type, or the structured types array, record, or set; it may also denote a component of an array or record. In addition, <var> may denote the buffer variable associated with a file, or the data object pointed to by a pointer; in either of these cases, the name given is followed by a caret to indicate its special meaning. <var> may not designate a variable of a file type, or a structured variable having a component of a file type.

Functions are also assigned return values with a statement of this form; <var> in this case is the function name. This kind of assignment statement may appear only in the body of the function named, and <expr> may be only of a simple or pointer type, since functions may return values only of those types.

<expr> is an expression formed by the rules explained in the subsection on Expressions. The value of <expr> must be "assignment-compatible" with the type of the object designated by <var>. That is, the value must have the same type as <var>, or one of the following conditions must hold:

1) <var> is of type real, <expr> of type integer.

2) the types of <var> and <expr> are subranges of the same ordinal type, or the type of one is a subrange of the type of the other,

and the value of <expr> lies in the range of values determined by the type of <var>.

3) the types of <var> and <expr> are set types based on the same ordinal type, or on subranges of the same ordinal type, and all the members of the set specified by <expr> are included in the type of <var>. Either both set types are designated packed, or neither is.

4) the type of <var> and <expr> is packed array of char, and each array contains the same number of components, with an initial index of one.

<pcident> [(<actuallist>)] - is an invocation of the procedure <pcident>, already known in the current block. The <actuallist>, if present, names the actual parameters to be passed to it, and takes the form:

<actual> [, <actual>]*

where each <actual> is either an expression, a <var> (as defined above), or an identifier naming a currently known function or procedure.

The actual parameters specified must correspond in number and type to the formal parameters given in the declaration of the procedure called. An actual corresponding to a formal value parameter must be assignment-compatible with the formal; an actual corresponding to a formal declared as var must be a <var>, while those corresponding to formals declared as function or procedure must be identifiers naming an appropriate subprogram.

if <boolexpr> then <statement> [else <statement>] - causes the conditional execution of a <statement>, based on the value of <boolexpr>, which must be an expression of type boolean. If <boolexpr> has the value true, the statement following the keyword then is executed. The statement following the keyword else, when present, is executed if <boolexpr> has the value false. An else clause is always attached to the closest preceding if-then pair not already matched with an else.

case <expr> of <caseitem> [; <caseitem>]* end - selects a statement to be executed according to the value of <expr>, which may be of any ordinal type. Each <caseitem> associates a statement with one or more values from the type of <expr>, and takes the form:

<cvalue> [, <cvalue>]* : <statement>

A given <statement> is selected if a <cvalue> in the same <caseitem> matches the value of <expr> on entry to the case statement. After the <statement> chosen is executed, execution resumes following the case statement.

Each <cvalue> must be unique from all others in the same case statement, and must evaluate to a constant whose value is known at compile-time. If the value of <expr> does not match any <cvalue> given, a range error occurs.

while <boolexpr> do <statement> - causes <statement> to be executed zero or more times, according to the value of <boolexpr>, which must be an expression of type boolean. The expression is evaluated before each execution of <statement> (including the first one). If it yields the value true, <statement> is executed; if not, <statement> is skipped, and execution resumes following the while statement.

repeat <statement> [; <statement>]* until <boolexpr> - causes the sequential execution of each <statement> given, then the evaluation of <boolexpr>, which must be an expression of type boolean. If the expression yields the value false, this process is repeated; otherwise, execution continues following the repeat statement.

for <ident> := <ex1> { to | downto } <ex2> do <statement> - executes <statement> zero or more times, while incrementing (or decrementing) <ident> from the value of <ex1> to (down to) the value of <ex2>. <ident> denotes a variable of ordinal type declared in the current block that is not a component of any other variable. <ex1> and <ex2> are expressions either of the same type as <ident>, or, if any of the three has a subrange type, of the ordinal type on which the subrange is based.

If <ex1> is less than or equal to <ex2> (greater than or equal to, when downto is specified), <ident> is set equal to it, and <statement> is executed. <ident> is then assigned the following (or preceding) value in the ordinal type; if it is still within the bound imposed by <ex2>, then <statement> is again executed, and the process repeated. Whenever the bound is exceeded (either by <ex1> or by <ident>), execution continues at the end of the for statement.

