PS-algol Reference Manual
Fourth Edition

Ron Morrison

Department of Computational Science,
University of St Andrews,
North Haugh,
St Andrews
Fife
Scotland.
KY16 9SS.

Persistent Programming Research Report 12
Reprinted 22/2/88.

Contents

Chapter

1. Introduction

2. Syntax Specification

3. Types and Type Rules

4. Literals
    - 4.1 Integer Literals
    - 4.2 Real Literals
    - 4.3 Boolean literals
    - 4.4 String Literals
    - 4.5 Pixel Literals
    - 4.6 Pointer Literal
    - 4.7 File literal
    - 4.8 Procedure Literals

5. Primitive Expressions and Operators
    - 5.1 Boolean Expressions
    - 5.2 Comparison Operators
    - 5.3 Arithmetic Expressions
    - 5.4 Arithmetic Precedence Rules
    - 5.5 String Expressions
    - 5.6 Picture Expressions
    - 5.7 Pixel Expressions
    - 5.8 Precedence Table
    - 5.9 Other Expressions

6. Identifiers and Object Declarations
    - 6.1 Variables, Constants and Declaration of Data Objects
    - 6.2 Sequences
    - 6.3 Brackets
    - 6.4 Scope Rules

7. Compound Data Objects
    - 7.1 Vectors
        - 7.1.1 **upb** and **lwb**
    - 7.2 Structures
        - 7.2.1 **is** and **isnt**
        - 7.2.2 Equivalence of Structure Classes
    - 7.3 Indexing
    - 7.4 Image Expressions

7.5 Assignment and Equality of Pointers

8. Procedures
8.1 Recursive Declarations
8.2 Equality of Procedure

9. Clauses
- 9.1 Assignment Clause
- 9.2 **if** Clause
- 9.3 **case** Clause
- 9.4 **repeat** ... **while** ... **do** ... Clause
- 9.5 **for** Clause
- 9.6 **abort** Clause
- 9.7 Exceptions
- 9.8 System Events and System Exceptions

10. Input and Output
- 10.1 Input
- 10.2 End of Input
- 10.3 Output
- 10.4 i.w, s.w and r.w
- 10.5 Input from and Output to files

11. Persistence
- 11.1 Database Procedures
- 11.2 Tables
- 11.3 Database Conventions

12. Standard Functions and Identifiers
- 12.1 Standard Functions
- 12.2 Standard Identifiers

Appendices

I PS-algol Syntax
II PS-algol Type Rules
III Program Layout
IV PS-algol Prelude
V System Event and System Exception Interpretation
VI Printing Images and Fonts

1. Introduction

This is the 4th edition of the PS-algol reference manual. It describes the same language that the 3rd edition does, the difference being in the way in which the language is defined. We have returned to the context-free syntax and the type rule definitions of S-algol and have abandoned the flirtation with two level grammers that no-one seemed to find helpful. This version of the manual has been constructed with a lot more care than any of the previous ones and hopefully it now accurately describes the language. Most people on the project have contributed to this by proof reading, re-writing or tidying up their sections of the manual. The main contributers to this have been Ron Morrison, Malcolm Atkinson and Paul Philbrow. John Robinson undertook the most meticulous proof reading of the whole document and suggested numerous improvements to the text and layout. Further proof reading was performed by Richard Connor, John Scott and Ray Carrick. Fred Brown and Al Dearle checked that the language definition and implementation were in harmony (a rare event in research languages). Tony Davie constructed the index.

PS-algol has changed considerable since the merging of the S-algol system with persistence. The early systems were designed and implemented by Malcolm Atkinson, Paul Cockshott, Ken Chisholm, Pete Bailey and Ron Morrison. Since then the major system builders have been Fred Brown and Al Dearle. Paul Philbrow added the exceptions and finished the **print** statement started by John Livingstone. Jack Campin and Fred Brown added database garbage collectors.

PS-algol is available on VAX Unix (Bsd 4.2), SUN workstations, Macintosh, Whitechapel MG/1 (Livingstone), and ICL 2900 (Robinson & Scott). Many other people are also working on implementations which hopefully will grow.

The language is one of the main deliverables of the PISA project and everyone on the project has contributed to the success of PS-algol in some way. The following are the current members of the PISA group.

Glasgow	**St Andrews**	**STC**
Malcolm Atkinson	Ron Morrison	Graham Pratten
Jack Campin	Fred Brown	Nick Capon
Richard Cooper	Raymund Carrick	Michael Guy
Douglas Mc Farlane	Richard Connor	John Robinson
Paul Philbrow	Al Dearle	John Scott
Francis Wai		

Ron Morrison
Malcolm Atkinson
28-6-87

2. Syntax Specification

It is important that a programming language can be formally defined since it gives implementors a standard to work on. There are two levels of definition, syntactic and semantic. Here we will deal with the formal syntactic rules giving an informal semantic description of the syntactic categories. That is we will define the set of all syntactically legal PS-algol programs remembering that the meaning of any one of these programs is defined by the semantics.

To define the syntax of a language we need another notation which we will call a **meta language** and in this case we choose a variation of Backus-Naur form.

The syntax of PS-algol is specified by a set of rules or **productions** as they are normally called. Each production specifies the manner in which a particular syntactic category (e.g. a clause) can be formed. The syntactic category name is enclosed in the meta symbols '<' and '>' thus distinguishing it from names or reserved words in the language. These syntactic categories can be mixed in productions with terminal symbols which are actual symbols of the language itself. Thus, by following the productions until we have only terminal symbols, we can derive legal programs.

Other meta symbols include '|' which allows a choice in a production. The square brackets '[' and ']' are also used in pairs to denote an object is optional. When used with a '*' we have a zero or many times repetition. You should not confuse the meta symbols '|','<', '>', '*', and '[', ']' with the actual symbols in PS-algol and we will be careful to keep the two concepts completely separate in our description.

As you may expect with any reasonably powerful programming language, the productions for PS-algol are recursive which means that there are an infinite number of legal PS-algol programs. However the syntax of PS-algol can be described in about 60 productions. We will break ourselves in gently with an example.

```
<identifier> ::=    <letter>[<letter> | <digit> | .]*
```

indicates that an identifier can be formed as a letter, optionally followed by zero or many letters, digits or dots.

The full context-free syntax of PS-algol is given in Appendix I.

The order of execution of a PS-algol program is strictly from left to right and top to bottom. This rule becomes important in understanding side-effects in the store.

3. Types and Type Rules

There are an infinite number of data types in PS-algol defined recursively by the following rules.

a. The primitive data types are integer, real, boolean, picture, pixel, file and string.

b. #pixel is the type of an image made up of pixels arranged as a rectangular matrix.

c. For any data type T, *T is the data type of a vector with elements of type T.

d. The data type pointer comprises a structure with any number of fields, and each field consists of a binding of a name and a value of any type.

e. For any data types T_1,...T_n and T, **proc**(T_1,...T_n->T) is the data type of a procedure taking parameters of type T_1 to T_n and producing a result of type T. The type of a similar resultless procedure is **proc**(T_1,...T_n)

In addition to the above data types there are a number of other objects in PS-algol to which it is convenient to give a type in order that the compiler may check their use for consistency.

f. Clauses which yield no value are of type void.

g. The class of a structure with fields of type T_1,...T_n is of type (T_1,...T_n)-structure and its fields are of type T_i-field.

The world of data objects is defined by the closure of rules a and b under the recursive application of rules c, d and e.

The generic types that are require for the formal definition of PS-algol can be described by the following rules.

```
type arith       is   int | real
type comparable  is   arith | string
type printable   is   comparable | bool
type literal     is   printable | pntr | proc | file | pixel
type image       is   #pixel | #cpixel
type nonvoid     is   literal | image | *nonvoid
```

typetype   **is**   nonvoid | void

For example, in the above rule the generic arith can be either an int or a real, representing the types integer and real in the language. In the type rules, the language types and generic types are written in shadow font to distinguish them from the reserved words in the language.

To check that a syntactic category is correctly typed the context free syntax is used in conjunction with a type rule. For example, the type rules for the two-armed **if** clause is t : type, **if** <clause> : bool **then** <clause> : t **else** <clause> : t => t

This rule may be interpreted as follows. t is given as a type from the table above. That is, it can be any type including void. Following the comma, the type rule states that the reserved word **if** may be followed by a clause which must be of type boolean. This is indicated by : bool. The **then** and **else** alternatives must have clause of type t, whichever type it is. That is, both alternatives must have the same type. The resultant type, indicated by =>, of this production is also t, the same as the alternatives.

The type rules will be used throughout this manual, in conjunction with the context-free syntax rules to describe the language. A complete set of type rules for PS-algol is given in Appendix II.

4. Literals

Literals are one of the basic building blocks of a program and allow values to be introduced. A literal is defined by

```
<literal>	::=	<int_literal> | <real_literal> | <bool_literal> |
<string_literal> |
		<pixel_literal> | <pntr_literal> | <file_literal> |
<proc_literal>
```

4.1 Integer Literals

These values of type int are defined by

```
<int_literal> ::=	<digit>[<int_literal>]

<int_literal> => int
```

An integer literal is one or more digits.

e.g. 1   0   1256 8797

Notice that negative literals are not allowed. However negative integers can be written down as expressions.

4.2 Real Literals

These are of type real and are defined by

```
<real_literal> ::=  <int_literal>.[<int_literal>][e<scale_factor>]
<scale_factor> ::= [<add_op>]<int_literal>
<add_op> ::=		+ | -

<real_literal> => real
```

Thus, there are a number of ways of writing a real literal. For example

```
1.2	3.1e2	5.e5
1.	3.4e-2	3.4e+4
```

3.1e-2 means 3.1 times 10 to the power -2 (i.e. 0.031)

4.3 Boolean Literals

There are only two literals of type bool. They are the symbols **true** and **false**. They may be used with obvious meaning when a boolean literal is required.

<bool_literal> ::= **true | false**

<bool_literal> **=>** `bool`

4.4 String Literals

A string literal is a sequence of characters in the character set (normally ASCII) enclosed by double quotes.

<string_literal>	::=	<double_quote>[<char>]*<double_quote>
<char>	::=	<other> | <special_character>
<special_character>	::=	<single_quote><special_follow>
<special_follow>	::=	n | p | o |t | b | <single_quote> | <double_quote>
<other>	::=	any ascii character except ' and "

<string_literal> **=> string**

The empty string is denoted by "". Examples of string literals are

"This is a string literal"
"I am a string"

The programmer may wish to have a double quote itself inside a string literal. This requires using a single quote as an escape character and so if a single or double quote is required inside a string literal it must be preceded by a single quote. For example

"a'"" has a value a" and
"a''" has a value a'

There are a number of other special characters which may be used inside string literals. They are

'n	newline
'p	newpage
'o	carriage return
't	horizontal tab
'b	backspace

These characters are normally used in input and output where their effect is device dependent.

4.5 Pixel Literals

A pixel literal is defined by

<pixel_literal> ::= **on**[&<pixel_literal>] | **off**[&<pixel_literal>]

<pixel_literal> **=> pixel**

For example

on **off** & **on** & **off**

4.6 Pointer Literal

PS-algol provides a unique pointer literal. It is defined by

<pntr_literal> ::= **nil**

<pntr_literal> **=> pntr**

4.7 File Literal

PS-algol provides a unique file literal. It is defined by

<file_literal> ::= **nullfile**

<file_literal> => **file**

4.8 Procedure Literal

Procedures are introduced into programs by their literal value. They are defined by

<proc_literal> ::= **proc**([<named_param_list>]
[<arrow><type>]);<proc_clause>
<named_param_list>::=<proc_param_type>[;<named_param_list>]
<proc_param_type> ::= <type1><identifier_list> | <structure_decl>
<proc_clause> ::= <clause> | **nullproc**

<type1> ::= [**c**]<type>
<type> ::= **int** | **real** | **bool** | **string** | **pntr** | **file** | **pixel** | **pic** |
<star><type1> | #**pixel** | #**cpixel** | <proc_type>
<proc_type> ::= **proc**([<proc_param_list>][<arrow><type>])
<proc_param_list> ::= <proc_para_type>[,<proc_param_list>]
<proc_para_type> ::= <type1> | <structure_decl>

For example

proc (**cint** n -> **int**) ; n

is a procedure literal. For the use of procedures see Chapter 8.

5. Primitive Expressions and Operators

5.1 Boolean Expressions

Objects of type bool in PS-algol can have the value true or false. There are only two boolean literals, **true** and **false** and three operators. There is one unary operator, ~, and two binary operators, **and** and **or**. They are defined by the truth table below.

a	b	~a	a **or** b	a **and** b	
true	false	false	true	false	
false		true	true	true	false
true	true	false	true	true	
false		false	true	false	false

The precedence of the operators is important and is defined in descending order as

~
and
or

Thus

~a **or** b **and** c

is equivalent to

(~a) **or** (b **and** c)

This is reflected in the syntax rules which are

```
<expression>      ::=   <exp1>[or<exp1>]*
<exp1>            ::=   <exp2>[and<exp2>]*
<exp2>            ::=   [~]<exp3>...
```

<expression> : **bool** **or** <expression> : **bool** **=>** **bool**
<expression> : **bool** **and** <expression> : **bool** **=>** **bool**
[~]<expression> : **bool** **=>** **bool**

The evaluation of a boolean expression in PS-algol is non-strict. That is in the left to right evaluation of the expression, as soon as the result is found, no more computation is performed on the expression. For example

true or <expression>

gives the value **true** without evaluating <expression> and

false and <expression>

gives the value **false** without evaluating <expression>.

5.2 Comparison Operators

Expressions of type bool can also be formed by some other binary operators. For example, a = b is either true or false and is therefore boolean in nature. The operators are called the comparison operators and are

<	less than
≤	less than or equal to (written <=)
>	greater than
≥	greater than or equal to (written >=)
=	equal to
≠	not equal to (written ~=)

The syntactic rules for these are

<exp2> ::= [~]<exp3>[<rel_op><exp3>]
<rel_op> ::= **is** | **isnt** | < | ≤ | > | ≥ | = | ≠

[~]<expression> : **bool => bool**

t : nonvoid, <expression> : **t** <eq_op> <expression> : **t => bool**
where <eq_op> ::= = | ≠

t : comparable,<expression> : **t** <co_op> <expression> : **t => bool**
where <co_op> ::= < | ≤ | > | ≥

<expression> : **pntr** <type_op><identifier> **=> bool**
where <type_op> ::= **is** | **isnt**

Note that the operators <, ≤, > and ≥ are defined on integers, reals and strings whereas = and ≠ are defined on all PS-algol data types. Their interpretation for these data types are given with each data type as it is introduced below. The operators **is** and **isnt** are for testing a structure class.

