CHAPTER 7

PASCAL

This chapter again provides a VDM definition of an actual programming language. In some senses this definition of standard Pascal is very revealing. The sheer length of the definition in comparison to that of ALGOL 60 requires some comment. The type definition facility in Pascal accounts for much of the length of the static semantics. But in many cases (e.g. variant records) one is forced to the conclusion that features of the language do not always fit together in a natural way. The language is, however, held in very high regard especially for teaching purposes. The resolution of this apparent contradiction appears to be that simple programs normally function in a "natural" way but that the combination of language features often leads to complexity. The text explains the models of those language concepts not discussed in chapter 4 and makes more detailed comments on the language. The chapter provides an example of how a formal definition can be used to study and criticize a language.

CONTENTS

7.1 INTRODUCTION

This chapter is an attempt to give a complete, formal definition of the Pascal programming language. Until recently, the "official" definition of Pascal was the "grey book" [Jensen 75a]. This definition is incomplete in that much of the fine detail of the language is undefined because the document attempts to be both a tutorial introduction and a definition. In 1979 it was proposed that Pascal should be officially standardized, and a BSI (British Standards Institute) document has been published to this effect. The standard goes a long way towards removing ambiguities and lack of rigor of the grey book, but being written in English, it still leaves some questions unanswered.

Several formal definitions of Pascal exist [Hoare 73d, Tennent 80a], but these do not address the complete language as defined in the BSI standard. The definition that follows covers all of the Pascal language, including conformant arrays and variant records.

The main difference between Pascal and the simple programming language in Chapter 4 is the addition of declared data-types and records. Both extensions are large, but can easily be handled, though there are some problems with Pascal variant records. There are also several other smaller extensions which need to be considered, but these do not fundamentally change the approach to the formal definition of a programming language.

One of the first major decisions to be made is the form of the abstract syntax. This definition has chosen to use one which is fairly close to the concrete syntax of Pascal. This has the advantage of making the relationship between the two more obvious; it also also has several disadvantages which will be discussed below.

Each additional tool which is necessary for the definition will be introduced under an appropriate heading.

7.1.1 Comments on Static Semantics

Static Environment

The static environment contains all global information necessary to prove harmony between the declaration and the context of an applied occurence of an identifier. Therefore the static environment contains the declara-

tions of all known identifiers. In addition to declarative information, some restrictions of Pascal require extra information to be present. For example, the control variable of a *for* loop must be declared in the block in which it used, and it cannot be passed by reference. This extra information is collected in components of the static environment, the selector names of which are suffixed by *-info*.

According to the different needs, the static environment is not simply a map, but a tuple of maps and sets. To facilitate mental a grasp of context condition formulas, access to the static environment is modelled by sets of functions: *in-X(id)ρ* and *out-X(id)ρ*. *in-X(id)ρ* checks for the occurrence of *id* in the *X*-component of the static environment ρ. *out-X(id)ρ* yields the value of *id* in the *X*-component of ρ.

Entering a new scope, local information is added to the static environment by the set of functions *merge-X(°)ρ*. According to the visibility rules, however, all information redefined in the inner scope must first be eliminated by the function *erase*.

Type Concept

Equality of types in Standard Pascal is based either on type equations relating them, or on them being the same required type or constructed type and on their structural equality. Depending on the context, type evaluation of components requires either the name or the structure of the type. Functions yielding the name of a type are prefixed by *T* (for type), and functions yielding the structure are prefixed *NT* (for new type). For example:

```
type A = array ... ;
type B = A;
var  C : B;
```

The type name evaluation of the component *C* gives the type name *B*. The new type evaluation of *C* gives *array* ... .

Some expressions do not posses a type name. Their type is either a required type (*Real*, *Integer*, *Boolean*, *Char*) or a constructed type (a type constructed by the 'string' or 'set' constructor or the 'nil' type). The constructed types are the main cause of complexity in the formulae below.

7.1.2 Comments on Dynamic Semantics

Passing Information to the Program

The Pascal Standard does not define how actual parameters are associated with the formal parameters of a program. The actual parameters could be passed by value, by reference, or by some other parameter-passing mechanism. This definition uses a value-result type mechanism; the actual parameters are described as a map from identifier into values, and results are returned by the same map. Providing that no side-effects occur while the program is running (e.g. the passing mechanism is by reference, and the operating system causes a change in an argument location) the value-result model is adequate.

Environments

The Block-environment is made up of three components, the type environment, the (static) environment, and the active identifier set. It contains all the information necessary for the declaration of identifiers, and the association of identifiers with their meaning. Type identifiers can be distinguished syntactically; all other identifiers can only have interpretations associated with them if it is known how they were declared. It is, in general, impossible to determine the meaning from the context of an identifier within a block. The type environment is used to map type identifiers into the associated type information. The static environment maps all other identifiers as appropriate: variables to their denotations, constants to their values.

The decision was made to represent Pascal programs by an abstract syntax very similar to the concrete syntax. Information that could have been associated with a node of the abstract syntax tree is in one of the maps of the Block-environment. Associating type information with the appropriate node of the program representation would remove the need for the type environment, and also remove some elements from the domain of the static environment, but at the expense of a more complicated abstract syntax. With the current choice of abstract syntax, the well-formed conditions can be more easily related to the semantic checking part of a compiler.

As type definitions have scope rules which are identical to the rules for variables, it is necessary that dynamically allocated storage (storage

obtained by the procedure *new*, and buffers) should use the correct environment. Because of this requirement, pointers and buffers have type and static environment information associated with them.

The *s-aid* component is used as in the simple language of Chapter 4.

Hops and Jumps and Goto's

The use of goto is heavily restricted in Pascal. It is only permissible to jump out of programming constructs. If the target of the goto is out of a procedure, the target statement must not be contained within any other statement of the procedure containing the target. These restrictions allow a straight-forward handling of gotos.

Jumps out of a procedure block are dealt with by the usual tixe mechanism. *cue-i-statement-list* is used to pick up the interpretation from the target statement. For a local goto, it is just necessary to "guard" the interpretation of a *flow-of-control* structure with a tixe statement whose domain is all the "immediate" labels. Again *cue-i-statement-list* can be used to pick-up the interpretation from the correct point (see chapter 5 or 6).

Records

Records which do not contain variant components are easily modelled. For example the record:

var r: record a: At; b: Bt; c: Ct end;

is modelled in the environment map by a component:

$\ldots, r \mapsto [a \mapsto loca, b \mapsto locb, c \mapsto locc], \ldots$

The storage mapping will map *loca*, *locb*, and *locc* into values. An access to a location, for example *r.a*, is given by *env(r)(a)*. The semantics of the *with* statement are defined by overwriting the current environment map with the range element of *env* corresponding to the identifier of the with statement. For example *with r do* ... will change the environment:

$nenv = env + env(r)$

Within the statement part of the *with* statement, the updated environment is used. A reference to the variable *a* will deliver the location *loca*.

If variant records are to be modelled, an extra complication occurs. Consider the following Pascal program fragments:

```
var r: record
         a: Int;
         case b: Colour of
            red:    (w: Int);
            yellow: (x: Char; y: Int);
            green:  (z: Real)
       end;
...
r.b := red;
r.w  = 4;
...
r.b  = green;
```

After the first assignment the integer location *w* is active - an integer value can be assigned to it. The other two locations corresponding to *x*, *y*, and *z* cannot be referenced. After the assignment of the value *green* to *r.b*, the value in the integer location, *r.w* is lost (and also, perhaps, the location). The *real* location corresponding to *r.z* now becomes active (and useable) but the value contained within it is undefined. A reference to the location *r.w* is now illegal, the location has "vanished". A valid implementation could be to allocate storage for a component of a variant record that became active, and free the storage of a component that becomes inactive. This may cause a different integer location to be allocated for *w* when, in the above example, the tag location is reset to the value *red*. This is not what happens in most implementations of Pascal. (Unlike PL/I, there is no way of detecting the address of a location in a variant record in Pascal, as there does not exist a function to exextract the address of a location.)

If the full implications are considered, the following problem occurs. The environment map associates an identifier with its location and, within a procedure, should be static - constructed on procedure entry, and remaining unchanged until procedure exit. If the environment is to deal with variant records, it must model the (possible) allocation and de-allocation of storage within a variant record, and this violates the

static nature of the environment map within a procedure. (The environment should be "read-only"; the state is designed to contain "changeable" information.)

It can be argued that it is unnecessary to model storage to this level of detail, but this would bias the definition towards an implementation that allocated all the storage of a variant record and re-used that storage as necessary. An implementation that wanted to allocate and free variant record storage as required (for space reasons, perhaps) would need to show that this equivalent to one which re-used storage; this may be difficult. A definition should be general enough to allow either implementation, and make it easy to show both are valid.

Having chosen to model variant records with storage allocated and freed as necessary, a solution to the environment problem is to split it into a static part, which remains unchanged during the life-time of a procedure, and a dynamic part which models the changes that can occur across an assignment to the tag field. Since the changes occur across a statement, the obvious place for the dynamic part of the environment to reside is in the state.

The model for variant records, then, is as follows. The static environment contains the identifiers associated with their (fixed) denotations; those which are part of a variant component have linking information to the dynamic environment. The association of the identifiers of the variant components with their denotation is in the dynamic environment. For the program above the static environment would contain:

$$\ldots\ r \mapsto [a \mapsto loca, b \mapsto \ldots, w \mapsto did,\\ x \mapsto did,\ y \mapsto did, z \mapsto did], \ldots$$

where *did* is an identifier. The dynamic environment would contain, if the value in *b* was *yellow*:

$$\ldots, did \mapsto [\ x \mapsto locx, y \mapsto locy], \ldots$$

ignoring for the moment what the tag denotation would.

The environment is now a mapping from *Id*entifiers into *Den*otations (for all except varient records, in which case it maps to a *D*ynamic environment *Id*entifier (so that the correct component of the dynamic environment can be found):

$Env: Id \xrightarrow{m} (Den \mid Did)$
$Denv: Did \xrightarrow{m} Env$

Now it is time to consider the denotation of the tag field. If the value in tag location is changed, then the storage associated with the old active component of the variant record needs to be freed, and new locations allocated for the new active component. The dynamic environment contains sufficient information for the de-allocation step, and will need to be updated to reflect the changes. Information as to what storage needs to be allocated is required, and this is best associated with the tag denotation in the static environment. The following tag denotation will allow this.

$t \mapsto mk\text{-}Tag\text{-}den(loc, did, type\text{-}information)$

The first component is the *tag* location, which contains the current tag value, and the *did* is the dynamic environment identifier which is used to obtain the appropriate part of the dynamic environment. The *type-information* is necessary so that the storage can be allocated; it is the variant-component from the abstract syntax.

A further complication occurs in Pascal because it is possible to have a tag field with no associated identifier. The rules for this type of variant record need to be understood:

```
var r: record
         a: A;
         case colour of
           red:   (c:X);
           green: (y:Y;z:Z)
       end;
```

On entry to the procedure containing this declaration, the location for a is allocated, but nothing can be done yet, with the variant part. Assignment to x makes the *red* component active, and assignment to y or z makes the *green* component active.

This causes further problems because there is now a tag location with no associated identifier. There is no natural home for this location - in either the static or dynamic environment. Further thought indicates that it is not necessary to have a location for the tag. There are other im-

plications - whereas if there is a tag location which has an associated identifier, it is only an assignment to the tag that causes the dynamic environment to change. Assignment to any component of a variant record which does not have a tag identifier can cause the dynamic environment to change (or become invalid). This can be solved by adding a second component to the dynamic environment which contains information about potentially valid references (which could cause storage to be allocated and de-allocated).

Consider the following declaration:

```
var r: record
         a: A;
         case colour of red:   (w:W);
                       green:  (x:X;
                                case sex of male:   (y:Y);
                                            female: (z:Z))
       end;
```

On entry to the procedure containing this declaration, the static environment from the *Block-env* component and the dynamic and variant environments of the state look like:

Static: $\ldots, r \mapsto [a \mapsto loca, w \mapsto did, x \mapsto did, y \mapsto did, z \mapsto did], \ldots$

Dynamic: $\ldots, did \mapsto [], \ldots$

Variant: $\ldots, did \mapsto [w \mapsto [red \mapsto ("w:W"), green \mapsto ("x:X; cases sex of \ldots")], x \mapsto (\ldots), y \mapsto (\ldots), z \mapsto (\ldots)]$

with x, y, and z mapped to the same value as w.

