PASCAL*: A PASCAL BASED SYSTEMS PROGRAMMING LANGUAGE

John Hennessy

TECHNICAL REPORT NO. 174

June 1980

The work described was supported by the Office of Naval Research, under a contract to the University of California Lawrence Livermore Laboratory, contract LLL PO no. 9628303

Pascal* : A Pascal Based Systems Programming Language†

John Hennessy
Computer Systems Laboratory
Departments of Computer Science and Electrical Engineering
Stanford University
Stanford, California 94305

Technical Report Number 174

June 1980

Abstract

Pascal* (Pascal-star) is a new programming language which is upward compatible with standard Pascal and suitable for systems programming. Although there are several additions to the language, simplicity remains a major design goal. The major additions reflect trends evident in newer languages such as Euclid, Mesa, and Ada, including: modules, simple parametric types, structured constants and values, several minor extensions to the control structures of the language, random access files, arbitrary return types for functions, and an exception handling mechanism.

Key Words and Phrases: systems programming, programming language, Pascal, modules, exception handling, parameterized types, language extension.

† This work described was supported by the Office of Naval Research, under a contract to the University of California Lawrence Livermore Laboratory, contract LLL PO no. 9628303.

1. Introduction

This report defines an upward compatible extension of the programming language Pascal [1,2]. This extension is designed to support the construction of large systems programs. A major attempt has been made to keep the orginal goals of Pascal intact; in particular, Pascal* strives for simplicity, efficient runtime implcmcntation, efficient compilation, language security, and compile-time checking. Many of the ideas in Pascal* arc influenced by other languages, especially the programming language Mesa [3].

The first part of this report introduces the language. The latter parts define the language features and give examples. Pascal★ is an upward compatible extension of standard Pascal, but it is syntactically distinct. A syntactically separate extension has two significant advantages: standard Pascal programs will execute as standard Pascal* programs, and Pascal* programs will not be parsed by standard Pascal compilers. Thus standard Pascal compilers will generate syntax errors for Pascal* programs which employ portions of the extension.

There are seven classes of extensions:

1. Type structure – parametric types, type constructors, random access files, explicit packing and allocation size control, and extensions to the notation used in variant records.

2. Constants, variables, and expressions – constant expressions, variable initialization, conditional boolean operators, and a type recasting function.

3. Control structures – set quantification statement, a loop exit statement, and a return statement.

4. Modules – a module construct, with identifier importing and exporting.

5. Exception handling – exception signalling and handling.

6. Procedures and functions – generalization of the function return type and a constant parameter type.

7. Predefincd objects – extended dynamic storage management, a string type and string operations, set functions, prcdefined exceptions, and additional implementation defined constants.

This report uses the cxtcndcd BNF notation used in the Pascal standard [2]. The existing Pascal BNF is not repeated but is employed in the syntax definition of Pascal*. The semantics are defined informally.

1.2 General Extension

A minor but general extension to the language is to relax the order of declaration rules for labels, constants, types, variables, procedures and functions. These declaration sections may occur in any order and any number of titnes. This rule permits structured constants, and grouping of declarations. An identifier must still be declared before it is used. Pointer types arc excluded from the declaration before USC rule, and procedures can be predeclarcd with a forward declaration. The new syntax for the declaration section is:

block = declarations statcmcnt-part .

declarations = {declaration} .

```
declaration = constant-definition-part
            | type-definition-part
            | variable-declaration-part
            | procedure-declaration
            | exception-declaration
            | handler-declaration
            | function-declaration
            | label-declaration-part .
```

2. Types

2.1 Parametric Types

```
type-definition = identifier [ type-parameter-list ] "=" type .

type = parametric-type
     | simple-type
     | structured-type .

parametric-type = identifier type-parameter-list
                | type .

type-parameter-list = "(" type-parameter { "," type-parameter } ")" .

type-parameter = identifier { "," identifier } ":" simple-type .

simple-type = scalar-type
            | subrange-type
            | type-identifier .

subrange-type = id-or-constant ".." id-or-constant .

id-or-constant = identifcr
               | constant .
```

A parametric type is not a conventional type; it is a template for a type. The type descriptor in a paramtric type may contain identifiers which would need to bc constants in a standard Pascal type definition. In a parametric type definition these identifiers are formal parameters. The parameters arc local to the type dcfintion and arc bound to the instances of the identifiers occuring in the type specifier. Instances of new types can bc created by using a parametric type name with a set of actual parameters. Parametric types provide a flexible solution to the problems of array handling which arise in Pascal [2,4].