5.3 Arithmetic Expressions

Arithmetic may be performed on data objects of type **int** and **real**. The syntax of arithmetic expressions is

<exp3> ::= <exp4>[<add_op><exp4>]*

```
<exp4>         ::=    [<add_op>]<exp5>[<mult_op><exp5>]*
<add_op>    ::=    + | -
<mult_op>   ::=    <star> | / | div | rem
```

t : arith, <expression> **: t** <add_op> <expression> **: t => t**

t : arith, <add_op> <expression> **: t => t**

<expression> : **int** <int_mult_op> <expression> : **int => int**
where <int_mult_op> ::= <star> | **div** | **rem**

<expression> : **real** <real_mult_op><expression> : **real => real**
where <real_mult_op> ::= <star> | /

The operators mean

+	addition
-	subtraction
*	multiplication
/	real division
div	integer division throwing away the remainder
rem	remainder after integer division

In both **div** and **rem** the result is negative only if exactly one of the operands is negative.

Some examples of arithmetic expressions are

a + b   3 + 2   1.2 + 0.5   -2 + a / 2.0

The language provides automatic coercions from **int** to **real**, but the transfer may be explicitly invoked by 'float', and the standard function 'truncate' is provided to transfer from **real** to **int** .

5.4 Arithmetic Precedence Rules

The order of evaluation of an expression in PS-algol is from left to right and based on the precedence table

*   /   **div**   **rem**
+   -

That is, the operations *, /, **div**, **rem** are always evaluated before + and -. However, if the operators are of the same precedence then the expression is evaluated left to right. For example

6 **div** 4 **rem** 2   gives the value 1

Brackets may be used to override the precedence of the operator or to clarify an expression. For example

3 * (2 - 1)   yields 3 not 5

5.5 String Expressions

There is only one string operator ++ defined on strings. It concatenates the two operand strings to form a new string. For example

```
"abc" ++ "def"
```

results in the string

"abcdef"

The syntax rule is

```
<exp4>          ::=    <exp5>[++<exp5>]*

<expression> : string ++ <expression> : string => string
```

A new string may be formed by selecting a substring of an existing string. For example

if s is the string "abcdef"
then s(3 | 2) is the string "cd"

That is, a new string is formed by selecting two elements from s starting at element 3. The syntax rule is

```
<exp5>          ::=    <name>[(<clause><bar><clause>)]*

<name> : string (<clause> : int <bar> <clause> : int ) => string
```

For the purposes of substring selection the first character in a string is numbered 1. The selection values are the start position and the length respectively.

The characters in a string are ordered according to the ASCII character code. Thus

"a" < "z"

To compare two strings, the characters are compared in pairs, one from each string, from left to right. Two strings are considered equal only if they have the same characters in the same order and are of the same length. Otherwise they are not equal.

The null string is less than any other string. Thus the less-than relation can be resolved by taking the characters pair by pair in the two strings until one is found to be less than the other. The other relations can be resolved by using '=' and '<'.

5.6 Picture Expressions

The picture drawing facilities of PS-algol allow the user to produce line drawings in two dimensions. The system provides an infinite two dimensional real space. Altering the relationship between different parts of the picture is performed by mathematical transformations which means that pictures are usually built up of a number of sub-pictures. The syntax of picture expressions is

```
<exp4>          ::=   <exp5>[<pic_op><exp5>]*
<pic_op>    ::=   ^ | &

<expression> : pic <pic_op><expression> : pic => pic

<picture_exp>     ::=   shift<clause>by<clause>,<clause> |
                  scale<clause>by<clause>,<clause> |
                  rotate<clause>by<clause> |
                  colour<clause>in<clause> |
text<clause>from<clause>,<clause>to<clause>,<clause> |
                <lsb><clause>,<clause><rsb>

shift<clause> :  pic by<clause> : real,<clause> : real => pic
scale<clause> :  pic by<clause> : real,<clause> : real => pic
rotate<clause>   :   pic by<clause> : real => pic
colour<clause>   :   pic in<clause> : pixel => pic
text<clause>  :  string  from<clause> : real,<clause> : real
                         to<clause> : real,<clause> : real => pic
<lsb><clause>:   real,<clause> : real <rsb> => pic
```

In a line drawing system the simplest picture is a point. For example, the expression

[0.1,2.0]

defines the point 0.1,2.0.

Points in pictures are implicitly ordered. A binary operation on pictures operates between the last point of the first picture and the first point of the second. The resulting picture has as its first point, the first point of the first picture, and as its last, the last point of the second.

There are two infix picture operators. They are '^', which joins one picture to another by a straight line from the last point of the first picture to the first point of the second, and '&' which includes one picture in another. The other transformations and operations on pictures are

shift The new picture consists of the points obtained by adding the x and y shift values and the x and y co-ordinates of the points in the old picture. The ordering of the points is preserved.

scale The new picture consists of the points obtained by multiplying the x and y scale values with the x and y co-ordinates of the points in the old picture, respectively. The ordering of the points is preserved.

rotate The new picture consists of the points obtained by rotating the x and y co-ordinates of the points in the old picture about the origin by the angle indicated. The ordering of the points is preserved.

colour The new picture is the old one in a new colour.

text Form a picture consisting of the text string. The two co-ordinates represent the start and end points of the string which will be scaled to fit.

5.7 Pixel Expressions

Pixels may be concatenated to produce another pixel of a greater depth using the operator '&'.

```
<exp4>          ::=    <exp5>[&<exp5>]*
```

<expression> : **pixel** &<expression> : **pixel => pixel**

A pixel has depth representing the number of planes in the pixel. The planes are numbered from 0 and new pixels can be formed from subpixels of others. The syntax is

```
<exp5>          ::=   <name>[(<clause><bar><clause>)]

<name> : pixel (<clause> : int <bar><clause> : int ) => pixel
```

For example

let b = **on** & **off** & **off** & **on**
b (1 | 2)   is the pixel **off** & **off**

This last expression is interpreted as the pixel formed by starting at plane 1 in 'b' and including all the planes up to plane 2.

5.8 Precedence Table

The full precedence table for PS-algol is now

/	*	**div**	**rem**	^	&		
+	-	++					
~							
=	≠	<	≤	>	≥	**is**	**isnt**
and							
or							

5.9 Other Expressions

All the PS-algol operators have now been described but we have not exhausted all the ways of writing down expressions. There are vector expressions, pointer expressions, vector and structure indexing expressions, procedure expressions, image expressions, **if** and **case** expressions as well as block expressions and the fact that literals, identifiers and procedure calls on their own constitute expressions. All of these are dealt with elsewhere in this manual.

6. Identifiers and Object Declarations

In PS-algol an identifier may be given to a data object, a procedure parameter, a structure field and a structure class. An identifier and may be formed by the syntactic rule

    <identifier> ::=   <letter>[<letter> | <digit> | .]*

That is, an identifier consists of a letter followed by any number of dots, letters or digits. For example

    x1 ron   look.for.record1 Ron

Note that case is significant in identifiers.

6.1 Variables, Constants and Declaration of Data Objects

Before an identifier can be used in PS-algol it must be declared. The action of declaring a n identifier associates it with a location of a certain type which can hold values that the identifier may take. In PS-algol the programmer may specify whether the value is constant or variable. A constant may be manipulated in exactly the same manner as a variable except that it may not be updated.

When introducing an identifier the programmer must indicate the identifier, the type of the data object, whether it is variable or constant and its initial value.

Variables are declared by

let <identifier>:=<clause>

let<identifier>:=<clause> : nonvoid => void

For example

let a := 1

introduces an integer variable with initial value 1. Notice that the compiler deduces the type.

A constant is declared by

let <identifier> = <clause>

For example

```
    let discrim = b * b - 4.0 * a * c
```

introduces a real constant with the calculated value. The compiler will detect and flag as an error any attempt to alter a constant.

6.2 Sequences

A sequence of clauses is made up of any mixture, in any order, of declarations and clauses. The type of the sequence is the type of the last clause or declaration. If there is more than one clause in a sequence then all but the last must be of type void.

<sequence> ::= <declaration>[;<sequence>] | <clause>[;<sequence>]

t : type, <declaration> : void ;<sequence> : t => t
t : type, <clause> : void ;<sequence> : t => t

6.3 Brackets

Brackets are used to make a sequence of clauses and declarations into a single clause. They are

```
begin                                  ! or     { < sequence > }
        < sequence >
end
```

t : type, **begin**<sequence> : t **end** => t
t : type, {<sequence> : t } => t

The {} method is there to allow a clause to be written clearly on one line. For example

```
let i := 2
for j = 1 to 5 do { i := i * i ; write i }
```

However, if the clause is longer than one line the first alternative should be used for greater clarity. Nonvoid blocks are sometimes called block expressions.

6.4 Scope Rules

Any identifier that is declared has its scope limited to the following sequence. This means that the scope of an identifier starts immediately after the declaration and continues up to the next unmatched } or **end**. For example

```
                              let sum = 3
^                             let std := sum + 15.0
|                             let var = readr ()
scope  ^                      if sum = var do
 of    |                      begin
sum     scope                       write sum,std,var
|       of                          let sum.sq = sum * std
|       var     ^                   write sum.sq,sum.sq * sum.sq
|       |       scope of
|       |       sum.sq        end
```

If the same identifier is declared in an inner sequence, then while the inner name is in scope the outer one is not.

7. Compound Data Objects

PS-algol allows the programmer to group together data objects into larger compound objects which may then be treated as single objects. There are three such object types in PS-algol : vectors, structures and images. If the constituent objects are of the same type a vector is used and a structure otherwise. Images are collections of pixels. Vectors, structures and images have the full **civil rights** of any other data object in PS-algol.

All compound data objects in PS-algol have pointer semantics. That is, when a compound data object is created, a pointer to the locations that make up the object is also created. The object is always referred to by the pointer which may be passed around by assignment and tested for equality. The location containing the pointer and the constituent parts of the compound data object may be independently constant or variable.

7.1 Vectors

A vector provides a method of grouping together objects of the same type. Since PS-algol does not allow undefined values all the initial values of the elements must be specified. The syntax is

```
<vector_constructor>::=vector<bounds>of<clause> |
                       @<clause>of<type1><lsb><clause_list><rsb>
<bounds>            ::= <clause>::<clause>[,<bounds>]

t : nonvoid, vector <bounds> of <clause> : t => *t
where <bounds> ::= <clause> : int :: <clause> : int [,<bounds>]

t : nonvoid,
        @<clause> : int of <type1><lsb><clause_list><rsb> => *t
where <clause_list> ::= <clause> : t [,<clause_list>]
```

For example

@1 **of int** [1,2,3,4]

is a vector of integers, i.e ***int**, with lower bound 1 and values 1, 2, 3 and 4. The elements are variable. Also

let abc := @1 **of int** [1,2,3,4]

introduces a variable 'abc' of type ***int** and that initial value.

Multi dimensional vectors which are not necessarily rectangular can also be created. For example

```
let Pascal = @1 of c*cint[      @1 of cint[ 1 ],
                             @1 of cint[ 1,1 ],
                             @1 of cint[ 1,2,1 ],
                             @1 of cint[ 1,3,3,1 ],
                             @1 of cint[ 1,4,6,4,1 ],
                             @1 of cint[ 1,5,10,10,5,1 ] ]
```

'Pascal' is of type **c*c*cint**. It is constant as are all its elements. This is a fixed table.

This form of vector expression is sometimes very tedious to write especially for large rectangular vectors with a common initial value. Therefore another form of vector expression is available. For example

vector -1::3 **of** -2

produces a five element integer vector with all the elements variable and initialised to -2.

In order that the compiler can check that no constant will be updated it is necessary that objects of type *T and type *cT are made type incompatible.

7.1.1 upb and lwb

Since vectors may be assigned, it is often necessary to interrogate the vector to find its bounds. The functions **upb** and **lwb** are provided in PS-algol for this purpose.

```
<bounds_op>(<expression> : *nonvoid) => int
where <bounds_op> ::= lwb | upb
```

7.2 Structures

Objects of different types can be grouped together into a structure. To ensure strong typing in the language the structure class and the fields have to be given names (which have to be unique in the block). The fields may of course be constant or variable.

Before a structure can be created the shape of its structure class must be declared. The syntax is

```
<structure_decl>  ::=  structure<identifier>[(<field_list>)]
<field_list>      ::=  <type1><identifier_list>[;<field_list>]
```

For example

structure person(**cstring** name ; **cbool** sex ; **int** age,height)

declares a structure class, 'person', with four fields of type **cstring**, **cbool**, **int** and **int** respectively. The string field is constant. It also declares the field names, 'name', 'sex', 'age' and 'height'. Their order is significant. To create such an object we use

```
<identifier>([<clause_list>])

<identifier>([<clause_list>]) => pntr
```

where<clause_list> ::= <clause> **: nonvoid** [,<clause_list>]

For example, the following expression is of type **pntr**.

person("Ronald Morrison",**true**,33,175)

7.2.1 is and isnt

The user may wish to check that a pointer is of a certain class. The binary operators **is** and **isnt** are provided. If the pointer is of the same class **is** gives the result **true** and **isnt** gives **false**. For example

P **is** person

All structure classes belong to the same infinite union.

7.2.2 Equivalence of Structure Classes

Two structure classes, which may be in separate programs, are considered the same if they have the same class identifier, field names and types in one to one correspondence and are different otherwise.

7.3 Indexing

To obtain the elements of a vector or the fields of a structure, indexing is used. For a vector, the index is an integer and for a structure the field name is used.

Before any indexing is performed the bounds of the vector are checked against the index for legality and in the case of the structure, the class of the structure is checked (hence the requirement for field names to be unique in a scope).

A comma notation may be used for vectors or structures when the elements or fields are themselves pointers or vectors. The indexing of vectors and structures may therefore be freely mixed. For example, if 'v' is a vector of vectors of persons then 'v(i)(j)(name)' and 'v(i,j,name)' and 'v(i,j)(name)' would be equivalent expressions.