An assignment to z produces the following changes to the dynamic and variant environemnts, note that the static environment remain unchanged:

$z := 1$

Dynamic: $did \mapsto [x \mapsto locx, y \mapsto did', z \mapsto did\ ]$
$did' \mapsto [z \mapsto locz]$

Variant: ..., $did \mapsto$ [as before],
$did' \mapsto [y \mapsto$ ("*case sex of* ..."),
$z \mapsto$ ("*case sex of* ...")]

The static and dynamic environments together give the locations which are currently active, those that could have values in them if initialized. The variant environment gives those components which are potentially active (no locations, and hence no values) but they can be assigned to. This can be seen with the following:

```
var r: record
         a: A;
         case c: colour of red:    (x:X)
                           green: (case sex of male:    (y:Y);
                                                female: (z:Z)
       end;
```

If $r.c$ contains *red*, the three environment components look like:

Static: ..., $r \mapsto [a \mapsto loca, c \mapsto (locc, didc, ...), x \mapsto didc,$
$y \mapsto didc, z \mapsto didc]$, ...

Dynamic: ..., $didc \mapsto [x \mapsto locx]$, ...

Variant: ...

Note that y and z do not appear in the variant environment since as the value *red* is in the tag location c, assignment to either of the values is illegal.

If $r.c$ contains *green*, and a value has been assigned to $r.z$, the dynamic and variant environment look like:

Dynamic: ..., $didc \mapsto [y \mapsto vdidc, z \mapsto locz]$, ...

Variant: ..., $vdidc \mapsto [y \mapsto$ ("*case sex of* ..."),
$z \mapsto$ ("*case sex of* ...")], ...

Assignment to $r.y$ is now valid, and there is sufficient information in the two environment mappings to allocate storage for $r.y$.

For a reference in an expression, or a similar context, it is the static and dynamic environment which give all the information. For a reference on the left-hand side of an assignment statement, or a similar context, it is the three environments together which combine to give the necessary information. If the information is not present, then the reference is illegal. The two operations, *lhs-apply* and *rhs-apply*, resolve this.

The advantage of this model is that variant record problems have been localized, and the simple treatment of the *with* statement need not be altered; the (static) environment can be changed locally as before. There are other solutions which can disguise the variant record problem. These code all the information in a structured form in the (static) environment, but *with* is not defined in such a natural way. There is also the problem of passing active locations by reference; the associated tag has to be marked as read-only.

The *allocated-den* component of the Semantic Domain is used to diagnose a restriction on record assignment: it is illegal to have a variant record on either (or both) sides of an assignment statement if it was created by the version of the *new* procedure which allows tag values to be specified.

Pascal allows a component of a record to be another record. This is modelled in the definition by *Record-den* being a subset of *Loc* objects.

Arrays

The storage model for arrays is easier for Pascal than for the simple language. The reason for this is that Pascal defines multidimensional arrays to be an array of an array of an ^{ooo} . Even though Pascal allows the use of any ordinal type as indices, rather than restricting them to be integers, this leads to a simpler model.

```
type colour = (red,blue,green,yellow);
var a: array [colour] of Integer
```

The array declared in the Pascal program fragment above can be represented by a mapping:

a: Colour $\overrightarrow{m}$ *Integer*

with a simple condition that the domain of the mapping is the set colour. A multi-dimensional array such as:

var b: array [colour,1..20] of Integer

is defined to be a syntactic short hand notation for:

var b: array [colour] of array [1..20] of Integer;

This array can be represented by a mapping of Colour into Objects which are *arrays [1..20] of Integer*, thus:

b: Colour $\overrightarrow{m}$ *Array*
Array: {1:20} $\overrightarrow{m}$ *Integer*

Note that there is a requirement that all range elements are the same type.

The model for arrays simplifies to:

Array-den: Ordinal $\overrightarrow{m}$ *Den*

where *Den* is any object that can be stored in an array, including another array.

Conformant Arrays

One of the major criticisms of the original Pascal language was the inability to write a general procedure or function which took array arguments, such as one to calculate the length of a vector. The bounds of an array are part of its type, and so a procedure needed to be written for each possible array size. Standard Pascal has extended the class of arguments to allow arrays to be passed to a procedure with information on their bounds. The bounds are declared as part of the parameter description. When the procedure is called, the identifiers corresponding to the array bounds are set to the correct value. Note that these identifiers have the same properties as the identifiers declared in the const part of a block.

Input-Output

The definition should allow implementations which free and allocate buffers after certain operations. The reason for modelling this is again for easier proof of correctness of a compiler. If the definition did not specify this possibility, it would be necessary to prove that an implementation that did free and reallocate buffers was valid. Note that I/O operations are not allowed file whose buffer has been passed by reference, since the buffer could be freed.

Invalid references

The problem of the "dangling reference" must be handled by the definition. This occurs when a program frees a storage location while there is still a reference to it. Either the storage must not be freed, or access to it through the reference must be an error. This problem can occur in PASCAL in several ways. A file buffer can be passed by reference to a procedure, and an I/O operation is performed on that file. As PASCAL allows the buffer to be freed and re-allocated, a reference to the original buffer is invalid. A component of a variant record can be passed by reference; changing the tag value will cause the invalid reference problem to occur. There is also the familiar problem of the *dispose* operation on a storage location to which there exists a (separate) reference.

The definition detects these problems by allocating and freeing storage for both buffers and variant record components as required. Whenever a new storage location is allocated, it will have a new identifier. Hence any reference to storage which has been freed will be in error, as the storage identifier will not be in the domain of the storage map. Where Pascal specifically forbids an action that could cause the invalid reference problem, this definition has chosen to record what would happen if it were allowed. This avoids additional complexity in the definition.

Conclusions

This definition of Pascal is long in comparison to that of the ALGOL 60 definition of chapter 6. Much of the additional length is concerned with the extra features of Pascal, and the associated problems.

The concept of user-defined types, adds considerably to the semantic checks which need to be done. For a larger, typed language, such as Ada,

it would be better to have two abstract syntaxes. The first would be close to the concrete syntax of the language being defined. The second would be suitable for use by the dynamic semantics. Pascal is not large enough to warrant this extra complication.

The type mechanism of Standard Pascal allows all type checking to be done at compile time, unlike grey book Pascal and ALGOL 60. In both these languages the matching of arguments, which are procedures or functions to their parameter declarations, has to be a run-time check. Unfortunately, the rules in Standard Pascal are too restrictive in this case; a procedure cannot be passed as an argument to itself.

The added complexity of of the storage model, and environment information necessary to deal with types, especially variant records, adds considerably to the size of the definition compared to that of ALGOL 60. The problems caused by variant records, both in the definition and in a "safe" implementation of Pascal, imply that they are not a good concept. ALGOL 68 style unions, perhaps with the option of visible tagfield, seem a better way to give similar facilities. With unions most of the required checking can be done at compile time, or at run time with simple code. The ability to pass a component of a variant record by reference will cause difficulties for a safe compiler (which would trap all errors). This is not allowed with ALGOL 68 unions. Variant records without a tag field identifier allow a "hole" in the type-checking for communicating with a non-Pascal environment. This is best done by code procedures as in ALGOL 60;

Finally, in ALGOL 60 most of the I/O is defined by procedures, while in Pascal it is defined by the same mechanism as the rest of the language. This implies that a definition claiming to be complete should include a formal definition of the appropriate operations.

The current definition shows that Pascal is not a simple language. Any compiler generating safe code, would need to include much run-time checking if all error situations are to be diagnosed. Much thought needs to be put into the design of a programming language; and a formal definition should be done against an abstract syntax as a way of testing designs. Much work has been done to show that the flows-of-control constructs of a programming language can benefit from using a formal approach. Building a (mathematical) storage model would benefit the other component of a programming language - the data.

7.2 SYNTACTIC DOMAINS

Program :: *s-name* :*Id*
s-args :*Id-set*
s-block :*Block*

Block :: *s-decls* :*Declarations*
s-stl :*Statement**

Declarations :: *s-labels* :*Label-set*
s-constants :(*Id* $\overrightarrow{m}$ *Constant*)
s-types :(*Id* $\overrightarrow{m}$ *Type*)
s-variables :(*Id* $\overrightarrow{m}$ *Type-id*)
s-procedures :(*Id* $\overrightarrow{m}$ *Procedure*)
s-functions :(*Id* $\overrightarrow{m}$ *Function*)

Statement :: *s-label* :[*Label*]
s-st :*Unlabelled-statement*

Unlabelled-statement = *Compound-statement* |
Assignment-statement |
Procedure-statement |
If-statement |
Case-statement |
While-statement |
Repeat-statement |
For-statement |
With-statement |
Goto-statement |
NULL-STATEMENT

Compound-statement :: *Statement**

Assignment-statement :: *s-lp:Target* *s-rp:Expression*

Target = *Variable-access* | *Function-id*

Function-id :: *s-id:Id*

Procedure-statement :: *s-nm:Id* *s-apl:Actual-parm**

```
If-statement          :: s-expr :Expression
                         s-th   :Statement
                         s-el   :[Statement]

Case-statement        :: s-expr:Expression  s-cc:Case-component+
Case-component        :: s-cs:Constant-set  s-st:Statement

While-statement       :: s-expr:Expression  s-st:Statement

Repeat-statement      :: s-st:Statement+    s-expr:Expression

For-statement         :: s-id        :Id
                         s-from      :Expression
                         s-direction :(TO|DOWNTO)
                         s-to        :Expression
                         s-st        :Statement

With-statement        :: s-var:Variable-access  s-st:Statement

Goto-statement        =  Local-goto | Nonlocal-goto

Local-goto            :: Label

Nonlocal-goto         :: Label

Expression            =  Constant-expression | Variable-expression |
                         Function-designator | Prefix-expression   |
                         Infix-expression    | Set-constructor

Constant-expression   =  Scalar-const | String-const | NIL-VALUE

Scalar-const          =  Real-const | Integer-const |
                         Char-const | Id-const

Real-const            :: Real

Integer-const         :: Integer

Char-const            :: Char

Id-const              :: Id
```

```
String-const            :: Char+

Variable-expression     :: Variable-access

Variable-access         =  Id                    | Component-variable |
                           Reference-variable    | Buffer-variable

Component-variable      =  Indexed-variable      | Field-designator

Indexed-variable        :: s-nm:Variable-access  s-expr:Expression

Field-designator        :: s-nm:Variable-access  s-id:Id

Reference-variable      :: Variable-access
Buffer-variable         :: Variable-access

Function-designator     :: s-nm:Id   s-apl:Actual-parm*

Actual-parm             =  Variable-access | Expression      |
                           Function-parm   | Procedure-parm |
                           Read-parm       | Write-parm

Function-parm           :: Id
Procedure-parm          :: Id

Write-parm              =  Default-write-parm | Width-write-parm |
                           Fixed-write-parm

Default-write-parm      :: s-expr :Expression

Width-write-parm        :: s-expr:Expression  s-len:Expression

Fixed-write-parm        :: s-expr :Expression
                           s-len  :Expression
                           s-frac :Expression

Read-parm               :: s-target :Variable-access
                           s-type   :{Integer,Real,Char}

Prefix-expression       :: s-op:Prefix-op  s-opr:Expression

Prefix-op               =  Sign | not
```

Sign = real-plus | real-minus |
integer-plus | integer-minus

Infix-expression :: *s-lp:Expression*
s-op:Infix-op
s-rp:Expression

Infix-op = and | or
| real-add | real-sub | real-mult
| real-div
| integer-add | integer-sub | integer-div
| integer-mod | integer-mult
| eq | ne | lt
| le | ge | gt
| union | intersection | difference
| super-set | sub-set | in

Set-constructor :: *Member**

Member :: *Expression* | *Interval*

Interval :: *s-low:Expression* *s-high:Expression*

Constant = *Prefix-const* | *String-const* | *Scalar-const*

Prefix-const :: *Sign* *Id*

Type = *Domain-type* | *Structured-type* |
Packed-type | *Reference-type*

Domain-type = *Type-id* | *Enumerated-type* | *Subrange-type*

Type-id :: *s-id:Id*

Enumerated-type :: *Id*$^{+}$

Subrange-type :: *s-first:Constant* *s-last:Constant*

Structured-type = *Record-type* | *Array-type* |
File-type | *Set-type*

```
Record-type              :: s-fp :(Id m→ Type-id)
                            s-vp :[Variant-part]

Variant-part             :: s-tag  :[Id]
                            s-tagt :Type-id
                            s-evp  :(Constant m→ Record-type)

Array-type               =  s-dtype:Type-id   s-rtype:Type-id

File-type                :: Type-id
Set-type                 :: Type-id
Reference-type           :: Type-id

Packed-type              :: s-type :Structured-type

Procedure                :: s-parms:Parameters  s-body:Block

Function                 :: s-parms  :Parameters
                            s-body   :Block
                            s-return :Type-id

Parameters               :: s-ids  :Formal-parameter*
                            s-type :(Id m→ Parameter-type)

Formal-parameter         =  Id+

Parameter-type           =  Var-formal-parameter          |
                            Value-formal-parameter        |
                            Function-formal-parameter     |
                            Procedure-formal-parameter    |
                            Var-array-formal-parameter    |
                            Value-array-formal-parameter

Var-formal-parameter     :: Type-id

Value-formal-parameter   :: Type-id

Function-formal-parameter :: s-parms:Parameters  s-return:Type-id

Procedure-formal-parameter :: s-parms:Parameters
```

Var-array-formal-parameter :: *s-schema:Conformant-array-schema*

Value-array-formal-parameter :: *s-schema:Conformant-array-schema*

Conformant-array-schema = *P-array-schema* | *Array-schema*

P-array-schema :: *s-ind:Index-type-spec* *s-type:Type-id*

Array-schema :: *s-ind :Index-type-spec*
s-type :Array-type-info

Array-type-info = *Conformant-array-schema* | *Type-id*

Index-type-spec :: *s-low:Id* *s-high:Id* *s-type:Type-id*

Auxiliary Definition

Required-type = {Integer,Real,Char,Boolean}

Notes on Concrete to Abstract Translation

Although the concrete syntax of Pascal and the translation algorithms from concrete to abstract syntax are not given here, a number of points about the translation mechanism need to be made.