2.1.1 Parametric type definition

A parametric type dcfinition consists of a type name, the formal parameters for the type with their type-name, and a type specifier which employs the parameter names in place of constants. Altcrnativcly, a parametric type T_1 may bc defined in terms of another parametric type T_2, where the **actual** parameters to T_2 consist of constants and the formal paramctcrs of type T_1. A parametric type definition must specify formal parameters for all parametric components of the type. For example, the following defines two parametric types T and Tarray:

```
type    T(a,b: integer) = a..b;
        Tarray(a,b: char; c,d: integer) = array [a..b] of T(c,d);
```

The type, Tarray, must specify paramctcrs for the array component, T, since it is parametric.

2. I.2 Parametric type instantiation

Since parametric types are templates, they are incompletely specified and denote a class of types. Conventional types (called concrete) can be formed by supplying constants as actual parameters to a parametric type. The resulting concrete type has all the properties of a Pascal type. For example:

type Concrete: Tarray ('A', 'H', 1, 20);

defines an array type whose components are type 1..20 , and which has index type 'A'..'H'. The type Concrete is a well-defined type and may be used in the usual manner.

2.1.3 Parameters and pointers

Pascal ★ allows variable parameters, constant parameters, and pointers to be declared as parametric types. The binding to a particular concrete type is delayed until runtimc, only the parametric type is known at compile-time.

When var and readonly parameters to procedures are defined as parametric types, the corresponding actual parameter must be a concrete instance of the parametric type. The binding to the specific concrete instance occurs at procedure call time.

A pointer type which is parametric is bound to a concrete type either by supplying values in calls to the procedure new, or by assigning to another pointer which has an associated concrete type. Thus every non-nil pointer has a concrete type. Since the paramctcrs for a parametric pointer type can be specified in a call to *new*, it is possible to create dynamic objects in the heap. Derefencing a non-nil parametric pointer yields a parametric type whose parameters come from the pointer type.

A pointer of type ↑T where T is parametric may bc assigned to any other instance of the same parametric type and to a concrete instance of the type whose parameters match. A parametric variable or constant parameter may only be assigned to a instance of the parametric type, with the same parameter values.

2. I. 2 Type Parameter Values

The values of the actual parameters for any instance of a parametric type arc accessed using record selection notation. The parameters arc treated as constants. For example:

```
type  S(a,b: T) =    record
        f1: T1 (a,b);
        f2: T2
end;
var     p :  ↑S;

procedure Q(var x: S);
begin
    { p↑.a and p↑.b are actual parameters for p↑;
        x.a and x.b are actual paramctcrs for x. }
end;
```

2.1.3 Example – Matrix Handling

{Example – matrix multiply.

The following procedure will multiply two matrices of arbitrary size. A work matrix of the resultant size is created in the heap. This allows the procedure to accomodatc the same actual parameter passed to both formal paramctcrs.)

```
const   maxsize = 25;
type    size: 1 ..maxsize;
        matrix (m,n: size) =  array [1..m,1..n] of real;
var     a: matrix (10,5);
        b: matrix (5,10);

procedure MatrixMultiply (var a: matrix; rcadonly b: matrix);
{Multiply a*b and store into a, using a scratch matrix in the heap)
type matrixptr  = ↑matrix;
var   i, j, k: 1..maxsize;
(Need a work matrix pointer -- we could declare a work matrix using: Matrix (maxsize,maxsize)}
        c: matrixp tr;
begin
    new (c (a.m,b.n));
    if (a.n <> b.m)   or (a.m <> b.n) then error ('matrix incompatability')
    else   for i := 1 to a.tn do for j : = 1 to b.n do begin
        c↑ [i,j] : = 0.0;
        for k : = 1 to a.n do c↑ [i,j] : = c↑ [i,j] + a[i,k] * b [k,j]
    end;
    for i : = 1 to a.m do for j : = 1 to b.n do a[i,j] : = c↑ [i,j];
    dispose (c);
end { matrixmultiply };

begin
    MatrixMultiply (a,b)
end.
```

2.2 Type Compatibility

The rule for type compatibility is the rule used in the Pascal standard with appropriate extensions for parametric types. Two paramteric types arc compatible if either they are compatible in the standard Pascal sense, or their types names arc the same and the paramctcrs are equal. Additionally, two parametric pointer types are assignment compatible if' they have the same type name (even if their paramctcrs differ). Because of the strict rules for type checking, problems may bc cncountcrcd in multi-module programs where two modules provide different names for structurally equivalent objects. This type of problem can bc solved with simple applications of type recasting.