7.4 Image Expressions

An image is a rectangular grid of pixels. Images may be created and manipulated using the raster operations provided in the language. The operations on images are defined by

<image_constructor> ::= **image**<clause>**by**<clause>**of**<clause> |

limit<clause>[**to**<clause>**by**<clause>][**at**<clause>,<clause>]

image<clause> : int **by**<clause> : int **of**<clause> : pixel => #pixel

t : image, **limit**<clause> : t [**to**<clause> : int **by**<clause> : int]
[**at**<clause> : int,<clause> : int>] => t

The integer values above must be ≥ 0 and are subjected to an upper bound check. An image is a 3 dimensional object made up of a rectangular grid of pixels. To form an image we could write

let c = **image** 5 **by** 10 **of on**

which creates 'c' with 5 pixels in the X direction and 10 in the Y direction all initially on. The origin of all images is 0,0 and in this case the depth is 1.

Full three dimensional images may be formed by, for example

let d = **image** 64 **by** 32 **of on** & **off** & **on** & **on**

Images are first class data objects and may be assigned, passed as parameters or returned as results.

let b := a

will assign the image 'a' to the new one 'b'. In order to map the operations usual on bitmapped screens, the assignment does not make a new copy of 'a' but merely copies the pointer to it. Thus the image acts like a vector or pointer on assignment.

There are 8 raster operations which may be used as described in the following syntax.

```
<raster>      ::=  <raster_op><clause>onto<clause>
<raster_op>   ::=  ror | rand | xor | copy | nand | nor | not | xnor

<ratser_op><clause> : image onto<clause> : #pixel => void
```

thus

xor b **onto** a

performs a raster operation of 'b' onto 'a' using **xor**. Notice that 'a' is altered 'in situ' and 'b' is unchanged. Both images have origin 0,0 and automatic clipping at the extremities of the destination image is performed.

The raster operations are performed by considering the images as bitmaps and altering each bit in the destination image according to the source bit and the operation. The operations mean

ror	inclusive or
rand	and
xor	exclusive or
copy	overwrite
nand	not and
nor	not inclusive or
not	not the source
xnor	not exclusive or

The limit operation allows the user to set up windows in images. For example

let c = **limit** a **to** 1 **by** 5 **at** 3,2

sets 'c' to be that part of 'a' which starts at 3,2 and has size 1 by 5. 'c' has an origin of 0,0 in itself and is therefore a window on 'a'.

Rastering sections of images on to sections of other images can be performed by, for example

xor limit a **to** 1 **by** 4 **at** 6,5 **onto limit** b **to** 3 **by** 4 **at** 9,10

Automatic clipping on the edges of the limited region is performed. If the starting point of the limited region is omitted 0,0 is used and if the size of the region omitted then it is taken as the maximum possible. That is, from the starting point to the edges of the host image. Limited regions of limited regions may also be defined.

If the source and destination images overlap, then the rester operation is performed in such a manner that each bit is used as a source before it is used as a destination.

The standard identifier 'screen' is an image representing the output screen. Performing a raster operation onto the image 'screen' alters what may be seen by the user. The standard identifier 'cursor' is an image representing the cursor. The cursor may be altered in the same manner as any image.

In systems that support multiple planes the standard identifiers screen and cursor will have a depth greater than 1. All the operations that we have already seen on images(raster, limit and assignment) work more generally with depth. Thus the raster operations perform the raster function plane by plane in one to one correspondence between source and destination. Automatic depth clipping at the destination is performed and if the source is too small to fill all the destination's planes then these planes will remain unaltered. The limit and assignment operations also work with the depth of the image.

The depth of the image may be restricted by the depth selection operation. For example

let b = a(1|2)

yields 'b' which is an alias for that part of 'a' which has the two depth planes 1 and 2. 'b' has depth origin 0 and dimensions 64 by 32.

The full syntax of the depth selection operation is

```
<exp5>          ::=    <name>[(<clause><bar><clause>)]
```

t : image, <name> : **t** (<clause> : **int** <bar><clause> : **int**) **=> t**

In programs , the type image is written as **#pixel** or **#cpixel** and it should be noted that objects of type **#pixel** are assignment incompatible with objects of type **#cpixel** (cf vectors). An image of type **#cpixel** may be used as the source of a raster operation but not the destination. Furthermore a limit operation on an image of type **#cpixel** produces another of the same type.

There is a standard function that allows the user to detect the position of the mouse relative to the screen image. The function is called 'locator' and has form,

locator(-> **pntr**)

and when called it will return a structure when the mouse movement or button depression satisfies a condition which is implementation dependent. The structure class is defined in the PS-algol prelude by

structure mouse(**cint** X.pos,Y.pos ; **cbool** selected ; **c*cbool** the.buttons)

The boolean 'selected' indicates whether the screen is currently selected for input or not. The vector of booleans indicates which buttons are depressed. The order of these buttons depends on operating system on which PS-algol is executing.

A convenient and efficient method of writing to a specific image in a specific font is provided by the **print** clause.

```
<print>            ::=
print<print_list>[onto<clause>][at<clause>,<clause>]
                          [in<clause>][using<raster_op>]
<print_list>  ::=   <clause>[:<clause>][,<print_list>]
```

print<print_list>[**onto**<clause> : pntr]
[**at**<clause> : int,<clause> : int][**in**<clause> : pntr]
[**using**<raster_op>] => void
where <print_list> ::= <clause> : nonvoid[:<clause> : int][,<print_list>]

print causes a sequence of values provided by the print_list to be interpreted and placed on an image. The destination for the output is specified by the **onto** clause, which should be a 'print.destination' structure ; the default destination is 'enhanced.screen'. The **at** clause indicates the location on the destination image at which to place the next left hand origin of the tile generated ; it defaults to the right hand origin of the last tile placed. The **in** clause specifies a font in which parts of the print_list which are converted to characters will then be converted to tiles. The **using** clause specifies how to combine tiles with the destination and defaults to **copy**. All of these clauses, **onto**, **at**, **using** and **in** take effect from the start of the **print** clause and continue in effect until explicitly changed. If any clause within the **print** clause itself contains a **print** clause, the inner **print**s will occur before the outer **print** starts, as the clauses forming the outer **print** are evaluated left to right. Should any part of the tile being combined with the destination fall outside the target, an 'edge.violation' will occur. There is a default 'edge.violation' handler (which will usually provide scrolling) activated by this event and the data structure describing the destination provides a means of changing this behaviour. Further details of this data structure and associated printing data may be found in appendix VI.

7.5 Assignment and Equality of Pointers

Equality is defined on all data objects in PS-algol. In the case of compound objects equality means the equality of the pointer. That is, for two vectors or structures to be equal they must be the same incarnation of the same object. It follows that assignment of a compound data object means copying the pointer only.

8. Procedures

Procedures in PS-algol constitute the abstractions over expressions, if they return a value, and clauses of type void if they do not. In accordance with the principle of correspondence any method of introducing a name in a declaration has an equivalent form as a parameter.

Thus, in declarations of data objects, giving a name an initial value is equivalent to assigning the actual parameter value to the formal parameter. Since this is the only type of declaration for data objects in the language, it is also the only parameter passing mode and is commonly known as **call by value**.

Like declarations, the formal parameters representing data objects must have a name, a type and an indication of whether they are variable or constant. A procedure which returns a value must also specify its return type.

Structure classes and associated fields may also be passed as parameters to complete the principle of correspondence. For type checking, the argument and result types of the procedures and the field types of structure classes must be given in full when they are passed as parameters. Whereas the constancy of a formal parameter of a procedure which is itself a formal parameter is immaterial it is not so with structure fields and to avoid the possibility of altering a constant, the constancy attribute of the fields must be the same in this case.

The syntax of the procedure call is

> <expression>([<clause_list>])
>
> <expression> : **proc** ([<clause_list>]) **=> type**
> **where** <clause_list> ::= <clause> : **nonvoid** [,<clause_list>]

There must be a one-to-one correspondence between the actual and formal parameters and their types.

8.1 Recursive Declarations

In PS-algol all identifiers must be declared before they can be used and an identifier comes into scope immediately after its declaration. This is awkward when recursive procedure definitions are involved. Note that

> **let** fac = **proc**(**int** n -> **int**) ; **if** n = 0 **then** 1 **else** n * fac(n - 1)

is not a recursive definition and is only legal if 'fac' has already been declared in an outer scope. To get recursion we must write

> **let** fac := **proc**(**int** n -> **int**) ; **nullproc**

```
fac := proc( int n -> int ) ; if n = 0 then 1 else n * fac( n - 1 )
```

8.2 Equality of Procedures

Two procedures are equal in PS-algol if and only if their values are derived from the same evaluation of the same procedure expression. For the cognoscenti this means that they have the same closure.

9. Clauses

The expression is a special type of clause which allows the operators in the language to be used to produce data objects. There are other statements in PS-algol which allow the data objects to be manipulated and some which are there to control the flow of the program.

9.1 Assignment Clause

The assignment clause has the following syntax.

```
<exp5>      ::=    <name>:=<clause>
```

t : **nonvoid**, <name> : t :=<clause> : t => **void**

For example

```
discriminant := b * b - 4.0 * a * c
```

gives 'discriminant' the value of the expression on the right. Of course, the name must have been declared as a variable and not a constant. The clause alters the value denoted by the name.

9.2 if Clause

There are two forms of the **if** clause defined by

if<clause>**do**<clause> | **if**<clause>**then**<clause>**else**<clause>

if <clause> : **bool** **do** <clause> : **void** => **void**
t : **type**, **if** <clause> : **bool** **then** <clause> : t **else** <clause> : t => t

In the single pronged version, if the condition after the **if** is true then the clause after the **do** is executed. For example

if a < b **do** a := 3

The second version allows a choice between two actions to be made. If the first clause is **true** the second clause is executed, otherwise the third clause is executed. Notice that the second and third clauses are of the same type and the result is of that type. For example

if x = 0 **then** y := 1 **else** x := y - 1
let temp = **if** a < b **then** 1 **else** 5

9.3 case Clause

The **case** clause is a generalisation of the **if** clause which allows the selection of one item from a number of possible ones. The syntax is

case<clause>**of**<case_list>**default** :<clause>
<case_list>   ::=   <clause_list>:<clause>;[<case_list>]

t : type ; t1 : nonvoid, **case** <clause> : t1 **of** <case_list>
default : <clause> : t => t

where <case_list> ::= <clause_list>:<clause> : t ; [<case_list>]
where <clause_list> ::= <clause> : t1 [,<clause_list>]

An example of the use of the **case** clause is

case next.car.colour **of**
1,4 : "green"
2 : "blue"
3 : "red"
default : "any"

The value 'next.car.colour' is compared in strict order, i.e left to right, top to bottom, with the expressions on the left hand side of the colon. When a match is found the clause on the right hand side is executed. Control is then transferred to the next clause after the **case** clause. If no match is found then the default clause is executed. The above **case** clause has result type **string**.

9.4 repeat ... while ... do Clause

There are three forms of this clause which allow loops to be set up with the test at the start, the end or the middle of the loop. The three forms are encapsulated in the two productions.

repeat<clause>**while**<clause>[**do**<clause>] | **while**<clause>**do**<clause>

repeat <clause> : void **while** <clause> : bool
[**do** <clause> : void] => void
while <clause> : bool **do** <clause> : void => void

In each of the three forms the loop is executed until the boolean clause is **false**. The **while do** version is used to perform a loop zero or many times whereas the **repeat while** is used for one or many times.

9.5 for Clause

The **for** clause is included in the language as a bit of syntactic sugar where it is known in advance how many times the loop will be executed. It is defined by

for<identifier>=<clause>**to**<clause>[**by**<clause>]**do**<clause>

for <identifier>=<clause> : int **to** <clause> : int
[**by**<clause> : int]**do**<clause> : void => void

The clauses are the initial value, the limit, the increment and the clause to be repeated respectively. The first three are of type int and are calculated only once at the start. If the increment is 1 then the **by** clause may be omitted. The identifier or control constant is declared at the start of the void clause taking on the values defined by initial value, increment and limit. For example

```
**let** factorial := 1 ; **let** n = 8
**for** i = 1 **to** n **do** factorial := factorial * i
```

With a positive increment, the **for** loop terminates when the control constant is initialised to a value greater than the limit. With a negative increment, the **for** loop terminates when the control constant is initialised to a value less than the limit.

9.6 abort Clause

The reserved word **abort** simply stops the program when used.

9.7 Exceptions

An exception in PS-algol is an exception condition together with an associated excepted value. The excepted value is always of type **pntr**. The termination model of exceptions is supported.

An exception is raised by a **raise** statement

raise<clause>

raise<clause> : pntr => void

The structure class and its fields provide data for the handler on an error. The execution of the **raise** clause causes the current flow of control to be transferred to the clause of the most recently declared handler capable of handling this exception. The clause following **do** is then evaluated, and on its completion control leaves the block containing the handler.

Associated with the exception condition is an excepted value of type **pntr**. The result of the pointer expression in the **raise** statement becomes the excepted value. For example:

structure foot(**cstring** size)
raise foot("big")

The excepted value here is an instance of the structure class whose name is foot. If the structure class of the excepted value has fields, then we have the functionality of parameterised exceptions.

Exception handlers are declared using the **when** statement.

when <ex_id_list>[**as** <identifier>] **do** <clause>
<ex_id_list> ::= **any** | <identifier_list>

t : type, **when**<ex_id_list>[**as**<identifier>]**do**<clause> : t => t

For example:

when foot **do write** "12 inches'n"

The excepted value can be made available to the handler by using the **as**<identifier> phrase. Its type is **cpntr**. A catch-all handler is provided by using the reserved word **any**.

We may specify a list of exceptions that can be fielded by the handler. For example

```
structure arm
when toe,arm do write "limb"
```

The handler clause serves as an alternative completer for the enclosing block. The type of the clause must therefore match that of the block. For example

```
let len=
begin
    structure toe ( cint inches )
    when any do 0
    when toe as e do 1+e( inches )
     ...
    raise toe ( 3 )
     ...
    12
end
```

Here, the normal result of the block is the integer 12. If, however, the **raise** 'toe' statement is executed, the result will be 4. If any other exception is raised, the result will be 0.

9.8 System Events and System Exceptions

The PS-algol system may detect a number of system events. These may derive from receipt of hardware signals, for example keyboard interrupt, or from run-time errors such as attempting to apply a **nullproc**. The action taken, on the occurrence of each possible system event, is determined by a **proc**, called here an event procedure. These event procedures are visible to user programs via the global **pntr** 'events'.

If an exception is raised but is not handled by the user program, the event procedure 'events'('Uncaught.event') is invoked. This procedure displays diagnostics for all system exceptions, and makes an attempt at user defined exceptions. The full range of system events and system exceptions are described in Appendix V.