- Concrete delimiters (e.g. ";", and ",") and comments are dropped.

- Within expressions, brackets and operator precedence are used to determine the appropriate abstract tree structure of an expression.

- Array declaration abbreviations are removed, e.g.:

  var a : array[1..3,1..4] of integer;

  is expanded to:

  var a : array[1..3] of array[1..4] of integer;

- Certain concrete syntax constructions are considered as abbreviations and are not represented by the abstract syntax. Thus:

  var id_1, id_2, ... , id_n: *new-type-desc*

is considered to be an abbreviation for:

```
type typeid = new-type-desc;
var id1     : typeid,
    id2     : typeid,
    ...       ...
    idn     : typeid;          -- where typeid is not used elsewhere.
```

- Each structured type only has *Type-id*s describing the types of its components

- All type identifiers of a program are distinct

- I/O file abbreviations are completed:

```
write(x,y)
```

is expanded to the:

```
mk-Procedure-statement("output",< ... "x" ... >)
```

7.3 STATIC SEMANTICS

7.3.1 Static Semantic Domains

```
Static-env :: s-local-labels : Label-set
              s-global-labels: Label-set
              s-constants    : Id m (Constant | Required-value)
              s-types        : Id m (Type | Required-type)
              s-variables    : Id m (Type-id | Conformant-array-schema)
              s-procedures   : Id m (Procedures |
                                     required-procedure-heading)
              s-functions    : Id m (Function-heading |
                                     required-function-heading)
              s-for-info     : Id-set
              s-function-info: Id-set
              s-tag-info     : Id-set
              s-packed-info  : Id-set

Required-type = Real | Integer | Boolean | Char | Text
```

Required-value = *Bool*

Function-heading :: *Parameters Type*

All-type = *Type* | *Required-type* | nil-type | empty-set-type
Constructed-set-type | *Constructed-string-type*

Constructed-set-type :: *Domain-type*

Constructed-string-type :: *Integer*

Test Functions of Static Environment

in-local-labels(lab)ρ Δ *labϵs-local-labels(ρ)*
in-global-labels(lab)ρ Δ *labϵs-global-labels(ρ)*
in-constants(id)ρ Δ *idϵdom(s-constants(ρ))*
in-types(id)ρ Δ *idϵdom(s-types(ρ))*
in-variables(id)ρ Δ *idϵdom(s-variables(ρ))*
in-procedures(id)ρ Δ *idϵdom(s-procedures(ρ))*
in-functions(id)ρ Δ *idϵdom(s-functions(ρ))*
in-schema-info(id)ρ Δ *idϵdom(s-schema-info(ρ))*
in-for-info(id)ρ Δ *idϵs-for-info(ρ)*
in-function-info(id)ρ Δ *idϵs-function-info(ρ)*
in-tag-info(id)ρ Δ *idϵs-tag-info(ρ)*
in-packed-info(id)ρ Δ *idϵs-packed-info(ρ)*

in-bounds(id)ρ Δ *(∃id1ϵId)(in-variables(id1)ρ) ∧*
let tid=s-variables(ρ)(id1) in
mk-Index-type-spec(id,,)ϵbounds-of(tid) ∨
mk-Index-type-spec(,id,)ϵbounds-of(tid)

in-enumerated(id)ρ Δ *(∃id1ϵdoms-types(ρ))*
(is-Enumerated-type(s-types(ρ)(id1))∧
let mk-Enumerated-type(list)=s-types(ρ)(id1)
in (∃iϵindslist)(id=list[i]))

Access Functions of Static Environment

out-constants(id)ρ Δ *s-constants(ρ)(id)*
out-types(id)ρ Δ *s-types(ρ)(id)*
out-variables(id)ρ Δ *s-variables(ρ)(id)*

```
out-procedures(id)ρ  ≜ s-procedures(ρ)(id)

out-functions(id)ρ   ≜ cases s-functions(ρ)(id):
                         mk-function-heading(parms,rtype) → parms,
                         T → required-function-heading

out-return-type(id)ρ ≜ cases s-functions(ρ)(id):
                         mk-function-heading(parms,rtype) → rtype,
                         T → required-return-type

out-bounds(id)ρ      ≜ let type∈Domain-type be s:t:
                         (∃id1∈Id)(in-variables(id1)ρ) ∧
                         let tid∈s-variables(ρ)(id1) in
                         mk-Index-type-spec(id,,type)∈bounds-of(tid) ∨
                         mk-Index-type-spec(,id,type)∈bounds-of(tid))
                       in type

out-enumerated(id)ρ  ≜ let id1∈Id be s:t: id1∈doms-types(ρ)      in
                       let mk-Enumerated-type(l)=s-types(ρ)(id1) in
                       (∃i∈inds l)(id=l[i])
```

Transformation Functions of Static Environment

```
merge(mk-Declarations(,constm,typem,varm,procm,functm))ρ ≜
  mk-Static-env(s-local-labels(ρ),
                s-global-labels(ρ),
                s-constants(ρ)  ∪ constm,
                s-types(ρ) ∪ typem,
                s-variables(ρ)  ∪ varm,
                s-procedures(ρ) ∪ [id ↦ parms |
                                     mk-Procedure(parms,)=procm(id)],
                s-functions(ρ)  ∪ [id ↦ mk-Function-heading(parms,rtype) |
                                   mk-Function(parms,,rtype)=functm(id)],
                s-for-info(ρ),
                s-function-info(ρ),
                s-tag-info(ρ),
                s-packed-info(ρ))

merge-X(Y)ρ ≜ let ρ' be s:t: s-X(ρ')=s-X(ρ)∪Y
                  ∧ (other components unchanged) in ρ'
```

Note: The *Static-env*, ρ, has been restricted before the above *merge* functions are called.

*erase(ids,labst)*ρ ≜ all *id*∈*ids* are erased from all components of type *Id* ⇸ *X* or *Id-set*, all *labels* ∈ *labs* are erased from the *local-labels* and *global-labels* components of the *Static-enf*

Initialization

```
required-static-env =
   mk-Static-env(s-local-labels : {},
                 s-global-labels: {},
                 s-constant     : ["maxint" ↦
                                    mk-Integer-constant(maxint),
                                   "true"  ↦ true,
                                   "false" ↦ false],
                 s-types        : ["Boolean" ↦ Boolean,
                                   "Integer" ↦ Integer,
                                   "Real"    ↦ Real,
                                   "Char"    ↦ Char,
                                   "Text"    ↦ Text],
                 s-variables    : ["input"   ↦ Text,
                                   "output"  ↦ Text],
                 s-procedures   : ["dispose" ↦ required-proc-heading,
                                   also for "get","new","pack","put",
                                            "read", "readln" "reset",
                                            "rewrite", "unpack","write"
                                            "writeln" ],
                 s-functions    : ["abs" ↦ required-function-heading,
                                   also for "sqr","sin","cos","exp",
                                            "ln","sqrt", "arctan","chr",
                                            "trunc","round","ord","succ",
                                            "pred","odd","eof","eoln"],
                 s-for-info     : {},
                 s-function-info: {},
                 s-tag-info     : {},
                 s-packed-info  : {})
```

7.3.2 Context Condition Functions

Function Abbreviations

WF-	Well-formedness of a:	*WFP*	- Program
WFVA	- Variable Access	*WFDP*	- Parameter Declaration
WFB	- Block	*WFSCH*	- Conformant Array Schema
WFBD	- Block Declaration	*TE*	Type of an Expression
WFD	- Declarations	*TVA*	Type of a Variable Access
WFSL	- Statement List	*NTE*	New Type of an Expression
WFUS	- Unlabelled Statement	*NTVA*	New Type of a Variable Access
WFE	- Expression	*NTT*	New Type of a Type