2.3 Type Constructors

```
type-constructor  = type-identifier "[" type-specifier "]" .

type-identifier  =  identifier  .

type-specifier  =  expression
                 |   range-specifier ":" expression
                 | type-specifier { "," type-specifier }
                 | [type-identifier] "[" type-spccificr "]"
                 | [record-identifier] "[" record-specifier "]" .

record-specifier  =  field-identifier ":" type-specifier {"," field-identifier ":" type-specifier) .

range-specifier  =  expression ".." expression .

entire-variable  =  variable-identifier
                  | type-constructor .
```

A type constructor permits the construction of a value of any defined concrete type. Consider a type constructor of the form *T* "[" type-specifier "]". *T* must bc a concrete type, i.e. it can not be parametric. The type *T* dictates the form of the type specifier and the result of the type constructor according to the following rules:

1. *T* is a simple type or set type, then the type specifier must be an expression of type *T*. The result is a value of type *T* equal to the value of the type specifier expression.

2. *T* is a array type, then the type constructor must be either a list of expressions with length equal to the size of the index type, or a list of range specifiers. The result is a value of the array type *T*, with the components being the values in the type specifier. If the t, pe specifier is a list of values they are matched to the array components by matching increasing indices with increasing positions in the list of values. If the type specifier is a list of range specifiers, then each range range spccificr has the syntax *range : expression* and means that all elements of the array in the *range* should obtain the value of *expression*. The range values in the list of range specifiers may not overlap. In either case the types of the expressions must match the component type of the array.

3. *T* is a record type, then the type specifier must be a list of field-identifier – value pairs. The field identifiers must match the field names of the record and the type specifiers must correspond to the associated field types. No field-identifier may occur more than once. The result is a value of type T, with the record fields equal to the corresponding values from the type spccificr list.

The type name which normally precedes the type spccificr is optional for inner components of a type constructor. Type constructors may appear anywhere a variable of the constructed type could appear, except that they may not bc assigned to, nor passed as variable parameters. Furthermore, if all the expressions in a type specifier arc compile-time constants, then the result is a constant, and may appear anywhere a constant might appear. In the type constructor for a string or a packed array of char the abbreviation 'string of characters', may be used for a list of character constants; the number of elements in the list must match the length of the string of characters.

2.3. 1 Examples

```
type    T = array [1..3] of integer;
        R = record
            field 1: integer;
            ficld2: 1'
        end;
        s = set of 1..20;
        Matrix = array [1..2,1..2] of real;
const   ZeroMatrix = Matrix[[ 1..2: 0],[0,0]];
        OneTwoThree = T[1,2,3];
        DiagMatrix = Matrix[[1,0],[1,0]];
var     x: T;
        i, j: integer;
        sset: S;
        rrccord: R ;

begin
    i :=4;      j  :=5;
    sset : = S [8..10,i..j] ;
    rrecord : = R[ field1:i, field2: [i,10,25]] ;
    x : = T[4,5,j]
end.
```

2.5 Files and Random Access

Random access is available to all files, using the type *seektype* and the procedure seek:

```
type  seektype = (fromstart, fromcnd, relative);
procedure seek (var f: file; from: seektypc; var n: O..maxint)
```

f -- specifies the file to access

from -- specifies one of three seek types:

fromstart -- seeks from the start of file (n > 0);

fromend -- seeks from the end of the file (n < 0);

relative -- seeks from the current position.

n -- number of records to seek in the file. The actual offset of the record from the start of the file at which the seek stops is returned, in n.

The file operation overput is defined as:

procedure overput (f: file) - writes the value of the buffer variable, f↑, in the current file component. If eof(f) is true prior to execution then the value of f↑ is appended to the file.