<ex1> and <ex2> are evaluated only once, before the first execution of <statement>. <ident> may not be assigned to in any way within <statement>. When the for statement terminates, unless it is left via a goto statement, <ident> becomes undefined.

goto <labno> - causes execution to continue at the start of the statement labelled with <labno>, which must be a label already known in the current block. The label is accessible to a goto, however, only if one of the following conditions holds:

1) <labno> labels the statement containing the goto statement.

2) within the closest compound statement containing the goto statement, <labno> labels another statement.

3) within the <compound-statement> that ends some block enclosing the block in which the goto statement occurs, <labno> labels a statement at the outermost level of nesting.

The last rule explicitly permits a goto from within a procedure or function to a statement, at the outermost level of nesting, in a containing procedure or function. Such a goto causes early termination of all dynamic block activations up to that associated with the target label. Hence, files may be closed and expression evaluations may be left incomplete (and abandoned) as a side effect of the goto.

If no label <labno> is accessible to the goto statement, an error occurs.

with <recordvar> [, <recordvar>]* do <statement> - permits the fields of each record specified to be accessed in <statement> simply by naming them, without naming the record enclosing them. Each field is treated as if it were a discrete variable declared solely within the scope of <statement>.

Each <recordvar> is a variable of a record type known in the current block; any special actions needed to access the record (such as indexing an array or de-referencing a pointer) are performed only once, before <statement> is executed, and not with each reference to a field of the record.

NAME

Expressions - computing values in Pascal

FUNCTION

Expressions in Pascal consist either of a single factor, as defined below, optionally modified by unary operators, or pairs of such factors, each pair connected by a binary operator.

Pascal implements standard arithmetic operators for the types integer and real, as well as relational operators, logical connectives, and operations on sets. A single symbol often denotes differing operations according to the type of its operands. A four-level scale of precedence determines the order in which operations within a single expression are performed; operators fit into it as follows (in descending order of precedence):

1) the operator not.

2) the "multiplying operators": "*", "/", "mod", "div", "and".

3) signs (unary "+" or "-") and the "adding operators": "+", "-", "or".

4) relational operators: "<", "<=", "=", "<>", ">=", ">", "in".

All binary operators are left-associative, e.g., a+b+c is taken as (a+b)+c. Naturally, parentheses may be used to force a given order of evaluation of subexpressions, and override default precedences and associativity. Note, however, that this implementation makes few promises about the order of evaluation of operands to a single operator, and none about the order in which operations at the same level of precedence will be performed. Redundant computations may never be done at all.

FACTORS

A factor appearing in a Pascal expression may be either an unsigned constant, a function call, a set constructor, or a reference to a variable.

An unsigned constant consists of either a character, string, or unsigned numeric constant (as defined in Syntax), or an identifier naming a component of an enumerated type, or the pre-defined constant nil.

A function call has the format:

<fnident> [(<actuallist>)]

where <fnident> is the name of the function to be called, which must already be known in the current block, and <actuallist> has the same meaning as in the definition of a procedure call in Statements. The value and type of the factor are the return value and type of the function.

A set constructor builds a set from a series of expressions, and has the result as its value. It takes the form:

'[' [<smember> [, <smember>]*] ']'

where '[' and ']' represent actual brackets, and each <smember> has the format:

<sexpr> [.. <sexpr>]

where a <sexpr> is some expression of an ordinal type. Each <sexpr> given must be of the same effective type (see below), which becomes the base type of the set constructed. Two consecutive <sexpr> separated by a double dot denote the increasing ordinal values in the closed interval between the value of the first <sexpr> and the value of the second. The first <sexpr> may have a value greater than the second, in which case the two specify no values. A set constructor containing no <smember> has the null set as value, i.e., the null set is written as [ ].

A reference to a variable (call it <var>) may be arbitrarily complex, though it must be composed of some series of the following four constructs:

<ident> - references the value of the variable whose name is <ident>. If the variable is of a structured type, then naming it references its entirety.

<var>^ - references the data object pointed to by a pointer, or the buffer associated with a file. <var> must refer to an object of a pointer or file type.

<var>'[' <indexlist> ']' - where '[' and ']' represent actual brackets, references the array element specified by <indexlist>, which has the format:

<index> [, <index>]*

where each index is an expression of an ordinal type. The contents of <indexlist> must agree in number and type with the index ranges specified when the array was declared. <var> must refer to an object of an array type.

<var>.<f> - references the field <f> within the record referenced by <var>. <var> must reference an object of a record type, and <f> must be an identifier declared to be a field of the type.