10. Input and Output

Because of the persistence hooks for data in PS-algol, file I/O is generally unnecessary. However communication with the user is still required and PS-algol provides functions to read and write to the user's terminal and to files.

10.1 Input

The read functions regard the file as a stream of ASCII characters. The functions mean

read()	read the next character in the input.
readi()	ignore preceding layout and read an integer literal.
readr()	ignore layout characters and read a real literal.
readb()	ignore layout characters and read a boolean literal.
reads()	ignore layout characters and read a string literal.
peek()	look at the next character without consuming it.
read.a.line()	read from the current position up to a newline symbol. Give the result as a string without the newline symbol.
read.byte()	read one 8 bit byte as an integer.
eoi()	test for end of input.

t : type, <standard_exp> : t => t
where<int_read>([<clause> : file]) => int
where<int_read> ::= **readi** | **read.byte**
where<string_read>([<clause> : file]) => string
where<string_read> ::= **read** | **peek** | **reads** | **read.a.line**
where<bool_read>([<clause> : file]) => bool
where<bool_read> ::= **readb** | **eoi**
where **readr**([<clause> : file]) => real
where **read.name**(<clause> : string [,<clause> : file]) => string

Layout characters are those characters that control the position of the printed characters but do not themselves mark the paper; they are newline, space, and horizontal tab. The functions **readi**, **readr**, **readb** and **reads** will ignore any layout characters they encounter before the required symbol. If they encounter other characters before they encounter a symbol of the correct type, they call an events procedure.

If the file clause is omitted in an input operation, the input will operated on the standard input file 's.i'. Otherwise it will operate on the file indicated.

10.2 End of Input

A test for end of input can be made using the standard function **eoi**. For example

```
while ~eoi() do write read()
```

copies the input stream to the output stream until the end of the input stream is reached.

10.3 Output

Output will be to the file standard output, 's.o' when designated by the **write** clause. Otherwise it will be directed to the designated file. The syntax for output is

```
<write>       ::=   write<write_list> | output<clause>,<write_list> |
                    out.byte<clause>,<clause>,<clause>
<write_list> ::=    <clause>[:<clause>][,<write_list>]

<write> => void
where <write_list> ::= <clause> : printable
                             [:<clause> : int][,<write_list>]
where output<clause> : file ,<write_list>

where out.byte<clause> : int,<clause> : int,<clause> : int
```

For example

write "I am O.K.", 32160, 3.4e-2, "Done"

will write that list of objects and

write 13.2:16, 3:10, 152:3

will write 13.2, 3 and 152 right justified in fields of size 16, 10 and 3 respectively. Note that the field may be any integer expression allowing variable formats.

10.4 i.w, s.w and r.w

Types **int** and **real** have one other facility. The predefined integer variables

i.w	integer width initially 12
s.w	space width initially 2
r.w	real width initially 14

may be used to control output. 's.w' spaces are written out after any integer or real. 'i.w' and 'r.w' are used to set the field sizes for integers and reals respectively, but may be overridden by the field size in the **write** clause. If the number fits neither then the exact size is written.

10.5 Input from and Output to files

The above input and output facilities operate on data on the standard input 's.i', and standard output 's.o', streams. However, their actions may also be redirected to specific files by one of two methods, by assigning a file descriptor to 's.i' or 's.o' or by using corresponding I/O operation forms with an additional parameter, a file descriptor. File descriptors are obtained as the result of the 'open' and 'create' function calls. The definition of these calls is implementation dependent. Three other functions **seek**, **flush**, and **close** operate on file descriptors. **seek** sets a position in a file, **flush** forces the results of preceding output operations on this file to be recorded on disc, and **close** indicates a file is no longer required by this program. An automatic close of files may occur at the end of the program, but explicit closure will release resources, and in some implementations may be required in certain circumstances.

11. Persistent data

The persistence of data is the length of time that the data exists. In PS-algol any data item is allowed the full range of persistence. It is necessary for the programmer to identify which data is to persist and in which database it should persist. This section describes the mechanisms for the storage and retrieval of persistent data.

The mechanisms are available via a set of standard procedures which are described below. The actual transfer of data is automatic, data being brought into the program's active heap when the program attempts to access it, and that data which may still be accessible being migrated back at times which are left to the discretion of the implementor or on the insistence of the programmer.

The procedures are divided into two groups, the group concerned with identifying the relationship between data and database, and the implementation of transactions, and the group concerned with providing a new data structure, tables.

11.1 Tables

Tables are a system supported data structure in PS-algol. They are commonly used and needed for building databases, but may also be used for more temporary structures. A table stores an updatable mapping from keys to values. The keys may be integers or strings, and the values are pointers to instances of any structures. The implementor will probably have used B-trees or some adaptive hashing technique such as hashed trees to implement these maps.

> **let** table = **proc**(-> **pntr**)

This procedure creates a new empty table and returns a pointer as a token for it.

> **let** s.enter = **proc**(**string** key ; **pntr** table,value)

> **let** i.enter = **proc**(**int** key; **pntr** table,value)

These two procedures may be used to modify the entries in the table given as the parameter table. A table may contain entries whose keys are integers or entries whose keys are strings; a key of one type never matches a key of the other. A new association is recorded in the table between the key and the value, this supersedes any previous association for that key that was held in the table. If the value is **nil** the effect is to remove any existing entry for the given key from the table.

> **let** s.lookup = **proc**(**string** key ; **pntr** table -> **pntr**)

let i.lookup = **proc**(**int** key ; **pntr** table -> **pntr**)

These procedures return the value associated with the given key from the given table. If there is no entry for that key then the result is **nil**.

let s.scan = **proc**(**pntr** table ; **proc**(**string**,**pntr** -> **bool**) user -> **int**)

let i.scan = **proc**(**pntr** table ; **proc**(**int**,**pntr** -> **bool**) user -> **int**)

These two routines apply the function provided as 'user' to the entries in the stored table given as the first parameter. The function is applied to every element with a string key by 's.scan' and to every element with an integer key by 'i.scan'. The function is repeatedly called with a key as its first parameter and the associated value as the second parameter. Repetition continues until either all the entries for the specified type of key have been parameters to 'user' or until 'user' returns **false**. The result of the scan function is the number of times user was called.

11.2 Database procedures

All data which persists longer than a program execution is held in some database, any given item being in only one database. Pointers may refer to items in other databases. A database is identified by a database name which will often have the same syntactic form as that used for identifiers. A database may be created by:

let create.database = **proc**(**string** database.name,password -> **pntr**)

'password' is a string which must be quoted correctly to gain access.

A check is made that no other existing database has the same name as the one to be created. If this is successful a database is created and opened for writing. A table is inserted in the database and a pointer to it returned. If the 'create' is unsuccessful then an exeception will be raised. A database may be opened by:

let open.database = **proc**(**string** database.name,password,mode -> **pntr**)

'password' is a string which must be quoted correctly to gain access in the mode requested. 'mode' must have one of the values "read" or "write". A database may be opened by many users for reading or by one user for writing.

A check is made that the database is compatible with any others which are open, whether the user has quoted the password correctly and whether other programs are using this database in a mode incompatible with that requested. Any other databases that may be referenced by objects in that database are recursively opened in read mode so that every object encountered will refer to an object in an open database. If any of these checks or the recursive open fails then an exeception will be raised, otherwise the result is a table.

Whatever mode the database was opened in, the programmer may now access data in that database, and change the data so accessed. However, no changes are recorded unless and until the program executes a call of 'commit'

let commit = **proc**(-> **pntr**)

This procedure commits the changes made so far to all the databases opened by the program. Either all or none of the changes will be recorded, so the programmer can use this as a device for ensuring that databases are consistent. It is only after a 'commit 'that the changes made can be observed by other programs. Note that only the changes prior to the last 'commit' of a program, if any, are recorded in the databases. It is an error to perform 'commit' when any database containing changed objects is not open for writing. In this case an exeception will be raised. If the 'commit' is successful the result is the **nil** pointer.

11.3 Database conventions

The value returned by 'open.database' or 'create.database' is a pointer to a table, conventionally called the root table. Data will be preserved for as long as it is reachable from some entry in a root table via the transitive closure of all references.

12. Standard Functions and Identifiers

12.1 Public Standard Functions

The following standard functions comprise only those intended for the general user.

let sqrt = **proc**(**real** x -> **real**)
! the positive square root of x where x ≥ 0

let exp = **proc**(**real** x -> **real**)
! e to the power x

let ln = **proc**(**real** x -> **real**)
! the logarithm of x to the base e where x > 0

let sin = **proc**(**real** x -> **real**)
! sine of x(radians)

let cos = **proc**(**real** x -> **real**)
! cosine of x(radians)

let atan = **proc**(**real** x -> **real**)
! arctangent of x (radians) where - pi / 2 < atan(x) < pi / 2

let code = **proc**(**int** n -> **string**)
! string of length 1 where s(1|1) = character with numeric code abs(n **rem** 128)

let decode = **proc**(**string** s -> **int**)
! numeric code for s(1|1)

let truncate = **proc**(**real** x -> **int**)
! the integer i such that |i| ≤ |x| < |i| + 1 where i * x ≥ 0

let line.number = **proc**(-> **int**)
! the program line number

let rabs = **proc**(**real** x -> **real**)
! the absolute value of real number x

let abs = **proc**(**int** n -> **int**)
! the absolute value of integer n

let length = **proc**(**string** s -> **int**)
! the number of characters in the string s

let eformat = **proc**(**real** n ; **int** w,d -> **string**)
! the string representing n with w digits before the decimal
! point and d digits after with an exponent

let fformat = **proc**(**real** n ; **int** w,d -> **string**)
! the string representing n with w digits before the decimal point and d digits after

let gformat = **proc**(**real** n -> **string**)
! the string representing n in eformat or fformat whichever is suitable

let letter = **proc**(**string** s -> **bool**)
! length(s) = 1 **and**
! s ≥ "A" **and** s ≤ "Z" **or**
! s ≥"a" **and** s ≤ "z"

let digit = **proc**(**string** s -> **bool**)
! length(s) = 1 **and** s ≥ "0" **and** s ≤ "9"

let iformat = **proc**(**int** n -> **string**)
! integer n as a string of characters

let enhance.image = **proc**(**c#pixel** im -> **pntr**)
! return a print destination structure from the image

let options = **proc**(-> ***cstring**)
! the command line options are given as a vector of strings

let find.substr = **proc**(**string** target,substring -> **int**)
! return the starting position of string 'substring' in 'target', zero otherwise.

let interrupt = **proc**(-> **bool**)
! return whether an interrupt (normally control C) has been received
! since the last call of this procedure or the start of the program.

let date = **proc**(-> **string**)
! gives the date and time in a format determined by the implementation

let time = **proc**(-> **int**)
! returns the CPU time used by the running program as the number of 60Hz clock ticks.

let random = **proc**(**cint** seed -> **int**)
! random number generator as described in CACM vol 11,9 p642

let trace = **proc**()
! print a snapshot of the current procedure calls.

let environment = **proc**(-> ***cstring**)
! the environment vector for the process is returned as a
! vector of strings, (this is only available in UNIX).

let i.lookup := **proc**(**int** i ; **pntr** t -> **pntr**)
! this looks up pointer value in the table t corresponding to integer key i

let s.lookup := **proc**(**string** s ; **pntr** t -> **pntr**)
! this looks up pointer value in the table t corresponding to string key s

let i.enter := **proc**(**int** i ; **pntr** t,v)
! the pointer value v is entered into table t using integer key i

let s.enter := **proc**(**string** s ; **pntr** t,v)
! the pointer value v is entered into table t using string key s

let table := **proc**(-> **pntr**)
! a new table is created and a pointer to it returned

let i.scan := **proc**(**pntr** t ; **proc**(**int**,**pntr** -> **bool**) u -> **int**)
! every value in table t with an integer key is passed to procedure u until the
! procedure u returns **false** or all integer keys have been encountered; the
! result is the number of times procedure u is applied

let s.scan := **proc**(**pntr** t ; **proc**(**string**,**pntr** -> **bool**) u -> **int**)
! every value in table t with a string key is passed to procedure u until the
! procedure u returns **false** or all string keys have been encountered; the
! result is the number of times procedure u is applied

let commit = **proc**(-> **pntr**)
! all changes made to objects in the persistent store are made permanent,
! any objects reachable from changed objects are added to the persistent
! store; if successful **nil** is returned otherwise an error record.

let create.database = **proc**(**cstring** db.name,pass -> **pntr**)
! a database is created with name db.name and password pass; if successful
! a pointer to a table is returned otherwise an error.record

let open.database = **proc**(**cstring** db.name,pass,mode -> **pntr**)
! a database is opened with name db.name and password pass in mode "read"
! or "write"; if successful a pointer to a table is returned otherwise an
error.record

let change.passwd = **proc**(**cstring** db.name,opass,npass -> **pntr**)
! the database db.name has its password changed to npass if it is
! currently set to opass; if successful **nil** is returned otherwise an error.record

let change.owner = **proc**(**cstring** db.name ; **cint** nowner -> **pntr**)
! the database db.name has its ownership number changed to nowner, this
! may only be done by the PS-algol system owner; if successful **nil** is
! returned otherwise an error.record

let list.database.dir = **proc**(-> **pntr**)
! every database is listed in detail on standard output
! if successful **nil** is returned otherwise an error.record

let rename.database = **proc**(**cstring** oname,nname,pass -> **pntr**)
! the database oname is renamed nname if its password is pass and there isn't a
database
! called nname; if successful **nil** is returned otherwise an error.record

let remove.database = **proc**(**cstring** db.name,pass -> **pntr**)
! the database db.name has its name removed from the list of databases if
! its password is pass; if successful **nil** is returned otherwise an
! error.record, the database can then be deleted if no other database

! has references to it during disk garbage collection.

let restore.database = **proc**(**cstring** db.name -> **pntr**)
! the database db.name is restored to a consistent state from its before
! look file and any locks on it are released; if successful **nil** is returned
! otherwise an error.record, this procedure ignores locks held on the database

let line = **proc**(**c#pixel** i ; **cint** x1,y1,x2,y2,dx,dy ; **proc**(**c#pixel**)style)
! execute procedure 'style' along the line from x1, y1 to x2, y2 at intervals
! of dx, dy.

let draw = **proc**(**c#pixel** i ; **cpic** p ; **creal** x1,x2,y1,y2)
! draw the picture p on the image i.
! the picture is bounded by x1,x2,y1,y2 in its coordinate space.