The file procedure overwrite causes the current component to be replaced.

```
procedure overwrite ( var f:  file; x: T) is equivalent to:
        f↑ := x;       ovcrput (f)
```

2.6 Machine Dependent Structures

Specification of the packed attribute for type T, has the effect of creating a new type whose storage is minimized with respect to T. Only the structured types array and record are affected by packing; the standard procedures pack and unpack provide conversion between identical packed and unpacked types (as in the standard).

The exact effect of packing on a structure is only partially defined to permit a reasonable level of implementation flexibility. Packing will in general minimize storage; the exact layout of a packed object is not defined, except in the case of record fields. In a packed record each field of an unstructured type is allocated the smallest size possible in bits; however, no field of a simple type is required to cross a word boundary.

2.6.1 Size Control in Packed Records and Arrays

```
field = id-list ":" type [ size-spcc ] .
array-type = "array" "[" type "]" "of" type [ size-spec ] .
size-spcc = "allocate" "(" constant ")" .
```

Within a packed record or packed array, a size attribute may be specified. In a packed record the semantics of the size attribute for a field is to allocate that field as if it occupied the number of bits specified in the size attribute. In a packed array the semantics of the size attribute is to allocate each component in the array as if it occupied the number of bits specified in the size attribute. The number of bits required for the field or component must be less than or equal to the specified value. The field or component is always right justified; if the field or individual array components have a nonstructured type then they are not allocated across word boundaries.

The following example illustrates the use of the size attribute (assume 32 bit words):

```
type    demo = packed record
                   f1: 0..3 allocate (4);
                   f2: 0..1 allocate (25);
                   f-3: 0..20
               end;
        isarray = packed array [1 ..lOO] of 0..1000 allocate (14);
```

The record would occupy 2 words with the following storage layout:

field	word	bit positions
f1	1	0..3
f2	1	4..28
f3	2	0..4

The array will occupy 50 words with 2 components/word, in bit positions 0..13 and 14..27, respectively; bits 28..31 arc empty.

2.7 Variant Records

```
case-constant = constant
              | constant ".." constant .
```

The labels on variant records may also contain constant subranges.

3. Constants, Variables and Expressions

3.1 Extended Constants

```
constant = expression .
```

The definition of' a constant is extended to include constants defined by any compile-time computable expression. An extended constant may bc used anywhere a constant is normally required. A compile-time computable expression is an expression containing only the following: manifest constants, defined constants, constant type constructors, the standard Pascal operators, array indexing, record selection from the fixed portion of a record, and standard prcdcfined functions.

3.2 Variable Initialization

```
variable-declaration = id list ": " type [ : =  variable init ] .
variable-init = constant .
```

Variable initialization is only allowed to variables at the global level of a module or a program. All variables in the identifier list arc initialized to the same value.

3.3 Boolean Expressions

multiplying-operator = "/" | "div" | "mod" | "and" | "cand" .

adding-operator = " +" | "-" | "or" | "cor" .

Two additional operators are defined for boolean expressions. A **cand** B is equivalent to: if A then B else false. A **cor** B is equivalent to: if A then true else B. Note that the order of evaluation for **cand** and **cor** is strictly defined.

3.4 Type Recasting Function

The type recasting function will convert the type of an expression to another type with the same representation length. This function provides explicit type coercion which is implemented as a compile time (but not constant) operation. The function takes two parameters, one of which is a type name and the other the value to be coerced:

function *Recast* (value : Srctype; T: TypeName): T

To convert x to type 'I', the following call is made: *Recast* (x,T). For example, to convert an integer i to a real, the function invocation is: *Recast* (i,real), yielding a real result.

4. Control Structures

4.1 A set oriented for statement

for-statement = "for" control-variable (":=" for-list | "in" set-expression) "do" statement .

This for statement quantifies over a set expression while assigning successive values from the set, in the order smallest to largest, to the control variable.

4.2 Exit Statement

statement = exit .

The exit statement causes termination of the most local statically surrounding for, while, or repeat statement.

4.3 Extended Case Statement

case-statement = "case" expression "of" case-parts "end" [otherwise-part] .

case-parts = case-part { ";" case-part } .

case-part = case-label { "," case-label } ":" statement .

otherwise-part = "otherwise" statement .

case-label = constant [".." constant] .