Function Definitions

```
WFP: Program → Bool
WFP[mk-Program(,args,block)] ≜
    is-unique-declarations(<>,[],block) ∧ is-unique-labels(block) ∧
    (let ids = all-local-id(<>,[],block) in
    (∀i∈indsargs)(let id = args[i] in
                  if id∈{"input","output"} then id∉ids
                  else id∈doms-variables(s-decls(block))) ∧
     WFB[block]erase(ids,{})required-static-env

WFB: Block → Static-env → Bool
WFB[mk-Block(decls,stl)]ρ ≜
    all-for-id(stl) ⊆ doms-variables(decls)        ∧
    let ρ' = merge(decls)ρ in
    WFBD[decls]merge-global-labels(labels(stl))ρ' ∧
    WFSL[stl]ρ'

WFBD: Declarations → Static-env → Bool
WFBD[mk-Declarations(,constm,typem,varm,procm,functm)]ρ ≜
    (∀id∈domconstm)(WFD[constm(id)]ρ) ∧ (∀id∈domtypem)(WFD[typem(id)]ρ)  ∧
    (∀id∈domvarm)(WFD[varm(id)]ρ)     ∧ (∀id∈domprocm)(WFD[procm(id)]ρ)  ∧
    (∀id∈domfunctm)(WFD[functm(id)]merge-function-info({id})ρ)

WFSL: Statement* → Static-env → Bool
WFSL[sl]ρ ≜
    let ρ' = merge-local-labels(labels(sl))ρ in
    (∀i∈indssl)(WFUS[s-st(sl[i])]ρ')
```

```
WFUS: Unlabelled-statement → Static-env → Bool

WFUS[mk-Compound-statement(sl)]ρ ≜ WFSL(sl)ρ

WFUS[mk-Procedure-statement(id,apl)]ρ ≜
   in-procedures(id)ρ ∧
   cases out-procedures(id)ρ:
      required-procedure-heading →
         is-required-procedure-correspondence(id,apl)ρ
      mk-Parameters(idl,type)     →
         is-corresponding(idl,type,apl)ρ

WFUS[mk-Assignment-statement(target,exp)]ρ ≜
   if is-Function-id(target)
      then let id = s-id(target) in
            in-functions(id)ρ ∧ in-function-info(id)ρ ∧
            WFE[exp]ρ ∧
            is-assignment-compatible(out-return-type(id)ρ,TE[exp]ρ)ρ
      else WFVA[target]ρ ∧
            WFE[expr]ρ     ∧
            (is-Id(target) ⊃ ¬in-for-info(target)ρ) ∧
             is-assignment-compatible(TVA[target]ρ,TE[exp]ρ)ρ

WFUS[mk-If-statement(exp,th,el)]ρ ≜
   WFE[exp]ρ  ∧  WFSL[<th>]ρ  ∧  (el≠nil ⊃ WFSL[<el>]ρ)  ∧
   NTE[exp]ρ=Boolean

WFUS[mk-Case-statement(exp,ccl)]ρ ≜
   WFE[exp]ρ ∧
   let type = TE[exp]ρ in is-ordinal(type)ρ ∧
   (∀i∈indsccl)(let mk-Case-component(cs,sp) = ccl[i] in
                  (∀c∈cs)(WFC[c]ρ ∧ is-same(type,TE[c]ρ)ρ) ∧
                  WFSL[<sp>]ρ) ∧
   (∀i,j∈indsccl)(s-cs(ccl[i])∩s-cs(ccl[j])≠{} ⊃ i=j)

WFUS[mk-While-statement(exp,st)]ρ ≜
   WFE[exp]ρ  ∧  WFSL[<st>]ρ  ∧   NTE[exp]=Boolean

WFUS[mk-Repeat-statement(sl,exp)]ρ ≜
   WFSL[sl]ρ  ∧  WFE[exp]ρ  ∧  NTE[exp]= Boolean
```

```
VFUS[mk-For-statement(id,from,,to,st)]ρ ≜
   in-variables(id)ρ ∧ ¬in-for-info(id)ρ ∧ is-ordinal(NTVA[id]ρ)ρ ∧
   WFE[from]ρ ∧ WFE[to]ρ ∧ WFSL[<st>]merge-for-info({id})ρ ∧
   is-compatible(TVA[id]ρ,TE[from]ρ) ∧ is-compatible(TVA[id]ρ,TE[to]ρ)

VFUS[mk-With-statement(vs,st)]ρ ≜
   WFVA[vs]ρ ∧
   let type = NTVA[vs]ρ in
   let type1 = if is-Packed-type(type) then s-type(type) else type in
   is-Record-type(type1) ∧
   WFSL[<st>]merge-variables(all-fields(type1))
             merge-tag-info(all-tags(type1))
             merge-packed-info(if is-Packed-type(type)
                               then all-fields(type1) else {})
             erase(all-fields(type1),{})ρ

VFUS[mk-Local-goto(id)]ρ ≜ in-local-labels(id)ρ

VFUS[mk-Global-goto(id)]ρ ≜ in-global-labels(id)ρ

VFE: Expression → Static-env → Bool

VFE[mk-Constant-expression(exp)]ρ ≜
   cases exp:
      mk-Real-constant()      → true
      mk-Integer-constant(i)  → -maxint<i<maxint
      mk-Char-constant()      → true
      mk-Id-constant(id)      → in-constants(id)ρ ∨ in-bounds(id)ρ
      mk-String-constant(cl)  → lencl>1
      nil-value               → true

VFE[mk-Set-constructor(mk)]ρ ≜
   mk=<> ∨
   let es = union{if is-Expression(mk[i]) then {mk[i]}
                  else {s-low(mk[i]),s-high(mk[i])} | i∈indsmk} in
   (∀e1,e2∈es)
       (let t1 = NTE[e1]ρ in
        let t2 = NTE[e2)ρ in
        is-ordinal(t1)ρ ∧ t1=t2)
```

```
WFE[mk-Variable-expression(va)]ρ ≜ WFVA[va]ρ

WFE[mk-Function-designator(id,apl)]ρ ≜
  in-functions(id)ρ ∧
  cases out-functions(id)ρ:
    required-function-heading →
              is-required-function-correspondence(id,apl)ρ
    mk-Parameters(idl,type)   →
              is-corresponding(idl,type,apl)ρ

WFVA: Variable-access → Static-env → Bool

WFVA(id)ρ ≜ in-variables(id)ρ

WFVA[mk-Indexed-variable(va,exp)]ρ = ≜
   WFVA[va]ρ ∧ WFE[exp]ρ ∧
   let tva = NTVA[va]ρ in
   let tva1 = if is-Packed-type(tva) then s-type(tva) else tva in
   is-Array-type(tva1) ∧
   is-assignment-compatible(s-dtype(tva1),NTE[exp]ρ)ρ

WFVA[mk-Field-designator(va,id)]ρ ≜
   WFVA[va]ρ ∧
   let tva = NTVA[va]ρ in
   let tva1 = if is-Packed-type then s-type(tva) else tva in
   is-Record-type(tva1) ∧ id∈domall-fields(tva1)

WFVA[mk-Reference-variable(va)]ρ ≜
   WFVA[va]ρ ∧ is-Reference-type(NTVA[va]ρ)

WFVA[mk-Buffer-variable(va)]ρ ≜
   WFVA[va]ρ ∧
   let tva = NTVA[va]ρ in
   let tva1 = if is-Packed-type(tva) then s-type(tva) else tva in
   is-File-type(tva1)

WFD: (Constant | Type | Procedure | Function) → Static-env → Bool

WFD[mk-Prefix-constant(sign,id)]ρ ≜
   in-constants(id)ρ ∧
   WFE[mk-Prefix-expression(sign,mk-Id-const(id))]ρ
```

```
WFD[mk-Real-const()]ρ              ≜ true
WFD[mk-Integer-const(i)]ρ          ≜ -maxint≤i≤maxint
WFD[mk-Char-const()]ρ              ≜ true
WFD[mk-Id-const(id)]ρ              ≜ in-constants(id)ρ
WFD[mk-String-const(cl)]ρ          ≜ lencl > 1
WFD[mk-Type-id(id)]ρ               ≜ in-type(id)ρ
WFD[mk-Enumerated-type(idl)]ρ      ≜ true

WFD[mk-Subrange-type(f,l)]ρ ≜
   WFE[f]ρ ∧ WFE[l]ρ ∧ is-ordinal(TE[f]ρ)ρ ∧
   is-same(TE[f]ρ,TE[l]ρ)ρ ∧
   values(mk-subrange-type(f,l))ρ≠{}

WFD[mk-Array-type(dt,rt)]ρ         ≜ is-ordinal(NTT[dt]ρ)ρ ∧ WFD[rt]ρ

WFD[mk-Record-type(fp,vp)]ρ ≜
   is-unique-fields(mk-Record-type(fp,vp),vp)  ∧
   (∀id∈domfp)(WFD[fp(id)]ρ) ∧
   vp≠nil ⊃ (let mk-Variant-part(,tid,evp) = vp in
             let nt = NTT[tid]ρ                  in
             (∀c∈domevp)(WFE[c]ρ ∧ is-compatible(tid,NTE[c]ρ)ρ ∧
              WFD[evp(c)]ρ) ∧ values(nt)ρ=domevp)

WFD[mk-Set-type(dt)]ρ              ≜ is-ordinal(NTT(dt)ρ)ρ

WFD[mk-File-type(t)]ρ              ≜ is-not-containing-file-type(t)ρ

WFD[mk-Reference-type(mk-Type-id(id))]ρ ≜ in-types(id)ρ

WFD[mk-Packed-type(t)]ρ            ≜ WFD[t]ρ

WFD[mk-Procedure(parms,block)]ρ ≜ is-wf-function-procedure(parms,block)ρ

WFD[mk-Function(parms,block,type)]ρ ≜
   let t = NTT[type] in
   (is-simple(t)ρ ∨ is-Reference-type(t)) ∧
    is-wf-function-procedure(parms,block)ρ

WFDP: Parameter-type → Static-env → Bool

WFDP[mk-Var-formal-parameter(type)]ρ ≜ in-types(s-id(type))ρ
```

```
WFDP[mk-Value-formal-parameter(type)]ρ ≜
    in-types(s-id(type))ρ ∧ is-not-containing-file-type(NTT[type]ρ)ρ

WFDP[mk-Var-array-formal-parameter(schema)]ρ ≜
    WFSCH[schema]ρ

WFDP[mk-Value-array-formal-parameter(schema)]ρ ≜
    is-not-containing-file-type(schema)ρ ∧ WFSCH[schema]ρ

WFSCH: Conformant-array-schema → Static-env → Bool

WFSCH[mk-P-array-schema(index-type,type)]ρ ≜
    in-types(s-id(s-type(index-type)))      ∧
    is-ordinal(NTT[s-type(index-type)]ρ)ρ ∧
    in-types(s-id(type))ρ

WFSCH[mk-Array-schema(index-type,schema)]ρ ≜
    in-types(s-type(index-type))ρ           ∧
    is-ordinal(NTT[s-type(index-type)]ρ)ρ ∧
    if is-Id(schema)
       then in-types(schema)ρ
       else WFSCH[schema]ρ

TE: Expression → Static-env → All-type

TE[mk-Real-const()]ρ              ≜ Real
TE[mk-Integer-const()]ρ           ≜ Integer
TE[mk-Char-const()]ρ              ≜ Char
TE[mk-String-const(cl)]ρ          ≜ mk-Constructed-string-type(len cl)
TE[mk-Variable-expression(va)]ρ ≜ TVA[va]ρ

TE[mk-Set-constructor(ml)]ρ ≜
   if ml=<>
      then empty-set-type
      else mk-Constructed-set-type(if is-Expression(ml[1])
                                   then NTE[ml[1]]ρ
                                   else NTE[s-low(ml[1])]ρ)

TE[nil]ρ ≜ nil-type
```

```
TE[mk-Prefix-expression(op,)]ρ ≜
   cases op:
      real-plus,real-minus          → Real
      integer-plus,integer-minus    → Integer
      not                           → Boolean

TE[mk-Id-const(id)]ρ ≜
   if in-constants(id)ρ
      then if in-enumerated(id)ρ
              then out-enumerated(id)ρ
              else let const = out-constants(id)ρ in
                   if const∈Bool then Boolean  else TE[const]ρ
      else out-bounds(id)ρ

TE[mk-Infix-expression(opr1,op,opr2)]ρ ≜
   cases op:
      real-add,real-sub,real-mult,real-div      → Real
      integer-add,integer-sub,integer-mult,
      integer-div,integer-mod                   → Integer
      eq,ne,le,ge,gt,and,or,in,superset,subset  → Boolean
      union,intersection,difference             → TE[opr1]ρ

TE[mk-Function-designator(id,apl)]ρ ≜
   let t = out-return-type(id)ρ in
   if t=required-return-type
      then cases id:
             "abs","sqr","pred","succ"                  → NTE[apl[1]]ρ
             "sin","cos","exp","ln","sqrt","arctan"     → Real
             "trunc","round","ord"                      → Integer
             "chr"                                      → Char
             "odd","eof","eoln"                         → Boolean
      else t

TVA: Variable-access → Static-env → (Type | Required-type)

TVA[id]ρ ≜ out-variables(id)ρ

TVA[mk-Indexed-variable(va,)]ρ ≜
   let t = NTVA[va]ρ in
   if is-Packed-type(t) then s-rtype(s-type(t)) else s-rtype(t)
```

```
TVA[mk-Field-designator(va,id)]ρ ≜
    let t = NTVA[va]ρ in
    let t1 = if is-Packed-type(t) then s-type(t) else t in
    all-fields(t1)(id)

TVA[mk-Reference-variable(va)]ρ ≜
    let mk-Reference-type(type) = NTVA[va]ρ in type

TVA[mk-Buffer-variable(va)]ρ ≜
    let t = NTVA[va]ρ in
    let t1 = if is-Packed-type(t) then s-type(t) else t in
    if t1=Text then Char else t1

NTE: Expression → Static-env → (All-type | Generated-type)
NTE[exp]ρ ≜ let t = TE[exp]ρ in if is-Type-id(t) then NTT[t]ρ else t

NTVA: Variable-access → Static-env → (Type | Required-type)
NTVA[va]ρ ≜ let t = TVA[va]ρ in if is-Type-id(t) then NTT[t]ρ else t

NTT: Type → Static-env → (Type | Required-type)
NTT[type]ρ ≜ cases type: mk-Type-id(id) → NTT[out-types(id)ρ]ρ
                         T              → type
```

7.3.3 Auxiliary Functions

is-unique-declarations: $Id^{**} \times (Id \overset{m}{\to} Type\text{-}id) \times [Block] \to Bool$
is-unique-declarations(idll,parm-decl,block) ≜

all identifiers occurring
- in *idll*,
- in *mk-Index-type-spec* at *s-low* or *s-high* position in *parm-decl*,
- in *doms-constants(s-decl(block))*,
- in *doms-types(s-decl(block))*,
- in *doms-variables(s-decl(block))*,
- in *doms-procedures(s-decl(block))*,
- in *doms-functions(s-decl(block))*,
- in *mk-enumerated-type* in *s-types(s-decl(block))* are different and

all identifers occurring in *idll* occur in *domparm-decl*, and there is no cyclic definition in *s-constants(block)* and each cyccyclic definition in *s-types(block)* contains at least one *mk-Reference-type*

all-local-id: Id** × *(Id* $\overrightarrow{m}$ *Type-id)* × [*Block*] → *Id-set*
all-local-id(idll,parm-decl,block) $\triangleq$ all identifiers tested above

is-unique-labels: Block → *Bool*
is-unique-labels(block) $\triangleq$
s-labels(s-decl(block)) is equal to the set of labels occurring in *s-stl(block)* and no label occurs more than once in *s-stl(block)*

all-tags: Record-type → *Id-set*
all-tags(mk-Record-type(,vp)) $\triangleq$
if vp=nil then {} *else let mk-Variant-part(tag,,evp) = vp in*
union {*all-tags(evp(c))* | *c* ∈ *dom evp*} ∪
if tag=nil then {} *else* {*tag*}

is-unique-fields: Record-type → *Bool*
is-unique-fields(mk-Record-type(fp,vp)) $\triangleq$
all identifiers occurring
in *domdfp*,
in *mk-variant-part* at *s-tag* position and
in *dom(s-fp(mk-Record-type()))* *in vp*
are different

all-for-id: Statement-list → *Id-set*
all-for-id (stl) $\triangleq$
all identifiers occurring in *mk-For-statement* at *s-id* position.