let device.draw = **proc**(string device.type -> proc(cfile,cpic,creal,creal,creal,creal))
! this procedure returns an outline draw function to draw pictures on
! terminals - the parameter 'device.type' specifies which draw function is required

let X.dim = **proc**(**c#pixel** i -> **int**)
! return the x dimension of i

let Y.dim = **proc**(**c#pixel** i -> **int**)
! return the y dimension of i

let locator = **proc**(-> **pntr**)
! returns a structure containing information about the status of the mouse.
! the structure returned is a mouse

let cursor.tip = **proc**(**cpntr** the.tip -> **pntr**)
! make the effective tip of the cursor 'the.tip' return old tip
! both pointers are pointers to point.strc(**cint** point.x,point.y)

let cursor.on = **proc**()
! make the cursor track the mouse (the default state).

let cursor.off = **proc**()
! make the cursor invisible.

let Pixel = **proc**(**c#pixel** i ; **cint** xpos,ypos -> **pixel**)
! return the pixel at xpos,ypos in i.

let constant.image = **proc**(**c#pixel** i -> **#cpixel**)
! return a copy of image i with constant pixels.

let variable.image = **proc**(**c#cpixel** i -> **#pixel**)
! return a copy of image i with variable pixels.

let fill = **proc**(**c#pixel** i ; **cpixel** col ; **cint** xpos,ypos)
! seed fill image i from position xpos,ypos with the pixel col.

let menu = **proc**(**c#pixel** title ; **c*#pixel** entries ;
                cbool vertical ; **c*proc**(**c#pixel**,**cint**) actions
                -> **proc**(**cint**,**cint** -> **bool**))
! Return a procedure that when called will cause a menu to appear with

! its bottom left hand corner at position xpos,ypos relative to 'screen'
! 'vertical' indicates the menu orientation
! The menu has a 'title' and a vector of icons called 'entries'
! Associated with each entry in the menu is a procedure which
! will be called when the entry is selected.
! This procedure is passed the associated image and the entry number
! The boolean returned indicates if the user made a selection

let depth = **proc**(**c#pixel** i -> **int**)
! return the number of planes in image i

```
let colour.map = proc( cpixel p ; cint i )
! when pixel p is displayed the integer i will be sent to the display hardware

let colour.of = proc( cpixel p -> int )
! return the integer sent to the hardware when pixel p is displayed

let string.to.tile = proc( cstring the.string,font -> #pixel )
! returns an image which contains 'the.string' in the 'font'
! now superseded by print on some machines.

! These procedures are used in file I/O.

let read.real := proc( cfile f -> real )
! this reads characters from file f to form a real number

let writer := proc( cfile f ; creal n ; cint a,b ) ;
! this writes a real number n to file f in a field
! width of a with b spaces after the number

let open := proc( cstring f ; cint m -> file )
! this opens the file with name f in mode m
! modes are:- 0 - read only, 1 - write only, 2 - read and write

let close := proc( cfile f )
! this closes the file for file descriptor f

let seek := proc( cfile f ; cint offset,key )
! this moves offset bytes through file f from 0 - start of file,
! 1 - current position and 2 - end of file

let create := proc( cstring f ; cint m -> file )
! this created a file with name fname in mode m, m is the decimal value
! of the file protection bitmap.

let flush := proc( cfile f )
! this causes the output buffer for file f to be written out

let read.blocks := proc( file f ; int blks -> *int )
! this read 'blks' four byte words from the file into an integer vector

let write.blocks := proc( file f ; *int blks )
! this writes a vector of integers to a file

let shift.r := proc( int value,count -> int )
! shift the first parameter right 'count' places bringing in zeros at the high order
end
```

let shift.l := **proc**(**int** value,count -> **int**)
! shift the first parameter left 'count' places bringing in zeros at the low order end

let b.and := **proc**(**int** value1,value2 -> **int**)
! logical (bitwise) 'and' of 'value1' 'and value2'

let b.or := **proc**(**int** value1,value2 -> **int**)
! logical (bitwise) 'or' of 'value1' 'and value2'

let b.not := **proc**(**int** i -> **int**)
! boolean 'not' operator

let mat.mult := **proc**(**c**real** a,b -> ****real**)
! multiply two matrices of real numbers

let createdb := **proc**(**cstring** db.name,pass -> **pntr**)
! this creates a database with name db.name and password pass, the root object is a
! opdb.result structure; if successful **nil** is returned otherwise an error.record

let opendb := **proc**(**cstring** db.name,pass ; **cint** m -> **pntr**)
! this opens the database with name db.name and password pass in mode m (0 for read
! and 2 for read/write); if successful a pointer to the root object is returned otherwise an
! error.record

let garbage.collect := **proc**(**cstring** name -> **pntr**)
! this is not currently defined

let Garbage.collect := **proc**(-> **pntr**)
! this is not currently defined

let structure.table := **proc**(-> ***cstring**)
! this returns a vector of strings containing all of the class identifiers used by the program

let class.identifier := **proc**(**pntr** p -> **string**)
! this returns the class identifier for the structure pointed at by p

let fiddle.r := **proc**(**real** n -> *** int**)
! splits a real into a vector of two integers

let exec := **proc**(***string** s)
! Unix exec - calls the shell using the command held in s, if successful it will not return

let fork := **proc**(-> **int**)
! Unix process fork - returns process number of the child to the parent, returns 0 to the
! child

let wait := **proc**(**int** i -> ***cint**)
! Unix wait - waits for a signal from a child process or for process i to die, returns a

! process id and status value in a vector

let system := **proc**(**string** s -> **int**)
! Unix system - does a fork and then execs a shell to execute the string s

let Simple.menu := **proc**(**string** title ; ***string** entries ; ***proc**(**string**,**int**) actions -> **proc**())
! this returns a procedure that will display the title and entries specified and allow the user
! to select one, associated with each entry is a corresponding action procedure, this
! procedure is passed the associated entry and entry number when its entry is selected

let pail.eval := **proc**(**pntr** tree)
! this is not currently implemented

let openrf := **proc**(**cstring** f ; **cint** m -> **file**)
! this opens the resource fork of a Macintosh file using the same parameters as open

let settyp := **proc**(**cstring** f ; **cint** c,t)
! this sets the file type to t and the creator to c for the Macintosh file f

let Environment := **proc**(-> **pntr**)
! this is not currently defined

let post.mortem := **proc**(-> **int**)
! this returns the line number at which the last event handler was automatically invoked

let line.end := **proc**(**#pixel** i ; **pixel** p ; **cint** x,y,direct -> **int**)
! searches for the first pixel of colour p in image i starting at position x,y, direct specifies
! the search direction, odd numbers do not look at boundary pixels, 0,1 - left ; 2,3 - right ;
! 4,5 - down; 6,7 - up, it returns the position of the pixel or 1 past the last position
! searched.

let set.locator := **proc**(**int** mode,tabs,ticks,flags)
! sets the conditions when a PERQ generates mouse input events

let set.cursor := **proc**(**cint** mode)
! sets the mode in which the cursor operates, the mode varies between implementations,
! cursor.on and cursor.off should be used instead

let plane.of := **proc**(**c#pixel** i ; **cint** p -> ***int**)
! this procedure returns the integer vector holding the pth plane of i

let pnx.line := **proc**(**c#pixel** i ; **cint** x1,y1,x2,y2,style)
! draws a line from the point x1,y1 to the point x2,y2 on the image i, the last point is not
! drawn; the style can be 0 - draw: set pixels to **on**, 1 - erase: set pixels to **off**, 2 - xor:
! invert pixels.

let coerce.proc := **proc**(**proc**() a.proc ; **cpntr** result ; **cint** gl.frame.ms,gl.frame.ps -> **pntr**)
! this procedure takes a void procedure and a structure containing a single procedure field
! and assigns the procedure to the structure's field, it also creates a frame copying
! gl.frame.ms items from the global main stack to the new frame and gl.frame.ps items

! from the global pointer stack,the new frame is made the static link of the structure's
! procedure field

let make.proc := **proc**(**cint** max.ms,max.ps ; **cbool** p.vec,s.vec ; ***proc**() proc.vec
***string** string.vec ; **cint** start,size ; ***int** codeword -> **proc**())
! this constructs a void procedure from the supplied parts, max.ms and max.ps are the
! stack sizes the procedure requires, p.vec and s.vec indicate if the vectors proc.vec and
! string.vec should be used, start and size describe which part of codeword contains the
! PS-code for the procedure

let compiler := **proc**(**cstring** file.name ; **cpntr** proc.holder -> **pntr**)
! compiles the procedure found in the file file.name and type checks it against the single
! procedure field of proc.holder, if successful a structure is returned containing the
! compiled procedure, it will be the same structure class as proc.holder, if the compilation
! fails a linked list of error messages will be returned

let wait.for := **proc**(**bool** move,press,input)
! on a PERQ this function will wait for one of the specified events to occur, move waits for
! the mouse to move, press waits for a mouse button to be pressed and input waits until
! input.pending would return **true**

let input.pending := **proc**(-> **bool**)
! returns **true** if a **read** or **peek** would complete without blocking

let pixel.depth := **proc**(**cpixel** i -> **int**)
! returns the number of planes in pixel i

12.2 Standard Identifiers

A number of standard identifiers exist in the language. They are

r.w	variable initially 14
s.w	variable initially 2
i.w	variable initially 12
maxint	constant, the maximum integer
epsilon	constant, the largest real e such that 1 + e = 1
pi	constant, pi
maxreal	constant, the largest real
screen	constant, image representing the screen
cursor	constant, image representing the cursor
error.records	variable, **c*c*cpntr** the array of error record structures
events	pointer to structure holding the event handlers
enhanced.screen	variable, default print destination
s.o	variable initially the file descriptor for standard output
s.i	variable initially the file descriptor for standard input

Note that the minimum integer value is -maxint - 1 and the minimum real value is -maxreal.

Appendix I

Context Free Syntax

```
<program>            ::= <sequence>?


<sequence>           ::= <declaration>[;<sequence>] | <clause>[;<sequence>]


<declaration>        ::= <let_decl> | <structure_decl> | <handler>

<let_decl>           ::= let<identifier><init_op><clause>

<init_op>            ::= = | :=

<structure_decl>     ::= structure<identifier>[(<field_list>)]

<field_list>         ::= <type1><identifier_list>[;<field_list>]

<identifier_list>    ::= <identifier>[,<identifier_list>]

<handler>            ::= when<ex_id_list>[as<identifier>]do<clause>

<ex_id_list>         ::= any | <identifier_list>


<identifier>         ::= <letter>[<letter> | <digit> | .]*

<letter>             ::= a | b | c | d | e | f | g | h | i | j | k | l | m |

                         n | o | p | q | r | s | t | u | v | w | x | y | z |

                         A | B | C | D | E | F | G | H | I | J | K | L | M |

                         N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<digit>              ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9


<type1>              ::= [c]<type>

<type>               ::= int | real | bool | string | pntr | file | pixel | pic |

                         <star><type1> | #pixel  | #cpixel  | <proc_type>

<proc_type>          ::= proc([<proc_param_list>][<arrow><type>])

<proc_param_list>    ::= <proc_para_type>[,<proc_param_list>]
```

<proc_para_type> ::= <type1> | <structure_decl>

<clause> ::= **if**<clause>**do**<clause> | **if**<clause>**then**<clause>**else**<clause>
|
repeat<clause>**while**<clause>[**do**<clause>] |
while<clause>**do**<clause> |
for<identifier>=<clause>**to**<clause>[**by**<clause>]**do**<clause>
|
raise<identifier>[(<clause_list>)] |
case<clause>**of**<case_list>**default** :<clause> |
<raster> | <print> | <write> |
<name>:=<clause> | **abort** | <expression>

<clause_list> ::= <clause>[,<clause_list>]
<case_list> ::= <clause_list>:<clause>;[<case_list>]
<raster> ::= <raster_op><clause>**onto**<clause>
<raster_op> ::= **ror** | **rand** | **xor** | **copy** | **nand** | **nor** | **not** | **xnor**

<print> ::= **print**<print_list>[**onto**<clause>][**at**<clause>,<clause>]
[**in**<clause>][**using**<raster_op>]
<print_list> ::= <clause>[:<clause>][,<print_list>]

<write> ::= **write**<write_list> | **output**<clause>,<write_list> |
out.byte<clause>,<clause>,<clause>
<write_list> ::= <clause>[:<clause>][,<write_list>]
<name> ::= <identifier> | <expression>[([<clause_list>])]*

<expression>	::= <exp1>[**or**<exp1>]*
<exp1>	::= <exp2>[**and**<exp2>]*
<exp2>	::= [~]<exp3>[<rel_op><exp3>]
<exp3>	::= <exp4>[<add_op><exp4>]*
<exp4>	::= [<add_op>]<exp5>[<mult_op><exp5>]*

<exp5> ::= <standard_exp> | <literal> | (<clause>) |
begin[<sequence>]**end** | {[<sequence>]} | <picture_exp>
<name>[(<clause><bar><clause>)]* | <compound_cons>

<standard_exp> ::= <bound_proc>(<clause>) | <read_proc>([<clause>]) |
read.name(<clause>[,<clause>])

<bound_proc> ::= **lwb** | **upb**

<read_proc> ::= **read** | **peek** | **readi** | **readb** | **reads** | **readr** | **eoi**
read.byte | **read.a.line**

<literal> ::= <int_literal> | <real_literal> | <bool_literal> | <string_literal>
|
<pixel_literal> | <pntr_literal> | <file_literal> | <proc_literal>

<int_literal> ::= <digit>[<int_literal>]

<real_literal> ::= <int_literal>.[<int_literal>][e<scale_factor>]

<scale_factor> ::= [<add_op>]<int_literal>

<bool_literal> ::= **true** | **false**

<string_literal> ::= <double_quote>[<char>]*<double_quote>

<char> ::= <other> | <special_character>

<other> ::= any ascii character except ' and "

<special_character> ::= <single_quote>[<special_follow>]

<special_follow> ::= n | p | o | t | s | <single_quote> | <double_quote>

<pixel_literal> ::= **on**[&<pixel_literal>] | **off**[&<pixel_literal>]

<pntr_literal> ::= **nil**

<file_literal> ::= **nullfile**

<proc_literal> ::=
proc([<named_param_list>][<arrow><type>]);<proc_clause>

```
<named_param_list>    ::= <proc_param_type>[;<named_param_list>]

<proc_param_type>     ::= <type1><identifier_list> | <structure_decl>
```