If a factor is of a subrange type, then its effective type becomes the ordinal type from which the subrange was taken. Similarly, if a factor is of a set type whose base type is a subrange, then its effective type is the set type whose base is the ordinal type from which the subrange was taken. The effective type of a set of a subrange of integers is taken as "set of [0 .. maxset]", where maxset is a compile time parameter. This effective type is used in all subsequent processing.

OPERATORS

Following are the operators defined in Pascal, presented in descending order of precedence. x is any left operand, y any right operand. In general, x and y must have the same effective type, though for arithmetic operators one may be of integer type and the other real. If this holds, the integer operand is converted to real before the operation. All type specifications for operands refer to their effective types.

not y - y must be boolean. If y has the value false, the result is the boolean value true; otherwise it is the boolean value false.

x * y - if x and y are of a set type, the result is the intersection of the two sets (only elements common to both). Otherwise, x and y must be arithmetic, and the result is their product, of integer type if both x and y are integers, else of type real.

x / y - x and y must be arithmetic. The result is the quotient of x divided by y, and is of type real.

x div y - x and y must be of type integer. The result is the truncated quotient of x divided by y, and is of type integer.

x mod y - x and y must be of type integer. The result is the remainder obtained by dividing x by y, and is of type integer.

x and y - x and y must be of type boolean. If both x and y have the value true, the result is the boolean value true; otherwise, it is the boolean value false. In this implementation, if x has the value false, y is not evaluated.

+y - y must be arithmetic, and the result is y.

-y - y must be arithmetic. The result is its negation, and is of the same type.

x + y - if x and y are of a set type, the result is the union of the two sets (elements present in either). Otherwise, x and y must be arithmetic, and the result is their sum, of type integer if both x and y are integers, else of type real.

x - y - if x and y are of a set type, the result is the set difference of the two sets (elements of x with those in y removed). Otherwise, x and y must be arithmetic, and the result is y subtracted from x, of type integer if both x and y are integers, else of type real.

x or y - x and y must be of type boolean. If either x or y has the value true, the result is the boolean value true; otherwise, it is the boolean value false. In this implementation, if x has the value true, y is not evaluated.

x = y - x and y may be of any set, simple or pointer type; in addition, they may be of a "string type", that is, packed arrays of char of identical length with an initial index of one. If the values of x

and y are equal, the result is the boolean value true; otherwise, it is the boolean value false.

If x and y are of type char, or of any other enumerated type, they are compared based on their ordinal values in the type. If x and y are of a string type, they are compared character by character, left-to-right. These two statements hold for the other relational operators (except in) as well.

x <> y - x and y may be of any set, simple, pointer or string type. If the values of x and y are not equal, the result is the boolean value true; otherwise, it is the boolean value false.

x < y, x > y - x and y must be of a simple or string type. If the relation specified holds between the values of x and y, the result is the boolean value true; otherwise, it is the boolean value false.

x <= y - x and y may be of any set, simple or string type. If x and y are of a set type, the result is the boolean value true if x is included in y (all elements of x present in y); else it is the boolean value false. If x and y are of a simple or string type, the result is true if the value of x is less than or equal to that of y, otherwise false. If x and y are of type boolean, this operation tests the implication of y by x (y or not x).

x >= y - x and y may be of any set, simple or string type. If x and y are of a set type, the result is the boolean value true if y is included in x; else it is the boolean value false. If x and y are of a simple or string type, the result is true if the value of x is greater than or equal to that of y, otherwise false.

x in y - x must be of some ordinal type, y of a set type based on that type. If the value of x is a member of the set specified by y, the result is the boolean value true; otherwise, it is the boolean value false.

NAME

Style - rules for writing good Pascal code

FUNCTION

The following practices and restraints are recommended for writing good Pascal code:

ORGANIZATION

If a Pascal program totals more than about five hundred lines of code, it should be split into files each no bigger than that; a smaller limit may be needed on microcomputers. There must, of course, be just one main file, characterized by having a program name following the keyword program on the header line; all other files have no program name and hence export all declarations in their outermost block. By using the preprocessor pp that comes with the C compiler, it is possible to assemble each Pascal file from multiple sources, so that common declarations and definitions can be #include'd as needed. As much as possible, related declarations should be packaged together. A good convention is to use uppercase for all #define'd identifiers, as a warning to the reader that the language is being extended. For further information on pp, see Preprocessor in Section I of the C Programmers' Manual.