Case labeling is extended to allow: constant subranges to act as labels. Also, the "otherwise" clause, which is executed if the case selector expression fails to match any case label, is added. Case labels must not overlap.

4.4 Return Statement

return statement = "return" .

The return statement causes immediate termination of the current procedure and a return to the calling procedure.

5. Modules

module = "module" module-identifier [program-paramctcrs] ";" declarations "." .

module-idcntificr = identifier .

Modules provide encapsulation of procedures, functions, exceptions, constants, types, and variables. Modules may also be separately compiled, and intcrmodule type compatability will insured by checking at compile time. Unless an identifier is explicitly exported from a module, it is local to that module and can not be used from other modules. Likewise, all identifiers referenced in a module must be either local or explicitly imported from another module. When an identifier, *x*, is imported from a module *M*, its local name is *M.x*. A with-statement can be used, as it is with a record, to qualify an identifier reference. The statement with *M* **do** S, will cause the identifier *x* to be interpreted as *M.x* within the statement S.

A program is a special instance of a module, declared as in standard Pascal. Only the program may contain a main body, and every executable group of modules must contain exactly one instance of a program.

5.1 Importing and Exporting of Names

declaration = "import" id-list "from" module-name
| "export" id-list ["to" module-list] .

module-list = id-list .

Named objects (i.e. constants, variables, types, proccdurcs, functions, and cxccptions) which are cxportcd or defined by other modules can bc accessed only if they arc named in an import statement. The export statcmcnt makes a list of objects available for importation by other modules. To reference an object declared by another module, it must bc cxportcd by the declaring module, called the implementor, and imported by the accessing module, called the client. If the export destination list is present in the implementor it must include the name of the client module, and the client module must always name the implementor. Omitting the module destination list on an export statcmcnt allows all potential clients access to the identifiers being exported. The compiler creates the necessary information for intcrmodulc type checking, which is done at compile-time. Modules must bc compiled according to the partial ordering derived from the usc of identifiers which arc imported and cxportcd.

5.2 Creating Module Interfaces

declaration = "defines" id-list ["for" module-list] .
| "implements" id-list "for" module-list .

It is often useful to separately define the interface between a client and an implcmcntor. There arc several potential benefits including the ability to work on both client and implcmcntor in parallel, and employing multiple implementors without requiring recompilation of the client. The

defines and **implements** constructs provide a way of specifying module interfaces separately from the implementing module.

Objects in the client/implementor interface are declared in an interface module, and listed in the defines clause within that module. An implementor may specify that it implements some subset of the objects defined in the interface module by naming the objects in an implements clause. The named objects will then reside in the implementor but continue to be defined by the interface module. The interface module need only be compiled once, then the implementor and client modules may be compiled independently.

The *defines* construct behaves like an export in that it makes a name accessible from both client and implementor modules. Client modules must import the object names which they require from the module which defines them. The for clause on a defines statement specifies the legal client modules just as a destination clause does on an export directive. Defines may be used to specify a procedure type (i.e. its parameters and return type if it is a function). In this case the procedure header is specified in the interface module, with the keyword definition replacing the body.

The *implements* statement is used to specify that a certain module implements a portion of an interface defined in an interface module. Implements specifies a list of' identifiers which are defined in other modules but which reside in the module with the implements construct. There can be at most one implementor for each defined object in a program. An object which is implemented in a module is not redeclared in that module; however, a procedure or function body must be elaborated in the implementor. Such a procedure or function is declared as **usual** except the parameters are omitted.

5.3 Restricted phrase

declaration = "defines" "restricted" id-list ["for" module-list] .

An exporting, defining, or implementing statement may restrict access to variables and types which it defines, through the use of the restricted construct. The construct **exports**, **defines**, or **implements** followed by **restricted** id-list, will result in a set of objects whose use by modules which import them is restricted. Restricted affects the use of both types and variables. For variables no alteration of the value is permitted. For types only assignment and test for equality are permitted. The implementor retains full access rights. If an identifier is restricted by an interface module it must also be restricted by the corresponding implementation module.