<proc_clause> ::= <clause> | **nullproc**

<picture_exp> ::= **shift**<clause>**by**<clause>,<clause> |
scale<clause>**by**<clause>,<clause> |
rotate<clause>**by**<clause> |
colour<clause>**in**<clause> |
text<clause>**from**<clause>,<clause>**to**<clause>,<clause> |
<lsb><clause>,<clause><rsb>

<compound_cons> ::= <vector_constructor> | <image_constructor>
<vector_constructor> ::= **vector**<bounds>**of**<clause> |
@<clause>**of**<type1><lsb><clause_list><rsb>
<bounds> ::= <clause>::<clause>[,<bounds>]
<image_constructor> ::= **image**<clause>**by**<clause>**of**<clause> |
limit<clause>[**to**<clause>**by**<clause>][**at**<clause>,<clause>]

<lsb> ::= [
<rsb> ::=]
<star> ::= *
<bar> ::= |
<add_op> ::= + | -
<mult_op> ::= <star> | / | **div** | **rem** | ++ | ^ | &
<rel_op> ::= **is** | **isnt** | < | ≤ | > | ≥ | = | ≠
<arrow> ::= ->
<single_quote> ::= '
<double_quote> ::= "

Appendix II

Type Matching Rules

type arith **is** int | real

type comparable **is** arith | string

type printable **is** comparable | bool | pixel

type literal **is** printable | pntr | proc | file

type image **is** #pixel | #cpixel

type nonvoid **is** literal | image | *nonvoid

type type **is** nonvoid | void

<sequence> : void ? **=>** void

t : type, <declaration> : void ; <sequence> : t **=>** t

t : type, <clause> : void ; <sequence> : t **=>** t

<declaration> **=>** void

where <let_decl> ::= **let**<identifier> <init_op> <clause> : nonvoid

where <handler> ::= **when** <ex_id_list>[**as** <identifier>]**do**<clause> : type

if <clause> : bool **do** <clause> : void => void

t : type, **if** <clause> : bool **then** <clause> : t **else** <clause> : t => t

repeat <clause> : void **while** <clause> : bool

[**do** <clause> : void] => void

while <clause> : bool **do** <clause> : void => void

for <identifier>=<clause> : int **to** <clause> : int

[**by**<clause> : int]**do**<clause> : void => void

raise<clause> : pntr => void

t : type ; t1 : nonvoid, **case** <clause> : t1 **of** <case_list>

default : <clause> : t => t

where <case_list>   ::= <clause_list>:<clause> : t ; [<case_list>]

where <clause_list>   ::= <clause> : t1 [,<clause_list>]

<raster> => void

where <raster> ::= <raster_op><clause> : image **onto** <clause> : #pixel

<print> => void

where <print> ::= **print**<print_list>[**onto**<clause> : pntr]

[**at**<clause> : int,<clause> : int][**in**<clause> : pntr][**using**<raster_op>]

where <print_list> ::= <clause> : nonvoid [:<clause> : int][,<print_list>]

<write> => void

where <write_list> ::= <clause> : printable [:<clause> : int][,<write_list>]

where **output**<clause> : file ,<write_list>

where **out.byte**<clause> : int,<clause> : int,<clause> : int

t : nonvoid, <name> : t := <clause> : t => void

where<name> ::= <identifier> : **t** | <expression> : **t** [([<clause_list>])]

where <clause_list> ::= <clause> : **nonvoid** [,<clause_list>]

abort => void

<expression> : **bool** **or** <expression> : **bool => bool**

<expression> : **bool** **and** <expression> : **bool => bool**

[~]<expression> : **bool => bool**

t : **nonvoid**, <expression> : **t** <eq.op> <expression> : **t => bool**

where <eq_op> ::= = | ≠

t : **comparable**, <expression> : **t** <co_op> <expression> : **t => bool**

where <co_op> ::= < | ≤ | > | ≥

<expression> : **pntr** <type_op><expression> : **structure**

where <type_op> ::= **is** | **isnt**

t : **arith**, <expression> : **t** <add_op><expression> : **t => t**

t : **arith**, <add_op> <expression> : **t => t**

<expression> : **int** <int_mult_op> <expression> : **int => int**

where <int_mult_op> ::= <star> | **div** | **rem**

<expression> : **real** <real_mult_op> <expression> : **real => real**

where <real_mult_op> ::= <star> | /

<expression> : **string** ++ <expression> : **string => string**

<expression> : **pic** <pic_op><expression> : **pic => pic**

where <pic_op> ::= ^ | &

<expression> : **pixel** & <expression> : **pixel => pixel**

t : type, <standard_exp> : t => t

where <bound_proc>(<clause> : file) => int

where <bound_proc> ::= **upb** | **lwb**

where<int_read>([<clause> : file]) => int

where<int_read> ::= **readi** | **read.byte**

where<string_read>([<clause> : file]) => string

where<string_read> ::= **read** | **peek** | **reads** | **read.a.line**

where **read.name**(<clause> : string [,<clause> : file]) => string

where<bool_read>([<clause> : file]) => bool

where<bool_read> ::= **readb** | **eoi**

where **readr**([<clause> : file]) => real

t : literal, <literal> : t => t

where<digit>[<int_literal>] => int

<int_literal>.[<int_literal>][e<scale_factor>] => real

true | **false** => bool

<double_quote><char><double_quote> => string

on[&<pixel_literal>] | **off**[&<pixel_literal>] => pixel

<pntr_literal> => null

t : type, **proc**([<named_param_list>][<arrow><type> : t]);<clause> : t t : nonvoid, (<clause> : t) => t

t : type, **begin** <sequence> : t **end** => t

t : type, { <sequence> : t } => t

<picture_op> => pic

where **shift**<clause> : pic **by**<clause> : real ,<clause> : real => pic

scale<clause> : pic **by**<clause> : real ,<clause> : real => pic

rotate<clause> : pic **by**<clause> : real => pic

colour<clause> : pic **in**<clause> : pixel => pic

text<clause> : string **from**<clause> : real ,<clause> : real

to<clause> : real ,<clause> : real => pic

<lsb><clause> : real ,<clause> : real <rsb> => pic

t : type ; t1 : nonvoid, <name> : t => t

where <name> : string (<clause> : int<bar> <clause> : int) => string

where <name> : image (<clause> : int<bar> <clause> : int) => image

where <name> : pixel (<clause> : int <bar> <clause> : int) => pixel

```
where <name> ::=     <identifier> : t1  |
                     <expression> : t1  (<clause_list>)[(<clause_list>)]*
where <clause_list>  ::= <clause> : t1  [,<clause_list>]
```

t : nonvoid, @<clause> : int **of** <type1> : t<lsb><clause_list><rsb> => *t

```
where <clause_list> ::= <clause> : t [,<clause_list>]

t : nonvoid, vector <bounds> of <clause> : t

where <bounds> ::= <clause> : int :: <clause> : int [,<bounds>]

image <clause> : int by<clause> : int of <clause> : pixel => #pixel

limit<clause> : #pixel[to<clause> : int][by<clause> : int]

                         [at<clause> : int ,<clause> : int] => #pixel
```

Appendix III

Program Layout

Semi-Colons

As a lexical rule in PS-algol, a semi-colon may be omitted whenever it is used as a separator and it coincides with a newline. This allows many of the annoying semi-colons in a program to be left out. However, to help the compiler deduce where the semi-colons should be, it is a rule that a line may not begin with a binary operator. For example

```
a *
b
```

is valid but

```
a
* b
```

is not.

This rule also applies to the invisible operator between a vector and its index list. For example

```
let b = a( 1,2 )
```

is valid but

```
let b = a
        ( 1,2 )
```

will be misinterpreted since vectors can be assigned.

Comments

Comments may be placed in a program by using the symbol '!'. Anything between the ! and the end of the line is regarded by the compiler as a comment. For example

```
a + b  ! add a and b
```

Directives

There are certain compiler directives used to annotate the listing of the program provided by the compiler that the user may wish to invoke. The symbol '%' is used to denote a directive. They are

%list	print the program listing.
%nolist	do not print the program listing.
%title,< string literal >	take a new page and use the string as a heading for this and subsequent pages.
%lines,< integer literal >	Inform the compiler of the number of lines on each page of output paper.
%ul	underline the reserved words in the listing.
%noul	turn off the reserved word underlining.

Appendix IV

PS-algol Prelude

```
structure error.record( cstring error.context,error.fault,error.explain )
! error.fault is the cause of the error. error.explain is a more detailed version of
error.fault.

structure opdb.result( pntr root.of.db )
        ! used by the interpreter

structure culr.strc( cpntr nxtre ; cstring shade )
        ! used by the compiler

structure oprtn.strc( cpntr lft,rght ; cint opoo )
        ! used by the compiler

structure scrbl.strc( cstring msge ; creal xxl,yyl,xxr,yyr )
        ! used by the compiler

structure poin.strc( creal pnx,pny )                    ! used by the compiler

structure trnsfrm.strc( cint trnsfrm ; cpntr mrtre ; creal trnsfrm.x,trnsfrm.y )
! used by the compiler

structure mouse( cint X.pos,Y.pos ; cbool selected ; c*cbool the.buttons )
! structure returned by locator().
! X.pos and Y.pos indicate the position of the mouse relative to screen.
! selected indicates if the screen is currently selected for input.
! the.buttons indicate which ( if any ) of the mouse buttons are down.

structure font( cint font.height,descender ; c*#cpixel the.chars ; cstring info )
! structure used to store fonts

structure point.strc( cint point.x,point.y ) ! returned by cursor.tip

let nilpic = text "" from float( 0 ),float( 0 ) to float( 0 ), float( 0 )

structure events.strc(
        cproc( cstring )            ! Error.event, Ferror.event ;
        proc( cpntr,cint )          ! Uncaught.event ; TooManyEvents.event,
                                    ! Interrupt.event, WindowChange.event ;
        cproc()                         ! Arithmetic.event ;
        proc( cstring -> int )          ! ReadI.event ;
        proc( cstring -> bool )     ! ReadB.event ;
        proc( cstring -> string )       ! ReadS.event ;
        proc( cstring -> real )          ! ReadR.event ;
        proc( cstring,cstring -> pntr )       ! OpenDatabase.event ;
        cproc()                               ! ApplyingNullproc.event ;
        cproc( cpntr,cstring )           ! StructureAccess.event ;
        cproc( cint,cint,cint )               ! VectorBounds.event ;
        cproc( cint,cint )               ! IliffeBounds.event ;
        proc( cpixel,cint -> pixel )          ! PixelBounds.event ;
        proc( c#pixel,cint,cint -> #pixel )  ! SubImage.event, Limit1.event ;
        proc( c#pixel,cint,cint,cint,cint -> #pixel )       ! Limit2.event ;
        proc( cstring,cint,cint -> string )                 ! SubString.event ;
        proc( cstring,cstring -> string )                   ! StringTooLong.event ;
        proc( cstring -> string )                           ! Decode.event ;
        proc( cpntr -> pntr )                               ! CursorTip.event ;
        cproc( cstring )                              ! Menu.event, StringToTile.event
        )                                             ! used by the global pntr events.
! the following structure classes are used by event procedures to raise exceptions.

structure TooManyEvents

structure Interrupt

structure WindowChange

structure Arithmetic

structure ReadI( cstring ReadI.char )

structure ReadB( cstring ReadB.char )

structure ReadS( cstring ReadS.char )

structure ReadR( cstring ReadR.char )

structure OpenDatabase( cstring OpenDatabase.name,OpenDatabase.explain )

structure ApplyingNullproc

structure StructureAccess( cpntr StructureAccess.pntr ; cstring
StructureAccess.class )

structure VectorBounds(cint
VectorBounds.index,VectorBounds.lwb,VectorBounds.upb)

structure IliffeBounds( cint IliffeBounds.lwb,IliffeBounds.upb )

structure PixelBounds( cpixel PixelBounds.pixel ; cint PixelBounds.index )
```

structure SubImage(**c#pixel** SubImage.image ; **cint** SubImage.start,SubImage.length)

structure Limit1(**c#pixel** Limit1.image ; **cint** Limit1.Xoffset,Limit1.Yoffset)

structure Limit2(**c#pixel** Limit2.image ; **cint** Limit2.Xoffset,
Limit2.Yoffset,Limit2.Xdim,Limit2.Ydim)

structure SubString(**cstring** SubString.string ; **cint** SubString.start,SubString.length)

structure StringTooLong(**cstring** StringTooLong.left,StringTooLong.right)

structure Decode(**cstring** Decode.string)

structure CursorTip(**cpntr** CursorTip.point)

structure Menu(**cstring** Menu.reason)

structure StringToTile(**cstring** StringToTile.reason)

Appendix V

System Event and System Exception Interpretation

This appendix describes the event procedures that are provided by the PS-algol system, along with the system exceptions that they may raise. Many of the default event procedures have been placed in variable fields of the events structure. This allows the programmer to replace a default procedure with a customised one. A number of the event procedures return values. This feature allows the programmer to provide an alternative value for a failed operation. Note that although every program has a constant called 'events' globally available to it as standard, it should be thought of as being declared locally to that program and will be initialized with a different instance of the 'events.strc' structure.

events(Error.event), events(Ferror.event)
These two procedures correspond to the old error reporting interface. They print their string argument and **abort** the program. They cannot be replaced by the programmer.

events(Uncaught.event)
If an exception is allowed to propagate all the way back to the top level of the program, without being intercepted by a handler, this event procedure is invoked. It is parameterized by the excepted value and the line number at which the exception was raised. If the exception is recognized as a system exception, the default procedure will print out the name of the exception and the values in the fields of the excepted value; it will explain the significance of these values as it does so. If the exception is not recognized as a system exception, it will print the class identifier of the excepted value instead. The line number at which the exception arose is printed. If the exception had been raised by an event procedure then the line number at which that event occurred is also given. To return from this procedure following the 'Uncaught.event' event, is to **abort** the program.

events(TooManyEvents.event)
This is initiated should an event overflow occur, that is, a second event occurred before the event procedure for the first had been called successfully. The default action is to raise the system-defined exception 'TooManyEvents'.

events(Interrupt.event)
Invoked on receipt of a keyboard interrupt (usually caused by typing Control-C). To be compatible with previous versions of PS-algol, the default action is to do nothing. Programs can still interrogate the standard boolean interrupt procedure to determine whether an interrupt has occurred. By replacing the default, a program could arrange to take some action, perhaps to interact with a user, and then resume from where it left off.