Within a large program, data declarations are typically clumped into related groups, e.g. all flags, all files, etc.; an explanatory comment should precede each group. Each function or procedure should be preceded by a one sentence comment that succinctly describes what it does. If the body of a function is sufficiently complex, a good explanatory comment is the half dozen or so lines of pseudocode that best summarize the algorithm. There is seldom a need for additional comments, but if they are used they are best placed to the right of the line being explained, separated by one tab stop from the end of the statement.

RESTRICTIONS

The goto statement should never be used. The only case that can be made for it is to implement a multi-level break, which is not provided in Pascal; but this seldom proves to be a prudent thing to do in the long run. If a subprogram has no goto statements, it has no need for labels. Another construct to avoid is the repeat-until statement, which frequently evolves into a safer while or for statement.

More than five levels of indenting (see FORMATTING below) is a sure sign that a subfunction should be split out, as is the case with a function body that goes much over a page of listing or requires more than half a dozen local variables. Naturally there are exceptions to all these guidelines, but they are just that -- exceptions.

Remember that, in portable Pascal, the operators "and" and "or" have no promised order of evaluation, even though they are strictly left to right with early termination in this implementation. Writing programs that depend upon this behavior may eventually cause problems.

If the relational operators > and >= are avoided, then compound tests can be made to read like intervals along the number axis, as in

```
if ((0 <= c) and (c <= 9)) then
    ...
```

which is demonstrably true when c is in the closed interval [0, 9].

FORMATTING

While it may seem a trivial matter, the formatting of a Pascal program can make all the difference between correct comprehension and repeated error. To get maximum benefit from support tools such as editors and cross referencers, one should apply formatting rules rigorously. The following high-handed dicta have proved their worth many times over:

All defining material -- data initializers or function bodies -- should be indented at least one tab stop, plus additional tab stops to reflect sub-structure. Tabs should be set uniformly every four to eight columns.

A function body, for instance, always looks like:

```
function fname(arg1, arg2, arg3: integer;
    arg4: real): integer;
    var
        loc1: integer;
        loc2: real;
    begin
        <statements>
    end;
```

Declarations are sorted alphabetically by type and alphabetically by name within type. Comma separated lists may be used.

<statements> are formatted to emphasize control structure, according to the following basic patterns:

```
if (test) then
    <statement>;

if (test) then
    begin
    <statement>;
    <statement>;
    ...
    <statement>
    end
else
    <statement>;
```

```
if (test1) then
    <statement>
else if (test2) then
    <statement>
else if (test3) then
    ...
else
    <default statement>;

case (value)
    of
A, B:
    <statement>;
C:
    <statement>;
D:;
    end;

while (test) do
    <statement>;

for <variable> := <expr> to <expr> do
    <statement>;

for <variable> := <expr> downto <expr> do
    <statement>;

while (true) do
    <statement>;
```

Note that, with the explicit exception of the else-if chain, each subordinate <statement> is indented one tab stop further to the right than its controlling statement.

Within statements, there should be no empty lines, nor any tabs or multiple spaces imbedded in a line. Semicolons should be used only when necessary as separators, since their presence in some contexts makes for hard to find control flow glitches. Each keyword should be followed by a single space, and each binary operator should have a single space on each side. No spaces should separate unary or addressing operators from their operands.

Parentheses should be used whenever there is a hint of ambiguity in an expression. Note that the relational operators are weaker than "and" and "or".

If an expression is too long to fit on a line (of no more than 80 characters) it should be continued on the next line, indented one tab stop further than its start. A good rule is to continue only inside parentheses, or with a trailing operator on the preceding line, so that displaced fragments are more certain to cause diagnostics.

NAME

Differences - comparative anatomy

FUNCTION

This implementation hews closely to the ISO standard, save for minor changes in emphasis. There are also many available implementations of Pascal that differ in more important respects. Herewith a summary of the things to look out for.

THE ISO STANDARD

- The major extension is that a "library file" may be written, with no program name, to specify declarations that may be separately compiled for import by other Pascal files. Functions and procedures are imported by declaring them with the "external" directive; variables are imported by being named in the parameter list in the program header.

- Conformant array parameters are currently disabled in this implementation, making it compatible with level 0 of the standard.

- Currently, there is no checking of accesses to variant records for proper tag values, nor is range checking performed on assignment to ordinal variables or sets of subranges. Range checking does occur, but may be disabled, for case values and for subscripts. Not all constraints on gotos and for variables are enforced.