5.4 Example

A module which implements some matrix operations is defined with an interface.

```
module MatrixInterface;

    {The interface to the module)
    defines restricted matrix;
    imports vector from   VectorInterface;
    exports vector, size;
    defines   Add, GetRow;
    type    size  =  1..50;
            Matrix (m,n: size) = array [1..m] of vector (n);
    procedure Add (var a: matrix; readonly b: matrix); definition;
    procedure GetRow (var v:vector; readonly a:matrix; rownumber:size); definition;

end; { MatrixInterface }
```

```
module Matrix Module;
    (Only two procedures arc included in thc module, obviously other procedures, e.g. multiply
    could also be included.}
    implements Add, GetRow for MatrixInterface;
    imports size from Matrix Interface;
    imports Add from VectorInterface;
    procedure Add;
    { assume no aliasing and compatible sizes }
    begin
        for i := 1 to a.m do VectorInterface.Add (a[i],b[i]);
    end { Add };
    procedure GetRow;
    begin
        v : = a[rownumber];
    end { Dimensions };
end { MatrixModule }.
```

Then a separate UserModule could employ the the matrix routines via the interface:

```
module UserModule;
    imports Add, GetRow, Size, Matrix from MatrixInterface;
    imports Vector from VectorInterface;
    { Create two matrices }
    var     M, N: Matrix( 10,10);
            i, j: Size;
            v: Vector;
    procedure UseMatrix;
    begin
        MatrixInterface.Add (M,N);
        Matrix Interface.GetRow (v,M,i)
    end
end { UserModule } .
```

6. Exception Handling

```
exception-declaration = exception-heading block .
exception-heading = "exception" "identifier" [ formal-parm-sections ] ";" .
formal-parm-sections = "(" formal-parm-section { ";" formal-parm-section) ")" .
statement =   "raise" "idcntificr" [ actual-parameter-list ]
            | "catch" cxccption-handler {";" exception-handler} "in" statement
            | "continue".
exception handler = exception-identifier {"," exception-identifier) ":" statement .
exception-identifier = "identifier" .
```

6.1 Exception Declarations

An exception declaration specifies an exception which can be raised and caught by a handler. The exception declaration specifies the name of the exception and its formal parameters. An cxccption must be declared (or imported) before it can bc caught or raised.

6.2 Exception Handlers

One or more exception handlers can be bound to a statement using the catch statement. The catch statement supplies a list of exceptions (and handlers) which are to be used if the named exception is raised within the statement. For example: catch E_1: S_1;...E_n: S_n in S, specifies that. the exceptions *El,...,En* are caught locally if the statement S terminates by raising any one of these exceptions. After catching the cxccption E_i, statcmcnt S_i is executed.

Scoping and access to identifiers in an exception handler is identical to that of the associated statcmcnt S, except that the exception parameters arc also visible. A handler is very much like a procedure; it can reference the parameters declared by the exception declaration, contain procedure calls and variable references, but is invoked in a special manner.

Additionally, the exception name *others* can be used to define an exception handler which fields any exception, except the special exception *unwind.* It has a single rcadonly parameter, which is type *string,* and will contain the name of the exccption which was actually raised.

6.3 Raising an Exception

An exception is raised by the execution of a raise statement, of the form raise exception-name "(" paramctcr-list")". Raising an exception causes one of two actions: either a program error or the invocation of an exception handler, with the name of the raised exception or the name *others.*

The invocation of a handler for a raised exception is dynamic according to the following rules:

1. If the exception E is raised in procedure P within statcmcnt S, then the most local exception handler for E (or *others)* which is associated with a statement which includes *E,* is invoked.

2. If no statement in I' which contains S has handler for E or *others,* then the search continues with the statement which invoked P.

3. If the search reaches the main block and no handler appears there, a runtime error is caused.

4. If a handler raises an exception the search begins with the statement which contains the catch statement.

There arc two types of exceptions *standard* and *user declared* Standard exceptions - including Overflow, DividebyZero, etc. can be raised by both programmer and translated program, standard parameters are dcfincd. It is not possible to continue after handling a predefined exception. User declared exceptions are raised explicitly by user program.

6.4 Terminating a Handler

The handler body can be thought of as an inline procedure, so that its execution interrupts the execution of the statement raising the exception. Thus, there are several ways in which the exception handler can complete execution:

normal completion - reaching the end of the handler body. This causes the statement to complete execution and control begins on the next statement.

goto - has the same effect as if executed from within the object of the catch statement.

return - effects a normal procedure return from the procedure enclosing the handler.

raise - raise another exception, accepts a parameter list. The search for the handler begins on the surronding statement.

continue - causes execution to continue following the orginal raise statement.