is-variant-constants: Record-type×Actual-parm-list → *Static-env* → *Bool*
is-variant-constants(mk-Record-type(,vp),apl)ρ $\triangleq$
vp≠nil ∧ *is-Constant(hdapl)* ∧ *WFE[hdapl]ρ* ∧
let mk-Variant-part(,,evp) = vp in
hdapl∈domevp ∧
(tlapl≠<> ⊃ *is-variant-constants(evp(hdapl),tlapl)ρ)*

all-fields: Record-type → *Id-set*
all-fields(mk-Record-type(fp,vp)) $\triangleq$
domfp ∪ *if vp=nil*
then {}
else let mk-Variant-part(tag,,evp) = vp in
union {*all-fields(evp(c)* | *c∈domevp*} ∪
if tag=nil then {} *else* {*tag*}

```
values: Domain-type → Static-env → Constant-set
values(type)ρ ≜
  cases type:
     mk-Subrange-type(f,l) →
        cases NTE[f]ρ:
           mk-Enumerated(idl) → {mk-Id-const(idl[i]) |
                                    (∃j,k∈inds idl)
                                       (idl[j]=f ∧ idl[k]=l ∧ j≤i≤k)}
           Integer → let mk-Integer-const(f1) = f in
                     let mk-Integer-const(l1) = l in
                     {mk-Integer-const(i) | f≤i≤l}
           Char    → Implementation defined set of Char-const
           Boolean → if f=mk-Const-id("true") ∧ l=mk-Id-const("false")
                     then {} else {f,l}
     mk-Enumerated(idl) → {mk-Id-const(idl[i]) | i∈inds idl}
     Integer →  {mk-Integer-const(i) | -maxint≤i≤maxint∧
     Char    →  Implementation defined set of Char-const
     Boolean →  {mk-Id-const("false"),mk-Id-const("true")}

is-not-containing-file-type: (Type | Conformant-array-schema) →
                                          static-env → Bool
is-not-containing-file-type(type)ρ ≜
     is-File-type(type) ∧
     for all id's occurring in mk-Type-id in type:
        is-Reference-type(out-types(id)ρ) ∨
        is-not-containing-file-type(out-types(id)ρ) holds.

is-same: All-type × All-type → Static-env → Bool
is-same(t1,t2)ρ ≜
   is-Type-id(t1) ∧ is-Type-id(t2) ∧
   (t1=t2 ∨ is-same(s-id(t1),t2)ρ ∨ is-same(t1,s-id(t2))ρ)

is-ordinal: All-type → Static-env → Bool
is-ordinal(t)ρ ≜
   let nt = NTT(t)ρ in
   nt=Integer ∨ nt=Char ∨ nt=Boolean ∨
   is-Enumerated-type(nt)

is-simple: All-type → Static-env → Bool
is-simple(t)ρ ≜
   is-ordinal(t)ρ ∨ NTT[t]ρ=Real
```

```
is-compatible: All-type × All-type → Static-env → Bool
is-compatible(t1,t2)ρ ≜
    is-same(t1,t2)ρ ∨
    let nt1 = NTT[t1]ρ in
    let nt2 = NTT[t2]ρ in
    (is-Subrange-type(nt1) ∧ is-Subrange-type(nt2) ∧
        is-compatible-subranges(nt1,nt2)ρ) ∨
    (is-all-set-type(t1) ∧ is-all-set-type(t2) ∧
        is-compatible-sets(nt1,nt2)ρ)        ∨
    (is-all-string-type(t1)ρ ∧ is-all-string-type(t2)ρ ∧
        is-compatible-strings(nt1,nt2)ρ) ∨
    is-compatible-references(nt1,nt2)

is-compatible-subranges: Subrange-type×Subrange-type→Static-env → Bool
is-compatible-subranges((mk-Subrange-type(f1,),mk-Subrange-type(f2,))ρ ≜
    let t1 = NTE[f1]ρ in
    let t2 = NTE[f2]ρ in
    t1=t2 ∨ is-same(t1,t2)ρ

is-compatible-sets: All-type × All-type → Static-env → Bool
is-compatible-sets(t1,t2)ρ ≜
 cases t1:
   empty-set-type                          → true
   mk-Constructed-set-type(base1)          →
     cases t2:
       mk-Constructed-set-type(base2)      → is-compatible(base1,base2)ρ
       mk-Set-type(base2)                  → is-compatible(base1,base2)ρ
       mk-Packed-type(mk-Set-type(base2))  → is-compatible(base1,base2)ρ
       T                                   → is-compatible-sets(t2,t1)
   mk-Set-type(base1)                      →
     cases t2:
       mk-Set-type(base2)                  → is-compatible(base1,base2)ρ
       mk-Packed-type()                    → false
       T                                   → is-compatible-sets(t2,t1)ρ
   mk-Packed-type(mk-Set-type(base1))      →
     cases t2:
       mk-Packed-type(mk-Set-type(base2))  → is-compatible(base1,base2)ρ
       T                                   → is-compatible-sets(t2,t1)ρ
```

```
is-all-set-type: All-type → Bool
is-all-set-type(t) ≜
    is-Set-type(t) ∨ is-Packed-type(t) ∧ is-Set-type(s-type(t)) ∨
    t=empty-set-type ∨ is-Constructed-set-type(t)

is-all-string-type: All-type → Static-env → Bool
is-all-string-type(t)ρ ≜
    is-Constructed-string-type(t) ∨
    let nt = NTT[t] in
    cases nt:
        mk-Packed-type(mk-Array-type(dt,ct)) →
            let ndt = NTT[dt] in
            let nct = NTT[ct] in
            is-Subrange-type(ndt) ∧
            (∃l∈Integer)(ndt=mk-Subrange(mk-Integer-const(1),
                                         mk-Integer-const(l)))∧nct=Char
        T                                    → false

is-compatible-strings: All-type × All-type → Static-env → Bool
is-compatible-strings(t1,t2)ρ ≜
  cases t1:
    mk-Constructed-string-type(l1)
      → cases t2:
          mk-Constructed-string-type(l2) → l1=l2
          T                              → l1=string-length(t2)ρ
    T → cases t2:
          mk-Constructed-string-type()   → is-compatible-strings(t2,t1)ρ
          T                              → string-length(t1)ρ=
                                           string-length(t2)ρ

string-length: Packed-type → Static-env → Integer
string-length(mk-Packed-type(mk-Array-type(dt,))ρ =
   let mk-Subrange-type(,l) = NTT[dt]ρ in l

is-compatible-references: All-type × All-type → Static-env → Bool
is-compatible-references(t1,t2) =
   (is-Reference-type(t1) ∨ t1 = nil-type) ∧
    t2 = nil-type ∨
    is-compatible-references(t2,t1)
```

```
is-assignment-compatible: All-type × All-type → Static-env → Bool
is-assignment-compatible(t1,t2)ρ ≜
   is-same(t1,t2)ρ ∧ is-not-containing-file-type(t1)ρ ∨
   let nt1 = NTT[t1] in
   let nt2 = NTT[t2] in
   nt1=Real ∧ nt2=Integer ∨ is-compatible(nt1,nt2)ρ

is-corresponding: Id** × (Id →m Parameter-type) ×
                          Actual-parm-list → Static-env → Bool
is-corresponding(idll,parm-decl,apl)ρ ≜
   apl=<> ∧ idll=<> ∨
   apl≠<> ∧ idll≠<> ∧
   let l = len hdidll in
   lenapl≥l ∧
   (is-Var-array-formal-parameter(parm-decl(hd hd idll)) ∨
    is-Value-array-formal-parameter(parm-decl(hd hd idll)) ⊃
    (∀i,j∈{1:l})(is-same(TE[apl[i]]ρ,TE[apl[j]]ρ)ρ)) ∧
    (∀i∈{1:l})(is-element-corresponding(parm-decl(hdidll[i]),apl[i])ρ)∧
     is-corresponding(tlidll,parm-decl,<apl[i] | i∈{l+1:lenapl}>)ρ

is-element-corresponding: Parameter-type×Actual-parm → Static-env → Bool
is-element-corresponding(pt,ap)ρ ≜
   cases pt:
      mk-Value-formal-parameter(id)                →
         is-Expression(ap) ∧ WFE[ap]ρ ∧
         is-assignment-compatible(mk-Type-id(id),TE[ap]ρ)ρ
      mk-Var-formal-parameter(id)                  →
         is-Variable-access(ap) ∧ WFVA[ap]ρ ∧
         (is-Id(ap) ⊃ ¬in-for-info(ap)ρ ∧ ¬in-tag-info(id)ρ ∧
                ¬in-packed-info(id)ρ) ∧ ¬is-tag-access(ap)ρ ∧
                ¬is-packed-component(ap)ρ ∧
          is-same(mk-Type-id(id),TVA[ap]ρ)ρ
      mk-Procedure-formal-parameter(parms)         →
         is-Procedure-parm(ap) ∧
         let mk-Procedure-parm(id) = ap in
         in-procedures(id)ρ ∧ is-congruous(out-procedures(id)ρ,parms)ρ
      mk-Function-formal-parameter(parms,type)     →
         is-Function-parm(ap) ∧
         let mk-Function-parm(id) = ap in
         in-functions(id)ρ ∧ is-same(out-return-type(id)ρ,type)ρ ∧
         is-congruous(out-functions(id)ρ,parms)ρ
```

```
      mk-Var-array-formal-parameter(cas)      →
         is-Variable-access(ap) ∧ WFVA[ap]ρ ∧
         ¬is-packed-component(ap)ρ           ∧
         is-conformable(cas,NTVA[ap]ρ)ρ
      mk-Value-array-formal-parameter(cas)    →
         is-Expression(ap) ∧ WFE[ap]ρ ∧ ¬is-conformant-array(ap)ρ
         is-conformable(cas,NTE[ap]ρ)ρ

is-packed-component: Variable-access → Static-env → Bool
is-packed-component(va)ρ ≜
   cases va:
      mk-Indexed-variable(va1,) → is-Packed-type(NTVA[va1])ρ
      mk-Field-designator(va1,) → is-Packed-type(NTVA[va1])ρ
      mk-Buffer-variable(va1)   → is-Packed-type(NTVA[va1])ρ
      T                         → false

is-tag-access: Variable-access → Static-env → Bool
is-tag-access(va)ρ ≜ is-Field-designator(va)∧s-Id(va)∈all-tags(NTVA[va]ρ)

is-congruous: Parameters × Parameters → Static-env → Bool
is-congruous(mk-Parameters(idl1,p1),mk-Parameters(idl2,p2))ρ ≜
   lenidl1=lenidl2 ∧
   (∀i∈indsidl1)(lenidl1[i]=lenidl2[i] ∧
   let t1 = p1(hdidl1[i]) in
   let t2 = p2(hdidl2[i]) in
   cases <t1,t2>:
     <mk-Value-formal-parameter(t1),mk-Value-formal-parmeter(t2)>,
     <mk-Var-formal-parameter(t1),mk-Var-formal-parameter(t2)>
          → is-same(t1,t2)ρ
     <mk-Procedure-formal-parameter(parms1),
      mk-Procedure-formal-parameter(parms2)>
          → is-congruous(parms1,parms2)ρ
     <mk-Function-formal-parameter(parms1,type1),
      mk-Function-formal-parameter(parms2,type2)>
          → is-same(type1,type2)ρ ∧ is-congruous(parms1,parms2)ρ
     <mk-Value-array-formal-parameter(cas1),
      mk-Value-array-formal-parameter(cas2)>,
     <mk-Var-array-formal-parameter(cas1),
      mk-Var-array-formal-parameter(cas2)>
          → is-equivalent-schema(cas1,cas2)ρ
     T    → false)
```

```
is-equivalent-schema: Conformant-array-schema × Conformant-array-schema
                                          → Static-env → Bool
is-equivalent-schema(cas1,cas2)ρ ≜
   cases <cas1,cas2>:
     <mk-P-array-schema(mk-Index-type-spec(,,dt1),t1),
      mk-P-array-schema(mk-Index-type-spec(,,dt2),t2) >→
         is-same(dt1,dt2)ρ ∧ is-same(t1,t2)ρ
     <mk-Array-schema(mk-Index-type-spec(,,dt1),t1),
      mk-Array-schema(mk-Index-type-spec(,,dt2),t2)>    →
         is-same(dt1,dt2)ρ ∧
         (is-Type-id(t1) ∧ is-Type-id(t2) ∧
          is-same(t1,t2)ρ ∨
         is-Conformant-array-schema(t1) ∧
         is-Conformant-array-schema(t2) ∧
         is-equivalent-schema(t1,t2))
      T                                                 → false

is-conformable: Conformant-array-schema × Type → Static-env → Bool
is-conformable(cas,t)ρ =
   cases <cas,t>:
     <mk-P-array-schema(mk-Index-type-sepc(,,dt1),ct1),
      mk-Packed-type(mk-Array-type(dt2,ct2))> →
         is-compatible(dt1,dt2)ρ ∧ is-same(ct1,ct2)ρ
     <mk-Array-schema(mk-Index-type-spec(,,dt1),ct1),
      mk-Array-type(dt2,ct2))>                  →
         is-compatible(dt1,dt2)ρ          ∧
         (is-Id(s-id(ct1)) ∧ is-same(ct1,ct2)ρ ∨
          is-conformable(ct1,ct2))
     T                                          → false

labels: Statement* → Label-set
labels(sl) ≜
   {lab | i∈indss l ∧
          mk-Statement(lab,)=sl[i] ∧
          lab≠nil}
```

```
is-required-procedure-correspondence: Id × Actual-parm* →
                                          Static-env → Bool
is-required-procedure-correspondence(id,apl)ρ △
  cases id: "rewrite","put"
            "reset","get"   → lenapl=1 ∧
                              is-Variable-access(apl[1]) ∧
                              WFVA[apl[1]]ρ              ∧
                              let t = NTVA[apl[1]]       in
                              is-File-type(t) ∨ is-Packed-type(t) ∧
                              is-File-type(s-type(t))
            "read"          → lenapl≥2 ∧
                              is-Variable-access(apl[1]) ∧
                              NTVA[apl[1]]ρ=Text         ∧
                              (∀i∈indsapl - {1})
                                 (is-Read-parm(apl[i])   ∧
                                  let mk-Read-parm(va,t) = apl[i] in
                                  NTVA[va]ρ=t)
            "write"         → lenapl≥2 ∧
                              is-Variable-access(apl[1]) ∧
                              NTVA[apl[1]]ρ=Text         ∧
                              (∀i∈indsapl - {1})
                                cases apl[i]:
                                  mk-Default-write-parm(exp)
                                    → check-write-expr(exp)ρ,
                                  mk-Width-write-parm(exp1,exp2)
                                    → check-write-expr(exp1)ρ ∧
                                      check-write-expr(exp2)ρ,
                                  mk-Fixed-write-parm(exp1,exp2,exp3)
                                    → check-write-expr(exp1)ρ ∧
                                      check-write-expr(exp2)ρ ∧
                                      check-write-expr(exp3)ρ,
            "new","dispose" → lenapl≥1 ∧
                              let t = NTVA[hd apl] in
                              is-Reference-type(t) ∧
                              lenapl>1 ⊃
                              let t1=if is-Packed-type(t)
                                     then s-type(t) else t in
                              is-Record-type(t1) ∧
                              is-variant-constants(t1,tlapl)ρ
```

```
            "pack"          → lenapl=3 ∧
                              check-pack(apl[1],apl[2],apl[3]),
            "unpack"        → lenapl=3 ∧
                              check-pack(apl[3],apl[1],apl[2])

check-write-expr: Expression → Static-env → Bool
check-write-expr(e)ρ ≜
    WFE[e]ρ ∧ let nt=NTE[e]ρ in nt∈Required-type ∨ is-all-string-type(nt)

check-write-fields: Expression → Static-env → Bool
check-write-fields(e)ρ ≜ WFE[e]ρ ∧ NTE[e]=Integer

check-pack: Actual-parm × Actual-parm × Actual-parm → Static-env → Bool
check-pack(upk,i,pk)ρ≜
   is-Variable-access(upk) ∧ WFVA[upk]ρ ∧
   is-Expression(i)        ∧ WFE[i]ρ    ∧
   is-Variable-access(pk)  ∧ WFVA[pk]ρ  ∧
   let upkt = NTVA[upk]ρ,
       pkt  = NTVA[pk]ρ in
   is-Array-type(upkt)   ∧
   is-Packed-type(pkt)   ∧
   let mk-Array-type(dt,rt) = upkt in
   is-same(rt,pkt)       ∧
   is-assignment-compatible(dt,NTE[i]ρ)

is-required-function-correspondence: Id × Actual-parm-list →
                                                  Static-env → Bool
is-required-function-correspondence(id,apl) ≜
   len apl=1 ∧
   is-Expression(apl[1]) ∧ WFE[apl[1]]ρ ∧
   let nt = NTE[apl[1]]ρ in
   cases id:
      "abs","sqr"                          → nt ∈ {Integer,Real}
      "sin","cos","exp","ln","sqrt","arctan",
      "trunc","round"                      → nt = Real
      "ord","succ","pred"                  → is-ordinal(nt)ρ
      "chr","odd"                          → nt = Integer
      "eof"                                → is-File-type(nt) ∨
                                             is-Packed-type(nt) ∧
                                             is-File-type(s-type(nt))
      "eoln"                               → nt = Text
```

```
is-wf-function-procedure: Parameters × Block → Static-env → Bool
is-wf-function-procedure(parms,block)ρ ≜
   let mk-Parameters(ids,type) = parms in
   is-unique-declarations(ids,type,block) ∧
   is-unique-labels(block)                ∧
   let ρ1 = erase(all-local-ids(ids,type,nil),{})ρ in
   (∀id∈domtype)(WFDP[type(id)]ρ1)        ∧
   let ρ2 = erase(all-local-ids(<>,[],block),{})ρ1 in
   let ρ3 = merge-variables([id ↦ t | id∈domtype]) ∧
              (type(id) = mk-Var-formal-parameter(t)              ∨
               type(id) = mk-Value-formal-parameter(t)            ∨
               type(id) = mk-Var-array-formal-parameter(t)        ∨
               type(id) = mk-Value-array-formal-parameter(t)])ρ2 in
   let ρ4 = merge-procedures([id ↦ t | id∈domtype]) ∧
               type(id) = mk-Procedure-formal-parameter(t)])ρ3 in
   let ρ5 = merge-functions([id ↦ t | id∈domtype])  ∧
               type(id) = mk-Function-formal-parameter(t)])ρ4 in
   WFB[block]ρ4

bounds-of: (Type-id | Conformant-array-schema) → Index-type-spec-set
bounds-of(tid) ≜
   cases tid:
      mk-P-array-schema(its,)   → {its}
      mk-Array-schema(its,ati)  → {its} ∪ bounds-of(ati)
      T                         → {}
```