events(WindowChange.event)

(A misnomer for a screen change event.) Called following an attempt by the user to change the size of the screen to which the program is attached. This is only raised in the Perq-pnx system. Other PS-algol systems resize the screen without corrupting it.

events(Arithmetic.event)
All hardware-detected arithmetic errors such as floating-point or integer overflow, divide by zero etc. map onto this event. The default procedure, which cannot be overwritten, will terminate by raising the exception Arithmetic.

events(ReadI.event), events(ReadB.event),events(ReadS.event),events(ReadR.event)
Each of these correspond to failures in a standard read procedure. They can all be replaced by the programmer with a more appropriate response. The defaults will raise the corresponding system exception, and this will carry the offending character along with it. The offending

character (for example a letter where a digit was expected) is made available to the event procedure as the string parameter. If the event procedure is allowed to return normally, it will provide a replacement value for the failed operation.

events(OpenDatabase.event)
If a call to 'open.database' fails, it will return the **pntr** yielded by this event procedure. By default, and to be compatible with previous versions of PS-algol, the event procedure returns an 'error.record'. A user program could use the event procedure to return a more useful value and to take some meaningful alternative action. This does away with the need to slavishly check the result of an 'open.database'. The string parameters are the name of the database, and an explanation for the failure, respectively. A corresponding structure for use as an exception, is available from the standard environment.

events(ApplyingNullproc.event)
Called when a program has attempted to apply a procedure whose body has not been defined. The constant default action is to terminate by raising the exception 'ApplyingNullproc'.

events(StructureAccess.event)
Called by the runtime type checking system when a program acts on a false assumption about the nature of a **pntr** object by attempting to dereference a field. The event procedure is passed the actual **pntr** value together with the expected class identifier of the object. The default action, which cannot be changed, is to raise a 'StructureAccess' exception.

events(VectorBounds.event)
This is another event procedure that cannot be replaced. It is due to an attempt to read or write outside the bounds of a vector. The offending index and the actual lower and upper bounds are its parameters, and these appear with the exception that the event procedure raises.

events(IliffeBounds.event)
supplied to a vector creation expression. For example

```
let i = 3
let v = vector i :: i - 1 of "upb < lwb not possible"
```

This event procedure raises the exception 'IliffeBounds', which is provided with the offending lower and upper bounds. It lives in a constant field.

events(PixelBounds.event)
This procedure can be replaced by a user program. It is called when a program attempts to extract a non-existent plane from a pixel. The pixel itself, and the depth level that had been requested, are available as its parameters. A programmer-supplied event procedure could provide an alternative value for the

failed indexing operation; the default action is to raise the 'PixelBounds' exception.

events(SubImage.event)
This event occurs when an attempt is made to alias a non-existent range of planes in an image. The offending starting plane, and the depth of the requested alias are the two integer parameters. The image parameter is the image on which the operation was being attempted. An alternative result can be supplied to replace the failed expression when the procedure is allowed to return normally. By default the exception 'SubImage' is raised, but the procedure can be replaced by a user program.

events(Limit1.event), events(Limit2.event)

These are caused by illegal limit operations. Both are supplied with the image on which the **limit** was being attempted. 'Limit1.event' is provided with the offending x and y offsets. The x and y are also the first two integer parameters of 'Limit2.event', in addition the requested dimensions are made available. By default, the corresponding terminating exceptions 'Limit1' and 'Limit2' are raised, and these carry the procedure arguments with them as part of the excepted value. A user program might decide instead to resume processing by supplying an alternative image value as the result of the failed expression.

events(SubString.event)
Caused by acting on the false assumption that a string is longer than it really is. The string being operated on, the start position in the original, and the length of the requested string, are its parameters. The default, but replaceable, action is to raise the system exception 'SubString'. When the event procedure is allowed to return, an alternative string, to replace the failed string expression, must be provided.

events(StringTooLong.event)
There is an implementation limit on the size of a string which if we try to exceed this event procedure is activated. The participating strings are received as parameters. These are propagated with the exception 'StringTooLong' in the default procedure. A user program might instead supply a procedure that displays a warning before providing an alternative result to the concatenation.

events(Decode.event)
This procedure is activated following an attempt to decode the empty string. By default the exception 'Decode' is raised, thus terminating the procedure. If allowed to resume, its string value is passed to the decode operation.

events(CursorTip.event)
There are two circumstances when a call of the standard graphics function 'cursor.tip' could initiate the 'CursorTip.event' procedure. 'Cursor.tip' expects a pair of integers wrapped in a structure, or if the otherwise valid coordinate pair is outside the dimensions of cursor. The parameter to the event procedure is the **pntr** which had been passed to cursor.tip. A user program can replace the default, which otherwise raises the exception 'CursorTip'.

events(Menu.event)
This event procedure is in a constant field of the events structure and so cannot be replaced. It is called following an unrecoverable error in the use of the menu function 'menu' may have been called with the actions and events vectors having mismatched bounds, or screen may be too small to accommodate the menu. The event procedure terminates by raising the exception 'Menu'. The string in the excepted value structure (and argument to the event procedure itself) is a description of the problem.

events(StringToTile.event)
There are three possible causes of the event that invokes this procedure, an empty string passed to the 'string.to.tile' function, a failure by 'string.to.tile' to open the font database, or failure to find the specified font. It is not permitted to resume this event and so the default (and non-replaceable) event procedure terminates with the exception 'StringToTile'. The string associated with this exception describes the reason for the failure.

Appendix VI

Printing on Images and Fonts

The **print** clause introduced in 7.4 is here further defined. For convenience the syntax and type rules are reiterated.

```
<print>          ::=
print<print_list>[onto<clause>][at<clause>,<clause>]
                        [in<clause>][using<raster_op>]
<print_list>  ::=   <clause>[:<clause>][,<print_list>]

<print_list>[onto<clause> : pntr]
        [at<clause> : int,<clause> : int][in<clause> : pntr]
        [using<raster_op>] => void
where  <print_list>  ::=  <clause>  :  nonvoid[:<clause>  :
int][,<print_list>]
```

The value which follows **onto** is a **pntr** to an instance of the following class.

```
structure print.destination (
    int X.at, Y.at ;                            ! position for next tile
    proc ( string, pntr, pntr, pntr, int, int  -> bool ) Edge.violation;
    pntr Font ;                                          ! the last font set/used
    proc ( c#pixel, c#pixel ) Raster .op ;               ! pixel combination rule
    cpntr Final.destination ;                            ! place where titles are put
    proc ( cpntr, cpntr  ) PRINT )
```

A declaration of this structure (among others) may be copied from the file '/usr/lib/ps/decls' on most machines running PS-algol under UNIX. An instance of the structure is generated by a call of 'enhance.image', when the image presented as a parameter will become that referenced by 'Final.destination', and will be the image that collects the interpreted tiles. An initial state for the destination will be established as described below. A user may construct a 'print.destination' instance in some other way (e.g. to control a plotter) provided that the construction exhibits the correct behaviour when used by **print**.

The fields of 'print.destination' are interpreted as follows:

X.at, Y.at	The x and y coordinates (in the 'Final.destination' image) at which the origin of the next tile should be placed. If these are not within the target an edge event will be raised when the next tile is positioned. After a tile has

been placed 'X.at' is incremented by the width of the tile. When the instance is constructed by 'enhance.image' they are initialised to the top left hand corner of the target. They are explicitly reset by the **at** clause.

Edge.violation	The edge violation event occurs if any part of a tile to be placed would lie outside the 'Final.destination' in which case this procedure in the current 'print.destination' is called. It may solve the problem by adjusting 'X.at', 'Y.at' or the tile to be placed, or may take some other action. If it requires the underlying system to retry then it returns **true**. When **false** is returned, clipping is used to include part of the tile. When set up by 'enhance.image' this field has a handler which simulates character placement on a simple character driven terminal, i.e. it inserts newline (decrements Y.at by the font height

	and sets 'X.at' to zero) on right edge violation, and scrolls up one line on bottom edge violation. Other handlers are often available in an installation's library, e.g. one which clips.
Font	This holds a pointer to the font currently in use (the data structure for fonts is described below). It will be used to convert characters to image tiles. The **in** clause will cause it to be reset. It is initially set to the font with key "default.font", normally "fix13".
Rasterop	This specifies how the tile and the target are to be combined, and defaults to a procedural encapsulation of **copy**. It is reset by the **using** clause.
Final.destination	This specifies a destination for the output operations. After a call of 'enhance.image' it refers to a container instance of **structure** cimage.box (**c#pixel** the.cimage) holding the image that was enhanced.
PRINT	This is a procedure called by the run time system once per **print** clause. The arguments in order are: the head of a list of **structure** cons. (**pntr** hd., tl.) instances corresponding to the print_list; and the 'print.destination' .

The 'hd.' fields of the first list refer to containers of the form
structure box (<type> content)
or one of the following
structure string.box.(**string** the.string.)
structure int.box.(**int** the.int.)
structure bool.box.(**bool** the.bool.)
structure real.box.(**real** the.real.)
structure pntr.box.(**pntr** the.pntr.)

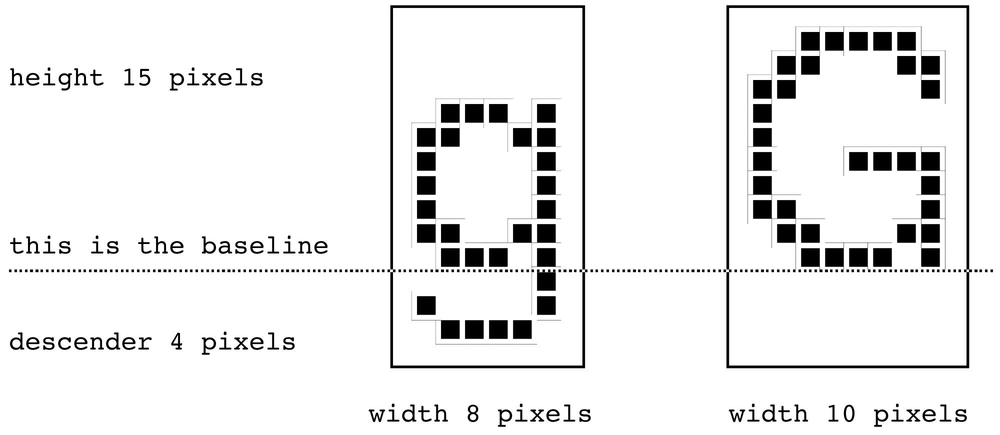
If a width clause has occurred then the 'hd.' field will refer to another 'cons.' cell the 'hd.' field of which will be a container, as described above, and the 'tl.' field will refer to an 'int.box.' holding the width. The compiler plants code to construct the appropriate data structure, optimisations should prevent its repeated reallocation.

The referend of the 'Font' field of a 'print.destination' must conform to the PS-algol representation of fonts. These are also used as arguments to some other procedures, and may be constructed and manipulated by PS-algol programs. New data structures may be introduced provided that the last 5 fields of a 'print.destination' are mutually consistent.

A number of different fonts are held in a database called 'FONTS'. This database should have the password 'friend' so that programmers may examine its contents. The database contains a table from font names to font structures which are defined as follows.

structure font (**cint** font.height,descender ; **c*#cpixel** the.chars ; **cstring** info)

The 'font.height' is the height of the characters in the font. The 'descender' distance from the bottom of an image holding a character to the base line. The widths of characters in a font may vary but the programmer may examine these by taking the x dimension of the appropriate image. For example



The vector called 'the.chars' contains font images, so they may be indexed by the ascii character code. Information about the font may be found in the string 'info' .

When using **print** the intersection of the base line with the left hand edge is placed at the current position (X.at, Y.at) and the new currrent position is then at the intersection of that line with the right hand side. If an image occurs in a print_list it is treated as if the descender were zero. Note the 'font.height' field of the current font determines the decrement of 'Y.at' on new line (or right hand event violation) and the movement on scrolling.

The parameters of an 'Edge.violation' are:

string	A string holding the character to be printed, or the empty string if the current element of print_list is not interpreted via characters.
pntr	An 'edge.strc' structure instance.

pntr	A pointer to the current font in use.
pntr	A pointer to the current 'print.destination'.
int	The current value of 'X.at'.
int	The current value of 'Y.at'.

where the 'edge.strc' is declared by

> **structure** edge.strc(**cbool** top, bottom, left, right)

any field of this being **true** indicates that the tile being placed overlaps at least that edge.