- Only the first eight characters of an identifier are significant in this implementation. The ISO standard provides no limit beyond which conformance is guaranteed.

CONVERTING FROM OTHER PASCALS

This implementation is much more religious about enforcing ISO restrictions than most, and has virtually none of the popular extensions; so the biggest problems arise from programs that take advantage of various nonstandard features.

- ISO Pascal requires that a type declaration of the form:

```
type
    pi: ^integer;
```

  refer to a definition of the type identifier "integer" in the same type statement, thus prohibiting a useful type definition. A compiler flag defeats this restriction.

- The directive "external" in this implementation is often written "extern" in others.

- File names are often associated with file variables by the use of nonstandard reset and rewrite functions. Here, this association is

made either by connecting file variables in the main file program header to files named as command line parameters, or by using the special functions described in Section III of this manual.

- This implementation permits no "otherwise" clause in case statements.

CONVERTING TO OTHER PASCALS

Because this implementation is so restrictive, programs developed under it have a good chance of running unchanged in other environments. Some coding habits, however, can lead to problems.

- Extensive use of the preprocessor for #includes and #defines may necessitate using the expanded text (the output from the preprocessor), if pp is not available in the new environment.

- The separate compilation facilities provided here are rather unlike those, if any, provided in other Pascals, particularly in their provisions for sharing variable declarations.

- In this implementation, order of evaluation in boolean expressions involving "and" and "or" operators is strictly left to right, with early termination. Programs that depend on such behavior are likely to fail in other environments.

- Set sizes can be enormous, in this implementation, as can integers (on some machines). Typical set limits on other Pascals are 64 to 256 elements; integers are often limited to a magnitude of 32,767 or less.

CONVERTING AMONG MACHINES

Even staying within this implementation, Pascal can present problems when moving programs from one machine to another, or between operating systems on the same machine.

- External identifiers may collide with names that must be loaded from system interface libraries; and different systems may load differing sets of routines.

- External identifiers are truncated, on some systems, to as few as six characters, in one case of letters. On other systems, as many as eight characters are significant, in two cases of letters.

- The maximum integer value that may safely be used, a.k.a. MAXINT, varies surprisingly from machine to machine. To meet <u>all</u> requirements of the ISO standard, the following upper bounds are required:

MACHINE	MAXINT
8080	16,383
PDP-11	32,766
MC68000	2,147,483,646
VAX	2,147,483,646

- When writing binary files, the internal representation of arithmetic quantities becomes a public matter. A file of integers written on one machine may well give different values when read as a file of integers on another. Since most modern computers have eight-bit bytes and perform arithmetic in two's complement, the difference usually involves byte order or length of multibyte numbers. Thus, it is best to communicate among programs by text files only.

NAME

Diagnostics - compiler complaints

FUNCTION

The Pascal to C translator should produce all user diagnostics; if any diagnostics at all are produced, the resulting C program is not necessarily well formed. Diagnostics from the C compiler after a successful translation, or translator messages containing an exclamation mark ! or the word "panic" indicate problems with the compiler per se (they should "never happen") and hence should be reported to the maintainers. Here is a summary of the diagnostics that can be produced by erroneous Pascal programs:

```
bad argument list
bad array argument
bad array argument type
bad for variable <name>
bad pack arguments
bad proc argument
bad var argument type
can't create <file>
can't read <file>
duplicate case value
excess argument
file in value argument
file of files
goto into block
illegal .field <name>
illegal ^
illegal argument
illegal character buf
illegal compare
illegal constant initializer <name>
illegal i/o argument
illegal nil
illegal packed type
illegal set range
illegal string
illegal subrange
illegal subscript
illegal type
illegal with variable
label out of range <labno>
missing <thing>
no case variant
no function value <name>
not a record
not a type <name>
not defined type <name>
not local type <name>
not ordinal tag
not ordinal type
null string
ordinal type required
proc argument mismatch
real literal too long
redeclared <name>
redeclared function <name>
redefined argument <name>
redefined field <name>
redefined label <labno>
scope conflict <name>
set too large
string too long
subscript range
type too complex
undeclared <name>
undeclared file <file>
undeclared label <labno>
undefined ^type <name>
undefined forward <name>
unexpected EOF
variable required
write error
wrong case type
wrong tag type
```

Note that items in angle brackets stand for something whose actual name is provided in the diagnostic, as in "missing end". These are too numerous to detail.