7.4 DYNAMIC SEMANTICS

7.4.1 Semantic Domains

```
External-values  =  Id m→ (Value | Value*)

Tr               =  State ~→ State × [Label-den]

State            ::  STORE  :(Sc-loc m→ [Sc-value])
                     FILES  :(File-id m→ File)
                     DENV   :(Did m→ Env)
                     VENV   :(Did m→ (Id m→ Variant-inf))
```

Sc-value = *Int* | *Real* | *Bool* | *Char* | *Enumerated* | *Set* | *Pointer*

Enumerated :: *s-value:Id* *s-order:Id*$^+$

Set = *Ordinal-set*

Pointer = *Loc* | NIL-VALUE

File :: *s-buffer :Buffer*
s-lpart :[Value**]*
s-rpart :[Value**]*
s-access :[Mode]

Buffer :: *s-loc :[Loc]*
s-type :Type
s-senv :Storage-env

Mode = inspection | generation

Env = *Id* $\overrightarrow{m}$ *(Den | Did)*

Variant-inf :: *s-varc :Variant-Component*
s-senv :Storage-env

Variant-component = *Constant* $\overrightarrow{m}$ *Record-type*

Den = *Sc-loc* | *Array-den* | *Record-den*
| *Tag-den* | *File-den* | *Pointer-den*
| *Proc-den* | *Fun-den* | *Subrange-den*
| *Label-den* | *Simple-value* | *Allocated-den*

Array-den = *Ordinal* $\overrightarrow{m}$ *Loc*

Constraint: $(\forall ad \in Array\text{-}den)(\forall rx, ry \in \underline{rng}\ ad)(is\text{-}same\text{-}loc\text{-}type(rx, ry))$

Record-den :: *(Id* $\overrightarrow{m}$ *Record-component)*

Tag-den :: *s-loc :Sc-loc*
s-did :Did
s-type :Variant-inf

File-den :: *File-id*

Pointer-den :: *s-loc* :*Sc-loc*
s-type :*Type*
s-senv :*Storage-env*

Proc-den :: $(Actual\text{-}parms^* \times Aid\text{-}set) \overset{\sim}{\to} Tr$

Fun-den :: *s-den*:$(Actual\text{-}parms^* \times Aid\text{-}set \overset{\sim}{\to} (State \to State \times [Label\text{-}den] \times Sc\text{-}value))$
s-result-loc:*Sc-loc*

Actual-parms = *Loc* | *Value* | *Proc-den* | *Fun-den* |
Actual-read-parm | *Actual-write-parm*

Subrange-den :: *s-loc*:*Sc-loc* *s-range*:*Interval*
Label-den :: *Label* [*Aid*]
Allocated-den :: *s-den*:*Record-den*

Value = *Sc-value* | *Array-value* |
Record-value | *Set-value*

Array-value = $Ordinal \overset{m}{\to} Value$
Record-value = $Id \overset{m}{\to} Value$
Set-value = *Ordinal-set*

Actual-read-parm :: *s-loc*:*Sc-loc* *type*:{Integer, Real, Char}

Actual-write-parm :: *s-val* :*Sc-value*
s-width :[*Int*]
s-frac :[*Int*]

Auxiliary Objects

Ordinal = *Int* | *Bool* | *Char* | *Enumerated*

Storage-loc = *Sc-loc* | *Array-den* | *Record-den*
| *Pointer-den* | *Subrange-den*

Loc = *Storage-loc* | *File-den*

Record-component = *Simple-value* | *Tag-den* | *Loc* | *Did*

Block-env :: *s-type* :*Type-env*
s-env :*Env*
s-aid :*Aid-set*

Type-env = *Id* $\overrightarrow{m}$ *Type*
Simple-value = *Ordinal* | *Real*
Storage-env = *Type-env* × *Env*

Aid, Did, File-id, Id, Label, Sc-loc : Disjoint Infinite Sets

Passed-by-denotation = *Var-formal-parameter* |
Function-formal-parameter |
Procedure-formal-parameter |
Var-array-formal-parameter

Passed-by-value = *Value-formal-parameter* |
Value-array-formal-parameter

Array-parameter = *Value-array-formal-parameter* |
Var-array-formal-parameter

7.4.2 Semantic Elaboration Functions

Meaning Function Abbreviations

MP *Meaning of a Program*
MB *Meaning of a Block*
MBD *Meaning of Block Declarations*
MD *Meaning of a Declaration*
MSL *Meaning of a Statement List*
MS *Meaning of a Statement*
MUS *Meaning of an Unlabelled Statement*
E *Meaning of an Expression*

Other Common Abbreviations

arg argument
const constant
decl declaration
exp expression
id identifier
nm name
op operator
parm parameter
st statement
stl *statement-list*

Semantic Functions

```
i-program : Program → (External-values → External-values)
i-program[p](extv) Δ
   (let σ = mk-State([],[],[],[]) in
    MP[p](extv)σ)
```

```
MP : Program → External-values → (State → State × [Label-den] ×
                                                    External-values)
MP[mk-Program(,args,block)](extv) ≜
    (let t      = required-types                                  in
     let ρ      = required-declarations                           in
     let mk-block(decls,stl) = block                              in
     def δ      : MBD[decls]mk-Block-env(t,ρ,{});
     bind[args](extv,δ);
     MSL[stl]δ;
     def result : unbind[args]δ;
     epilogue[decls](δ);
     return(result))

required-types = ["Integer" ↦ Integer,
                  "Boolean" ↦ Boolean,
                  "Real"    ↦ Real,
                  "Char"    ↦ Char,
                  "Text"    ↦ mk-File-type(Char)]

required-declarations =
   ["abs"     ↦ abs-denotation,        "page"    ↦ page-denotation,
    "arctan"  ↦ arctan-denotation,     "pred"    ↦ pred-denotation,
    "chr"     ↦ chr-denotation,        "put"     ↦ put-denotation,
    "cos"     ↦ cos-denotation,        "read"    ↦ read-denotation,
    "dispose" ↦ dispose-denotation,    "readln"  ↦ readln-denotation,
    "eof"     ↦ eof-denotation,        "reset"   ↦ reset-denotation,
    "eoln"    ↦ eoln-denotation,       "rewrite" ↦ rewrite-denotation,
    "exp"     ↦ exp-denotation,        "round"   ↦ round-denotation,
    "false"   ↦ false                  "sin"     ↦ sin-denotation,
    "get"     ↦ get-denotation,        "sqr"     ↦ sqr-denotation,
    "input"   ↦ mk-File-den(finput),   "sqrt"    ↦ sqrt-denotation,
    "maxint"  ↦ maxint-denotation,     "succ"    ↦ succ-denotation,
    "new"     ↦ new-denotation,        "true"    ↦ true
    "odd"     ↦ odd-denotation,        "trunc"   ↦ trunc-denotation,
    "ord"     ↦ ord-denotation,        "unpack"  ↦ unpack-denotation,
    "output"  ↦ mk-File-den(foutput)   "write"   ↦ write-denotation,
    "pack"    ↦ pack-denotation,       "writeln" ↦ writeln-denotation]
```