If a series of procedures is about to be aborted (because a handler is terminating without using continue), then the exception *unwind* which has no parameters is raised at the point of the orginal exception raise. *Unwind* is never propogated past the handler which caused its invocation. If an unwind handler does not terminate with the statement raise *unwind*, the propogation is halted and control continues in the manner indicated by the unwind handler.

7. Procedures and Functions

One extension is made to the procedure and function parameter mechanism, and one extension to the function mechanism.

7.1 Constant Parameters

```
formal-parameter-section = parameter-group
                         | "var" parameter-group
                         | "const" parameter-group
                         | procedure-heading
                         | function-heading .
```

A new parameter type const is introduced. Parameters of type const may only be referenced; they may not be assigned to nor passed as variable parameters.

7.2 Function Types

Functions are extended to allow them to return any type except parametric types and files.

8. Predefined Objects

8.1 Dynamic Storage Management

New allocates storage from the dynamic storage pool (heap) and sets a pointer to the allocated storage. There are two additional types of calls to new. Assume p is type ↑T, then

1. new(p(v_1,...v_n)) - if p is a pointer type described by p = ↑T(a_1,...,a_n) then this call to new creates an instance of type T(v_1,...,v_n) and returns it as p↑; until p is reassigned the type of p↑ is T(v_1,...,v_n).

2. new(p(v_1,..., v_n), t_1,...,t_m) - applies when the referent type of p is both parametric and a variant record. The vi are the parameters and the ti are the tags.

Dispose is an explicit mechanism to free storage which will make available for reallocation via subsequent calls to new. Assume p is type ↑T, then dispose (p) has two effects: p↑ is made available for allocation, and p is assigned the value nil. The operation dispose (p) has no effect on any other objects or pointers unless the object is p↑ or the pointer is equal to p. If p_1 = p and the operation dispose(p) is done, it is illegal to deference p_1 before assigning it a new value.

8.2 Set Functions

If S is type set of i..j, then the following functions are defined on S:

first (S) = the smallest k | k in S.
last (S) = the largest k | k in S.

8.3 Scalar Functions

ord - if x is an expression of scalar type T, ord (x) returns the ordinal position of x in the type. For example if T = (i_0,...,i_n), then ord (i_j) = j.

8.4 Type Functions

size - size (type name) yields the allocated size in storage units of the named type. If the type is parameterized the parameters must appear following the type name, e.g. size (type name (pl,...,pn)).

8.5 Standard Exceptions

A set of standard exceptions is defined in the module *DefinedExceptions*. To raise or catch a predefined exception, the exception name must be imported from the module into the local module. The predefined exceptions can not be continued. When a predefined exception is raised and a set of activations is terminated, the portion of the code executed in any activation need not be consistent with the order of the code and the position of the statement which causes the raise.

The exception *UndefinedException* is raised when a predefined exception whose name was not imported occurs; it can not be caught and is always reported as an error in the program (the variable values when this exception arises may not be well defined).

The following definitions specify the standard exception headers (handlers for these exceptions may not terminate with continue), the existence and type of parameters for the standard exceptions is implementation dcpcndent:

```
exception    IntegerOverflow.

exception    RealOverflow.

exception    RealUnderflow.

exception    ZeroDivide;   {By an integer}.

exception    ReferenceError; {Attempt to deference a null valued pointer}.

exception    IndexError; (Array index not a member of the array index type}.

exception    RangeError;   (Attempt to exceed the range of a scalar variable}.

exception    StorageOverflow;   {Available storage has been exceeded).

exception    ContinuationError;   (Attempt to continue for a prcdcfined exception}.

exception    InputError;   (Input data does not match the date type being read).
```

8.6 Implementation Defined Constants

minint - minimum integer	maxint - maximum integer
minreal - minimum real	max real - maximum real
minchar - minimum character	maxchar - maximum character
charsctsizc - size of character set	maxsetsizc - maximum set size

8.7 Implementation Requirements

These requirements may bc used in the creation of portable programs.