INDEX

! (comments)	51
" (string quotes)	7,8
"" (empty string)	8
#**cpixel**	9,22
#**pixel**	5,9,22
% (directives)	51
%lines	51
%list	51
%nolist	51
%noul	51
%title	51
%ul	51
& (picture include)	14
& (pictures)	13
& (pixel operator)	8
& (pixels)	14
' (escape character)	8
'"	8
'b	8
'n	8
'o	8
'p	8
't	8
(	12
)	12
* (multiplication)	11
* (syntactic repetition)	4
* (vector type)	5
*T	5
+ (add_op)	7,11
++	12
, (exception handlers)	27
, (images)	20
, (output)	30
, (pictures)	13
, (print)	22
, (vector bounds)	18
- (add_op)	7,11
->	9
. (decimal point)	7
. (in identifier)	16
: (output)	30
: (print)	22
:: (vector bounds)	18,19
:= (assignment)	25
:= (variable declaration)	16

;	51
;	9
; (sequences of clauses)	17
< (less than)	11
< (syntactic category names)	4
<=	11
= (equality)	11
= (constant declaration)	16
=> (resultant type)	6
> (greater than)	11
> (syntactic category names)	4
>=	11
@ (arrays)	18
[(arrays)	18
[(pictures)	14
[(syntactic grouping)	4
] (arrays)	18
] (pictures)	14
] (syntactic grouping)	4
^ (picture join)	13,14
{ (**begin**)	17
| (images)	22
| (pixels)	15
| (substrings)	13
| (syntactic choice)	4
} (**end**)	17
~=	11

A

abort	27,54
abs	33
absolute value	33
abstraction	24
actual parameter	24
addition	12
<add_op>	11,44
and — see also rand	
and	10
any (exception handlers)	27
ApplyingNullproc	53
ApplyingNullproc.event	54
arc tangent	33
`arith` type	5
Arithmetic	53

arithmetic expressions	11
Arithmetic.event	54
arrays — see also vectors	
arrays	18
arrow (->)	9,44
as (exception handlers)	27
ASCII	7,13
assignment clause	25
assignment incompatibility	22
assignment of pointers	23
association via tables	31
at (images)	20
at (print)	22
atan	33
automatic clipping	21

B

'b	8
b.and	37
b.not	38
b.or	37
backspace	8
Backus-Naur form	4
bar (strings)	13
bar (images)	22
bar (pixels)	15
bar |	44
begin (clause brackets)	17
bitmaps — see images	
block expressions	17
bool	9
Boolean expressions	10
Boolean literals	7
boolean type	5
<bool_literal>	7,43
<bounds>	18,44
bounds checking	54
<bounds_op>	19
bound_proc	43
brackets	12,17
button depression	22
by (images)	20
by (pictures)	13
by(for clause)	26

C

c (type constancy)	9
call by value	24
call of procedure	24
carriage return	8
case clause	25,26
case significance	16
<case_list>	25,42
change.owner	35
change.passwd	35
changing windows	54
<char>	8,43
character ordering	13
character sequences	7,8
civil rights	18
class.identifier	38
<clause>	42
clauses — see also expressions	
clauses	5,25
<clause_list>	42
clipping	21
close	30,37
closures	24
code	33
coerce.proc	39
coercions	12
colour (pictures)	13,14
colour.map	37
colour.of	37
comments	51
commit	32,35
comparable type	5
comparison	11
comparison (strings)	13
compiler	39
completeness	18
compound data objects	18
compound_cons	44
concatenation (++)	12
conditional clauses	25
constant.image	36
constants	16
context free syntax	41

control-c	54
conversion,ASCII to integer	33
conversion,integer to ASCII	33
conversion nmbrs to strings	33,34
copy (images)	21
correspondence, principle of	24
cos	33
co_op	11
#**cpixel**	9
creatdb	38
create	30,37
create.database	32,35
culr.strc	52
cursor	22,40
cursor.off	36
cursor.on	36
cursor.tip	36
CursorTip	53
CursorTip.event	55
CursorTip.point	53
customised event handlers	54

D

data types	5
database procedures	32
database, persistent	31
database conventions	32
date	34
decimal point	7
decimal scale factors	7
<declaration>	41
declaration before use	24
declarations of data objects	16
decode	33
Decode	53
Decode.event	55
Decode.string	53
default (case clause)	25
depth	36
depth (of images)	22,36
descender	52,58
device dependency	8
device.draw	36
digit	34,41

directives 51
div 11
divide by zero 54
division 12
do (conditional) 25
do (exception handlers) 27
do(for clause) 26
<double quote> " 44
double quotes 7,8
draw 36

E

e (2.718...) 33
e (decimal scale factor) 7
edge.violation 23
Edge.violation 57
eformat 33
else (conditional) 25
empty string ("") 8
end (clause brackets) 17
end of input 29
enhance 34
enhance.image 57
enhanced.screen 23
entering data to a table 31
environment 34
Environment 39
eoi() 29
epsilon 40
equality (strings) 13
equality of pointers 23
equality of procedures 24
equivalence, structure classes 20
eq_op 11
error reporting 54
error.context 52
Error.event 54
error.explain 52
error.fault 52
error.record 52
error.records 40
escape character (') 8
event overflow 54
events — see exceptions

events	40,54
events (global pntr)	28
events.strc	52
exception condition	27
exception handlers	27
exception handling (print)	23
exception value	27
exceptions	27,54
exceptions, predefined	53
exceptions, system	28
exclusive or — see xor	
exec	38
exp	33
<exp1>	10,42
<exp2>	10,11,42
<exp3>	10,11,42
<exp4>	11,13,14,42
<exp5>	11,13,22,43
<expression>	10,42
expression (Boolean)	10
expression (comparisons)	11
expression (pictures)	13
expressions — see also clauses	
expressions (pixels)	14
expressions (string)	13
<ex_id_list>	27,41

F

false	7,10
fformat	33
fiddle.r	38
field checking	54
field name as index	20
field names	19
field sizes on output	30
fields	5
<field_list>	19,41
file	9
file descriptors	30
file I/O	29,37
file literal	9
file type	5
files	29
<file_literal>	9,43

fill	36
Final.destination	57
find.substr	34
float	12
flush	30,37
font	52,58
Font	57
font.height	52,58
fonts	22,23,57,58
for clause	26
fork	38
formal parameter	24
forward declarations	24
forward refs to procedures	54
from	14
Frror.event	54

G

Garbage.collect	38
garbage.collect	38
generic types	5
gformat	33

H

handler	41
handlers (exceptions)	27
hole in scope	17
horisontal tab	8
horizontal tab (in input)	29

I

i.enter	31,34
i.lookup	31,34
i.scan	31,35
i.w	30,40
I/O	29
<identifier>	16,41
identifier scope	17
identifiers	16
<identifier_list>	41
if clause	25

iformat 34
ignoring layout chars in input 29
IliffeBounds 53
IliffeBounds.event 54
IliffeBounds.lwb 53
IliffeBounds.upb 53
illegal input 54
image 20
image expressions 20
image type 5
<image_constructor> 20,44
in (pictures) 13
in (print) 22
include (&) 14
inclusive or — see ror
increment in for loop 26
index out of range 54
indexing 20
info 52,58
init_op 41
input 29
input event 29
input exception 29
input.pending 40
int 9
integer division 12
integer literals 7
integer type 5
interrupt 34
Interrupt 53
Interrupt.event 54
interrupts — see exceptions
interrupts 54
<int_literal> 7,43
int_mult_op 12
is 11,18
isnt 11,18
iteration 26

J

join (^) 14
justification in output fields 30

K

key	31
keyboard interrupts	54

L

layout characters in input	29
left to right execution	4,10
length (of string)	33
let (declaration)	16
letter	34,41
let_decl	41
lft	52
limit (images)	20,21
limit in for loop (to)	26
Limit1	53
Limit1.event	54,55
Limit1.image	53
Limit1.Xoffset	53
Limit1.Yoffset	53
Limit2	53
Limit2.event	54,55
Limit2.image	53
Limit2.Xdim	53
Limit2.Xoffset	53
Limit2.Ydim	53
Limit2.Yoffset	53
line	35
line drawings	13
line.end	39
line.number	33
%lines	51
%list	51
list.database.dir	35
<literal>	43
literal type	5
literals	7
ln	33
locator	22,36
logarithm	33
looking up a table	31
loops	26
lsb ([ )	14,44
lwb (vector bounds)	19

M

magnifying pictures	14
make.proc	39
mat.mult	38
maximum integer value	40
maxint	40
maxreal	40
menu	36
Menu	53
Menu.event	55
Menu.reason	53
menus	38
meta language	4
meta symbols	4
minimum integer value	40
minus sign	7
mode (read or write) of db	32
modulo arithmetic	12
mouse	22,52
mrtre	52
msge	52
multi dimensional vectors	18
multiplication	12
<mult_op>	11,44

N

'n	8
name	13,42
<named_param_list>	9,43
nand (images)	21
natural logarithm	33
negative expressions	7
newline	8
newline (in input)	29
newpage	8
nil	8
nilpic	52
%nolist	51
non rectangular arrays	18
non-strictness (Booleans)	10
nonvoid clauses	17
nonvoid type	5
nor (images)	21

not (~)	10
not (images)	21
not and — see **nand**	
%noul	51
nullfile	9
nullproc	24
nxtre	52

O

'o	8
object declarations	16
of (arrays)	18
of (case clause)	25
of (images)	20
of (vectors)	18
off	8
on	8
onto (images)	21
onto (print)	22
opdb.result	52
open	30,37
open.database	32,35
OpenDatabase	53
OpenDatabase.event	54
OpenDatabse.explain	53
OpenDatabse.name	53
opendb	38
opening databases	54
openrf	38
opoo	52
oprtn.strc	52
options	34
or (Booleans)	10
<other>	8,43
out.byte	30
output	29,30
output screen	22
overflow	54
overwrite — see copy,:=	

P

'p	8
pail.eval	38

parameter types	5
parameters	24
parametrised exceptions	27
passwords on databases	32
peek()	29
persistent data	31
pi	40
pic	9,13
picture expression	13
picture operators	14
picture transformation	14
picture type	5
<picture_exp>	13,44
<pic_op>	13
Pixel	36
pixel	9
pixel expressions	14
pixel literals	8
pixel planes	15,54
pixel type	5
pixel.depth	40
PixelBounds	53
PixelBounds.event	54
PixelBounds.index	53
PixelBounds.pixel	53
<pixel_literal>	8,43
plane.of	39
planes (image)	22
planes (pixel)	15
plus sign	7
pntr	9
<pntr_literal>	8,43
pnx	52
pnx.line	39
pny	52
poin.strc	52
point.strc	52
point.x	52
point.y	52
pointer checking (is isnt)	18
pointer literal	8
pointer semantics	18
pointer type	5
points	14
position of the mouse	22
post.mortem	39

precedence, bool operators 10
precedence rules (arith ops) 12
precedence table (all ops) 15
prelude 52
primitive data types 5
primitive expressions 10
principle of correspondence 24
<print> 22,23,42,57
PRINT 57
print.destination 57
print.destination structure 23
printable type 5
printing fonts 57
printing images 57
<print_list> 22,57
proc 9
procedure call 24
procedure literals 9
procedure type 5
procedures 24
<proc_clause> 9,44
<proc_literal> 9,43
<proc_param_list> 9,41
<proc_param_type> 9,43
<proc_para_type> 9,42
<proc_type> 9,41
productions 4
<program> 41
program layout 51
pub.dummyptr 40
public standard functions 33

Q

quoted quote ('") 8

R

r.w 30
r.w 40
rabs 33
raise (exceptions) 27
rand (images) 21
random 34
range checking 54

<raster>	21,42
raster operations	21
Raster.op	57
<raster_op>	21,42
read()	29
read.a.line()	29
read.blocks	37
read.byte()	29
read.real	37
ReadB	53
readb()	29
ReadB.char	53
ReadB.event	54
ReadI	53
readi()	29
ReadI.char	53
ReadI.event	54
reading	29
ReadR	53
readr()	29
ReadR.char	53
ReadR.event	54
ReadS	53
reads()	29
ReadS.char	53
ReadS.event	54
read_proc	43
real	9
real division	12
real literals	7
real type	5
<real_literal>	7,43
real_mult_op	12
recursive declarations	24
recursive procedures	54
<rel_op>	11,44
rem	11
remainder	12
remove.database	35
rename.database	35
repeat clause	26
resizing the screen	54
restore.database	35
result type	5
resultant types	6
rght	52

root table	32
root.of.db	52
ror (images)	21
rotate (pictures)	13,14
rsb ( ])	14,44
run time errors — see exceptions	

S

s.enter	31,34
s.i	29
s.lookup	31,34
s.o	30,40
s.scan	31,35
s.w	30,40
scale (pictures)	13,14
scale factors	7
<scale_factor>	7,43
scope rules	17
scrbl.strc	52
screen	22,40,54
scrolling	23
seek	30,37
selected	22,52
selection from strings	13
selection of cases	25,26
semi-colons	51
<sequence>	17,41
sequence of characters	7,8
sequences	16
set.cursor	39
set.locator	39
settyp	39
shade	52
sharing	18
shift (pictures)	13,14
shift.l	37
shift.r	37
side effects	4,10
signals — see exceptions	
Simple.menu	38
sin	33
single quote (')	8,44
space (in input)	29
special characters	8

<special_character>	8,43
<special_follow>	8,43
sqrt	33
square root	33
standard function	33
standard identifiers	33
standard_exp	43
star (*)	11,44
step (increment in for loop)	26
strictness	10
string	9
string comparison	13
string concatenation (++)	12
string expressions	12
string literals	7,8
string quotes (")	7,8
string type	5
string.to.tile	37
StringTooLong	53
StringTooLong.event	55
StringTooLong.left	53
StringTooLong.right	53
StringToTile	53
StringToTile.event	55
StringToTile.reason	53
<string_literal>	8,43
structure	19
structure class declaration	19
structure classes	5
structure classes, parameters	24
structure equivalence	20
structure fields	5
structure type	5
structure.table	38
StructureAccess	53
StructureAccess.class	53
StructureAccess.event	54
StructureAccess.pntr	53
structures	19
<structure_decl>	9,19,41
SubImage	53
SubImage.event	54
SubImage.image	53
SubImage.length	53
SubImage.start	53
subpixels	15

subscript out of range 54
SubString 53
SubString.event 55
SubString.length 53
SubString.start 53
SubString.string 53
substrings 55
subtraction 12
sunstrings (|) 13
syntactic categories (<...>) 4
syntactic choice (|) 4
syntactic grouping ([...]) 4
syntactic repetition (*) 4
syntax 4
system 38
system events 28
system exceptions 28

T

't 8
tab 8
tab (in input) 29
table 31,34
tables 31
terminal I/O 29
terminal symbols 4
testing classes (is isnt) 11
text 14
the.buttons 22,52
the.chars 52,58
then (conditional) 25
three dimensional images 20
tiles 23,57
time 34
times ten to the power 7
%title 51
to 14
to (for clause) 26
to (images) 20
TooManyEvents 53
TooManyEvents.event 54
trace 34
transactions 31
transformations on pictures 14

translation of pictures 14
trnsfrm 52
trnsfrm.strc 52
trnsfrm.x 52
trnsfrm.y 52
true 7,10
truncate 12,33
truth tables 10
<type> 9,41
type (procedures) 9
type correctness 5,6
type deduction 16
type matching rules 45
type rules 5
type type 5
<type1> 9,41
types 5
type_op 11

U

%ul 51
uncaught events 54
Uncaught.event 28,54
underflow 54
UNIX calls 38
upb (vector bounds) 19
using (print) 22

V

value, call by 24
variable.image 36
variables 16
vector 18
vector bounds 18
vector type 5
VectorBounds 53
VectorBounds.event 54
VectorBounds.index 53
VectorBounds.lwb 53
VectorBounds.upb 53
vectors 18
<vector_constructor> 18,44
void type 5

W

wait	38
wait.for	39
when (exception handlers)	27
where	11
while clause	26
WindowChange	53
WindowChange.event	54
windows	21,54
<write>	30,42
write	30,42
write.blocks	37
writer	37
<write_list>	30,42
writing	29
writing to images	22

X

X.at	57
X.dim	36
X.pos	22,52
xnor (images)	21
xor (images)	21
xxl	52
xxr	52

Y

Y.at	57
Y.dim	36
Y.pos	22,52
yyl	52
yyr	52