$|minint| \quad maxint > 60 > 2^{15} - 1$

$charsetsize > 60$

$|minreal| \quad , \quad max\ real > 2^{21} - 1$

8.8 String Type

A string facility which provides varying length strings with a maximum size on each string is included. The type *string* is a parametric type: the parameter specifics the upper bound on the length. The logical view on a string is defined by:

```
type    string (n: 0..maxint) = record
            length: O..n;
            text: packed array [1..n] of char
        end
```

Pascal string constants of the form 'characters' are treated as string constants where the actual parameter is the length of the string. Thus the constant 'AK', has type string(3), and is considered an abbreviation for: string(3) [3, ['A', 'B', 'C']]. The predcfincd constant *nullstring* is defined as a string with a zero length field.

The type string is supported by a variety of predcfined procedures for manipulating strings. Since strings arc standard variables they may be compared for equality and assigned, but these operations may be less efficient than using string procedures. The string procedures are briefly defined below in psuedo-Pascal* :

```
procedure ASSIGN(readonly source: string; var dest: string);
begin
    dest : = source; { only the characters in positions 1..length in source are copied }
end;

function LENGTH(readonly source:string): O..maxint;
begin LENGTH: = source.length end;

function POS(readonly string1 ,string2: string): 0..maxint;
begin if ∃i (string1.text = string2.text [i..i+string1.length-1] ) then POS: = i else POS: =0
end;

procedure SUBSTR( readonly source:string; sourcepos:1..maxint; leng:0..maxint;
    var dest: string; destpos: 1..maxint);
begin dest.text[destpos..destpos+leng-1] : = source[sourcepos..sourcepos+leng-1];
      dest.length: = max(dest.length,destpos + leng-1) end;

procedure APPEND(readonly source: string; var dest: string);
begin     dest.text[dest.length + 1..dest.length + source.length]:=source.text end;

function GETCHAR(readonly source:string; sourcepos:1 ..maxint): char;
begin  GETCHAR: = source.tex t[sourcepos] end;

procedure PUTCHAR(source:char; var dest:string; destpos:1..maxint);
begin     dest.text[destpos]: = source end
```

Additionally, the following comparison functions are provided: S'I'REQ, STRN E, STRGT, STRGE, STRLT, STRLE. Each accepts two strings and returns a boolean result, They are defined as follows:

```
function STREQ(readonly S1,S2: string): boolean;
begin  STREQ : = (S1 .length = S2.length) and (S1.text[1..S1.length] = S2.text[1..S1.length]) end;

function STRNE(readonly S1,S2: string): boolean;
begin     STR N E : = not STREQ(S1,S2) end;

function STRGT( readonly S1,S2: string): boolean;
begin  if (S1.length = S2.length) and (S1.length = 0) then false
       else if S1.length = 0 then STRGT : = false
       else if S2.length = 0 then STRGT : = true
       else STRGT : = (GETCHAR(S1,1) > GETCHAR(S2,1)) o r
            ((GETCHAR(S1,1) = GETCHAR(S2,1)) and
             STRGT (SUBSTR(S1,2,S1.leng-1),SUBSTR(S2,2,S2.leng-1))) end;
```

```
function STRGE(readonly S1,S2: string): boolean;
begin    STRGE := STREQ(S1,S2) or STRGT(S1,S2) end;

function STRLT(readonly S1,S2: string): boolean;
begin    STRLT := not STRGE(S1,S2) end;

function STRLE(readonly S1,S2: string): boolean;
begin    STRLE := not STRGT(S1,S2) end
```

The standard procedures Read and Write will also accommodate strings.

Acknowledgments

Forest Raskett provided the initial motivation for Pascal★ and supplied numerous valuable criticisms and suggestions during the design. Many other people contributed comments and criticisms especially the S-1 staff members here and at Lawrence Livermore Laboratories, and faculty and graduate students at Stanford.

References

[1] Jensen, K. and Wirth, *N., Pascal User Manual and Report,* Springer Verlag, New York, 1974.

[2] Addyman, A., *ISO Draft Pascal Standard,* SIGPLAN Notices, June 1980.

[3] Michell, J.G., Maybury, W., and Sweet, R., *Mesa Language Manual Version 5.0,* Tech. Rep. CSL-79-3, Xerox PARC, Palo Alto, Ca., 1979.

[4] Habermann, A.N., *Critical Comments on the Programming Language Pascal,* Acta Informatica 3, 47-57, 1973.