Note: When referencing the translated identifiers (members of the set *Id*), quotes are used (e.g. *"Integer"* for the translation of the identifier *Integer*.)

```
bind : Id-set → External-values × Block-env =>
bind[ids](extv,mk-Block-env(t,ρ,)) ≜
  for all id∈ids do
      if ρ(id)∈File-den
         then let mk-File-den(fid) = ρ(id)        in
              let buffer =
                  if id="input" ∨ id="output"
                     then mk-Buffer(nil,mk-File-type(Char,(t,ρ))
                     else buffer-of(ρ(id))        in
              let lp,rp ∈ Value* s.t. lp^rp=extv(id) in
              FILES := c FILES + [fid ↦ mk-File(buffer,lp,rp,nil)];
              (id="input"  → reset(ρ(id)),
               id="output" → rewrite(ρ(id)))
         else assign(ρ(id),extv(id))


unbind: Id-set → Block-env => External-values
unbind[ids](δ) ≜ [id ↦ unbind-val[id]δ | id∈ids]


unbind-val: Id → Block-env => (Value | Value*)
unbind-val[id](mk-Block-env(,ρ,)) ≜
   if ρ(id)∈File-den
      then let mk-File-den(fid) = ρ(id)          in
           def mk-File(,lp,rp,) : (c FILES)(fid)  ;
           return (lp^rp)
      else contents(ρ(id))


MB: Block × Block-env =>
MB[mk-block(decls,stl)]δ ≜
   def δ': MBD[decls]δ;
   always epilogue[decls]δ' in MSL[stl]δ'


epilogue: Declarations → Block-env =>
epilogue[decls]mk-Block-env(,ρ,) ≜
   let vdecls = s-variables(decls)    in
   let vdens  = {ρ(x) | x∈domvdecls}  in
   deallocate(vdens)
```

```
MBD: Declarations → Block-env => Block-env
MBD[mk-Declarations(labs,constm,typem,varm,procm,funm)]δ ≜
   let mk-block-env(t,ρ,cas) = δ in
   let aid∈Aid - cas in
   let t' = t + [id ↦ typem(id) | id ∈ dom typem] in
   def ρ' : ρ + ([id ↦ mk-Label-den(id,aid) | id ∈ labs] ∪
                 [id ↦ MD[constm(id)](t',ρ') | id ∈ dom constm] ∪
                 [id ↦ MD[varm(id)](t',ρ') | id ∈ dom varm] ∪
                 [id ↦ MD[procm(id)](t',ρ') | id ∈ dom procm] ∪
                 [id ↦ MD[funm(id)](t',ρ') | id ∈ dom funm] ∪
                 enumerated-values(rng varm ∪ rng typem,t');
   return(mk-Block-env(t',ρ',cas ∪ {aid}))

MSL: Statement* → Block-env =>
MSL[stl]δ ≜
  let ρ = s-env(δ) in
  tixe [ρ(lab) ↦ cue-i-statement-list[lab,stl]δ | lab∈labels-of[stl]] in
  i-statement-list[stl]δ

cue-i-statement-list: Label × Statement* → Block-env =>
cue-i-statement-list[lab,stl]δ ≜
   let n = index(lab,stl) in
   for i=n to lenstl do MS[s-st(stl[i])]δ

MS: Statement → Block-env =>
MS[mk-Statement(lab,st)]δ ≜
   let ρ = s-env(δ) in
   tixe [l ↦ MUS[st]δ | l ∈ if lab=nil then {} else {l}] in
   MUS[st]δ

MUS: Unlabelled-statement → Block-env =>

MUS[mk-Compound-statement(stl)]δ ≜ i-statement-list[stl]δ

i-statement-list: Statement* → Block-env =>
i-statement-list[stl]δ ≜
   tixe [lab ↦ cue-i-statement-list[lab,stl]δ | is-dcont(lab,stl)] in
   for i=1 to len stl do MS[s-st(stl[i])]δ
```

```
MUS[mk-Assignment-statement(target,exp)]δ ≜
   def rhs: E[exp]δ;
   def lhs: e-left-reference[target]δ
   assign(lhs,rhs)

MUS[mk-Procedure-statement(id,apl)]δ ≜
   def denl : <e-actual-parameter[apl[i]]δ | 1≤i≤lenapl>;
   let f    = s-env(δ)(id)
   in f(denl,s-aid(δ))

MUS[mk-If-statement(exp,th,el)]δ ≜
   def v: E[exp]δ;
   if v then MS[th]δ else if el≠nil then MS[el]δ else I

MUS[mk-Case-statement(exp,ccl)]δ ≜
   def cv: E[exp]δ;
   if cv∈union{s-cs(ccl(i)) | i∈indsccl}
      then let s=(Δi)(c∈s-cs(cll(i))) in MS[ccl[i]]δ
      else error

MUS[mk-While-statement(exp,st)]δ ≜ while E[exp]δ do MS[st]δ

MUS[mk-Repeat-statement(stl,exp)]δ ≜
   MSL[stl]δ; while E[exp]δ do i-statement-list[stl]δ

MUS[mk-Local-goto(id)]δ ≜ exit(id)

MUS[mk-Nonlocal-goto(id)]δ ≜ exit(s-env(δ)(id))

MUS[mk-For-statement(id,from,direction,to,st)]δ ≜
   let next    = if direction=TO then succ else pred in
   def initial : E[from]δ;
   def final   : E[to]δ;
   if ftest(initial,final,direction)
      then (let control = mk-Variable-access(id) in
            assign(control,initial);
            MS[st]δ)
      else I;
   while contents(control)≠final do
      (assign(control,next(control));
       MS[st]δ)
```

```
ftest: Sc-value × Sc-value × {TO,DOWNTO} → Bool
ftest(v1,v2,dir) ≜
    (dir = TO     ↦ v1≤v2
     dir = DOWNTO ↦ v1≥v2)

MUS[mk-With-statement(vs,st)]δ ≜
    def wρ : e-left-reference[vs]δ;
    let mk-Block-env(t,ρ,aid) = δ        in
    let δ' = mk-Block-env(t,ρ+wρ,aid)    in
    MS[st]δ'

e-left-reference: Target → Block-env → Den
e-left-reference[d]δ ≜
    (dεVariable-access → e-reference(d,δ,lhs-apply)
     dεFunction-id     → let fden = s-env(δ)(s-id(d))   in
                         s-result-loc(fden))

assign:  Den × Value =>
assign(den,value) ≜
   cases den:
      mk-Array-den(d)                →
         for all iεdom d do assign(d(i),value(i)),
      mk-Record-den(d)               →
         for all idεdom d do assign(lhs-apply(d,id),value(id)),
      mk-Subrange-den(loc,range)     →
         if s-low(range)≤value≤s-high(range)
            then assign(loc,value)
            else error,
      mk-Tag-den(loc,did,vinf)       →
         if contents(loc)≠value
            then let mk-Variant-inf(vc,senv) = vinf in
                 deallocate({did});
                 assign(loc,value);
                 DENV := c DENV + [did ↦ MD[vc(value)](senv)]
            else I,
      mk-Pointer-den(loc)            → assign(loc,value),
      T    → if denεdom c STORE
                then STORE := c STORE + [den ↦ value]
                else error

MUS[NULL-STATEMENT] ≜ I
```

```
E: Expression → Block-env => Value

E[mk-Variable-expression(va)]δ ≜
   def den : e-reference[va](δ,rhs-apply);
   if ¬(den∈Allocated-den) then contents(den) else error

E[mk-Constant-expression(exp)]δ ≜
   (cases exp:
      mk-Real-constant(r)     → represent(r),
      mk-Integer-constant(i)  → test(i),
      mk-Char-constant(c)     → return(c),
      mk-Id-constant(i)       → return(s-env(δ)(i)),
      mk-String-constant(s)   → return([i → s(i) | i∈inds s]),
      NIL-VALUE               → return(NIL-VALUE)

E[mk-Function-designator(id,apl)]δ ≜
   def denl : <e-actual-parameter[apl[i]]δ | 1≤i≤len apl>;
   let f    =  s-env(δ)(id)  in
   def v    : f(denl,s-aid(δ));
   return(v)

E[mk-Prefix-expression(op,exp)]δ ≜
   def value: E[exp]δ;
   apply-prefix-operation(op,value)

E[mk-Infix-expression(lexp,op,rexp)]δ ≜
   def lvalue : E[lexp]δ;
   def rvalue : E[rexp]δ;
   apply-infix-operation(lvalue,op,rvalue)

E[mk-Set-constructor(ml)]δ ≜ union{e-member(ml(i),δ) | i∈indsml}

apply-prefix-operation: Prefix-opr × Value => Value
apply-prefix-operation(op,value) ≜
   cases op:
      integer-plus,real-plus → return(value)
      integer-minus          → return(-value)
      real-minus             → represent(-value)
      not                    → return(¬value)
```

```
apply-infix-operation: Value × Infix-op × Value => Value
apply-infix-operation(v,op,w) ≜
   cases op:
      real-add     → represent(v + w)
      real-sub     → represent(v - w)
      real-mult    → represent(v * w)
      real-div     → if w ≠ 0
                     then represent(v / w)
                     else error
      integer-add  → test(v + w)
      integer-sub  → test(v - w)
      integer-mult → test(v * w)
      integer-div  → if w≠0
                     then let d be s.t. (∃r∈Integer)(0≤r<w∧v=d*w+r)
                          in test(d)
                     else error
      integer-mod  → if w≠0
                     then let r be s.t. 0≤r<w ∧
                                        (∃d∈Integer)(d>0 ∧ v=d*w+r)
                          in test(r)
                     else error
      lt           → return(v < w)
      le           → return(v ≤ w)
      eq           → return(v = w)
      ne           → return(v ≠ w)
      ge           → return(v ≥ w)
      gt           → return(v > w)
      union        → return(v ∪ w)
      intersection → return(v ∩ w)
      difference   → return(v - w)
      super-set    → return(v ⊇ w)
      sub-set      → return(v ⊆ w)
      in           → return(v ∈ w)
      and          → return(v ∧ w)
      or           → return(v ∨ w)
```

The six relational operators $<$, $\leq$, $=$, $\neq$, $\geq$, $>$ have been extended to the sets *Bool*, *Char*, and *Enumerated* as follows:

Bool	is given the order: *false*, *true*
Char	is given an implementation-defined order.
Enumerated	are ordered according to the associated value list.

```
e-member: Set-constructor → Block-env => Ordinal-set
e-member[m]δ ≜
   if m∈Expression
      then {E[m]δ}
      else let mk-Interval(l,h) = m in
           def lv : E[l]δ;
           def hv : E[h]δ;
           return( {x | lv≤x≤hv} )

represent: Real => Real
represent(r) ≜
   if -maxreal<r<-minreal ∨ minreal<r<-maxreal ∨ r=0
      then return(an implementation defined approximation of r)
      else error

test: Int => Int
test(i) ≜ if -maxint≤r≤maxint then return(i) else error

e-reference: Variable-access → Block-env × (Env × Id → Den) => Den
e-reference[va](δ,apply) ≜
   cases va:
      mk-Index-variable(ar,exp) →
         (def mp    : e-reference[ar](δ,apply);
          def index : E[exp]δ;
          if index ∈ dom mp then return(mp(index)) else error),
      mk-Field-designator(re,id) →
         (def r  : e-reference[re](δ,apply);
          let mp = if r∈Allocated-den then s-den(r) else r in
          apply(mp,id)),
      mk-Reference-variable(rv) →
         (contents(e-reference[rv](δ,apply))),
      mk-Buffer-variable(br) →
          def fid  : e-reference[rv](δ,apply);
          def file : (c FILES)(fid);
          return(s-loc(s-buffer(file))),
      T → apply(s-env(δ),id)

rhs-apply: Env × Id => Den
rhs-apply(m,id) ≜ if id∈dom m then let r = m(id) in
                                   r∈Den → return(r),
                                   r∈Did → rhs-apply((c DENV)(r),id))
                               else error
```

```
lhs-apply: Record-component × Id => Den
lhs(m,id) ≜ if id∈dom m then let r = m(id) in
                                (r∈Den → return(r),
                                 r∈Did → cont-apply(r,id))
                             else error

cont-apply: Did × Id => Den
cont-apply(did,id) ≜
   def m : (c VENV)(did) + (c DENV)(did);
   let r = m(id) in
  (r ∈ Den          → return(r),
   r ∈ Did          →  cont-apply(r,id),
   r ∈ Variant-inf  →  (set-up(did,id); cont-apply(did,id)))

set-up: Den × Id =>
set-up(did,id) ≜
   def mk-Variant-inf(vc,senv) : (c VENV)(did)(id);
   deallocate(rng((c DENV)(did)));
   let ntag∈Constant be s.t. id∈ids-of(vc(ntag)) in
   DENV := c DENV + [did ↦ MD[vc(ntag)]senv]

contents: Loc => Value
contents(d) ≜
 cases d:
   mk-Array-den(ad)       → [i ↦ contents°ad(id) | i∈domad] ,
   mk-Record-den(rd)      → [id ↦ contents°rhs-apply(rd,id) | id∈domrd] ,
   mk-Pointer-den(loc,,)  → contents(loc),
   mk-Subrange-den(loc,)  → contents(loc),
   mk-Tag-den(loc,,)      → contents(loc),
   T                      → if d∈dom(c STORE)
                            then (c STORE)(d)
                            else error

e-actual-parameter: Actual-parm → Block-env => Actual-parms
e-actual-parameter[ap]δ ≜
 (ap ∈ Variable-access       → e-left-reference[ap]ρ,
  ap ∈ Expression            → E[ap]δ,
  ap = mk-Function-parm(id)  → return(s-env(δ)(id)),
  ap = mk-Procedure-parm(id) → return(s-env(δ)(id)),
  ap ∈ Write-parm            → return(e-write-parm(ap,δ)),
  ap ∈ Read-parm             → return(e-read-parm(ap,δ))
```

```
e-write-parm: Write-parm → Block-env => Actual-write-parm
e-write-parm[ap]δ ≜
   (let exp = s-expr(ap)  in
    def ev  : E[exp]δ;
    cases ap:
       mk-Default-write-parm()        →
           mk-Actual-write-parm(ev,nil,nil)
       mk-Width-write-parm(,e1)       →
           def v1 : E[e1]δ;
           mk-Actual-write-parm(ev,v1,nil)
       mk-Fixed-write-parm(,e1,e2)    →
           def v1 : E[e1]δ;
           def v2 : E[e2]δ;
           mk-Actual-write-parm(ev,v1,v2)

e-read-parm: Read-parm → Block-env => Actual-read-parm
e-read-parm[ap]δ ≜
   let d   = s-target(ap) in
   let t   = s-type(ap)   in
   def loc : e-left-reference[l]δ;
   mk-Actual-read-parm(loc,t)

MD: Type → Storage-env =>Den  Unless otherwise stated

MD[mk-Constant(cn)](t,ρ) ≜
   cases cn:
      mk-Prefix-const(sign,id) → def v = MD[mk-Constant(id)](t,ρ);
                                 if sign=PLUS then v else - v,
      mk-String-const(sc)      → sc,
      mk-Real-constant(r)      → represent(r),
      mk-Integer-constant(i)   → test(i),
      mk-Char-constant(c)      → c,
      mk-Id-constant(id)       → ρ(id)

MD[mk-Type-id(id)](t,ρ)          ≜ MD[t(id)](t,ρ)

MD[mk-Enumerated-type(idl)](t,ρ) ≜ generate-sc-loc()

MD[mk-Record-type(fp,vp)](t,ρ) ≜
   def f : e-fixed-part(fp,(t,ρ));
   def v : e-variant-part(vp,(t,ρ));
   return(f∪v)
```

```
MD[mk-Subrange-type(fc,lc)](t,ρ) ≜
    let fv = ρ(fc) in
    let lv = ρ(lc) in
    def loc : generate-sc-loc();
    return(mk-Subrange-den(loc,mk-interval(fv,lv)))

MD[mk-Array-type(dt,rt)](t,ρ) ≜
    def array : elements(dt,(t,ρ));
    return([d ↦ MD[rt](t,ρ) | d ∈ array])

elements: Type × Storage-env => Ordinal-set
elements(type,(t,ρ)) ≜
    let atype = type-of(type) in
   (atype ∈ Enumerated-type  → let {dt} = collect-values(atype,t) in
                               return (elems dt),
    atype ∈ Subrange-type    → let mk-Subrange(f,l) = atype in
                               def fc : MD[f](t,ρ);
                               def lc : MD[l](t,ρ);
                               return({x | x∈values(fc) ∧ fc≤x≤lc}))

MD[mk-File-type(ft)](t,ρ) ≜
    let buffer = mk-Buffer(nil,ft,(t,ρ))      in
    let file   = mk-File(buffer,nil,nil,nil) in
    def fid ∈ File-id - dom c FILES;
    FILES := c FILES + [fid ↦ file];
    return(mk-File-den(fid))

MD[mk-Set-type(dt)](t,ρ)        ≜ generate-sc-loc()

MD[mk-Packed-type(type)](t,ρ)   ≜ MD[type](t,ρ)

MD[mk-Reference-type(tp)](t,ρ) ≜
    def loc : generate-sc-loc();
    return(mk-Pointer-den(loc,tp,(t,δ)))

MD[rt](t,ρ) ≜ generate-sc-loc()

e-fixed-part: (Id → Type-id) × Storage-env => Den
e-fixed-part(fp,(t,ρ)) ≜ [id ↦ MD[fp(id)](t,ρ) | id∈domfp]
```

```
e-variant-part: Variant-part × Storage-env => Den
e-variant-part(vp,(t,ρ)) Δ
   if vp=nil then return([ ])
   else def did ∈ Did - dom c DENV;
        DENV := c DENV + [did ↦ [ ]];
        def td : e-tag-part(vp,did,(t,ρ))
        let v = [id ↦ did | id∈ids-of(s-evp(vp))] in
        return(td∪v)

e-tag-part: Variant-part × Did × Storage-env => Den
e-tag-part(vp,did,(t,ρ)) Δ
   let mk-Variant-part(tid,ttype,vc) = vp in
   let vinf = mk-Variant-inf(vc,(t,ρ)) in
   if tid≠nil then def tden : MD[ttype](t,ρ);
                   return([id ↦ mk-Tag-den(tden,did,vinf)] )
   else let venv = build-venv(vinf) in
        VENV := c VENV + [did ↦ venv];
        return([ ])

build-venv: Variant-Inf → (Id ⇗ Variant-component)
build-venv(vi) Δ merge{build-each-venv(rt,vi) | rt∈rngs-varc(vi)}

build-each-venv: Record-type × Variant-inf → (Id → Variant-inf)
build-each-venv(rt,vi) Δ
   let mk-Record-type(fp,vp) = rt in
       [id ↦ vi | id∈domfp] +
        if vp=nil then [ ]
        else let mk-Variant-part(tid,,lvc) = vp in
             if tid=nil then build-venv(mk-Variant-inf(lvc,s-senv(vi))
             else [tid ↦ vi]

generate-sc-loc: => Sc-loc
generate-sc-loc() Δ def loc ∈ Sc-loc - used-storage();
                  STORE := c STORE ∪ [loc ↦ nil];
                  return(loc)

MD[mk-Procedure(parms,block)](t,ρ) Δ
   let f(apl,cas) = def ρ' : ρ + build-parm-env(parms,apl,(t,ρ));
                    MB[block](mk-Block-env(t,ρ',cas));
                    deallocate-parameter-locs[parms]ρ'
   in f
```

```
deallocate-parameter-locs: Parameters → Env =>
deallocate-parameter-locs[parms]ρ ≜
   deallocate({ρ(id) | id∈elems s-ids(parms)})

MD[mk-Function(parms,block,return)](t,ρ) ≜
   let f(apl,cas) =
      def ρ'     : ρ + build-parm-env(parms,apl,(t,ρ'));
      def rloc   : MD[return](t,ρ);
      i-block[block](mk-Block-env(t,ρ',cas);
      def result : contents(rloc);
      deallocate-parameter-and-return-locs[parms](rloc)ρ';
      return(result)
   in mk-Function-den(f,rloc)

deallocate-parameter-and-return-locs: Parameters → Den → Env =>
deallocate-parameter-and-return-locs[parms](rloc)ρ ≜
   deallocate({ρ(id) | id∈elems s-ids(parms)}∪{rloc})

build-parm-env: Parameters × Actual-parm* × Storage-env → Env
build-parm-env(mk-parameters(fpl,type),apl,(t,ρ)) ≜
   let idl = conc fpl in
   [idl[i] ↦ apl(i)
           | i∈indsapl ∧ type(idl[i])∈Passed-by-denotation]   ∪
   [idl(i) ↦ e-value-parameter(apl[i],type(idl[i]),(t',ρ))
           | i∈indsapl ∧ type(idl[i])∈Passed-by-value]        ∪
   merge{set-bounds(apl(i),cas)
           | i∈indsapl ∧ type(idl[i])∈Array-parameter ∧
             cas=s-schema(type(idl[i]))}

e-value-parameter: Value × Type × Storage-env => Den
e-value-parameter(val,type,(t,ρ)) ≜
   cases type:
      mk-Value-formal-parameter(pt)        →
         def loc : MD(pt)(t,ρ);
         assign(loc,value);
         return(loc)
      mk-Value-array-formal-parameter(pt) →
         let at = construct-array-type(pt,ρ) in
         e-value-parameters(val,at,(t,ρ))
```

```
construct-array-type: Conformant-array-schema × env → Array-type
construct-array-type(cas,ρ) Δ
   let its = s-ind(cas)                                  in
   let mk-Index-type-spec(lid,hid,type) = its            in
   let l   = ρ(lid)                                      in
   let h   = ρ(hid)                                      in
   let dt  = mk-Subrange-type(l,h)                       in
   cases cas: mk-P-array-schema(,type) → mk-Array-type(dt,type)
              mk-Array-schema(,type)   →
                 let rt = if type∈Conformant-array-schema
                          then construct-array-type(type,δ)
                          else type
                 in mk-Array-type(dt,rt)

set-bounds: Actual-parameter × Conformant-array-schema → Env
set-bounds(d,cas) Δ
   let its = s-ind(cas)                                  in
   let mk-Index-type-spec(lid,hid,)    = its             in
   let lbound = mins(dom d)                              in
   let hbound = maxs(dom d)                              in
   let m = [lid ↦ lbound, hid ↦ hbound]                  in
   cases cas: mk-Array-schema(,type)   →
                 if type∈Conformant-array-schema
                    then let nl∈rng d in set-bounds(nl,type) + m
                    else m,
              mk-P-array-schema(,type) → m

enumerated-values: Type-set × Type-env → (Id → Simple-value)
enumerated-values(types,t) Δ
   let tev = union{collect-values(type,t) | type∈types} in
   [id → mk-Enumerated(id,idl) | id∈elemsidl ∧ idl∈tev]

collect-values: Type × Type-env → Id+-set
collect-values(type,t) Δ
   cases type:
      mk-Type-id(id)          → collect-values(t(id),t)
      mk-Enumerated-type(idl) → {idl}
      mk-File-type(ft)        → collect-values(ft)
      mk-Set-type(st)         → collect-values(st)
      mk-Packed-type(pt)      → collect-values(pt)
```

```
mk-Record-type(fp,vp)    →
   union{collect-values(tp) | (∃x∈fp)(tp=s-type(x))} ∪
   if vp=nil then {}
   else union{collect-values(tp) | tp∈rng s-evp(vp)}
mk-Array-type(dt,rt)     → collect-values(dt) ∪ collect-values(rt)
mk-Reference-type(rt)    → collect-values(rt)
T                        → {}
```

Denotations for Standard Procedures and Functions

The denotations for the functions and procedures *abs, arctan, cos, exp, maxint, odd, pack, readin, round, sin, sqr, sqrt, trunc* and *unpack* are Pascal programs that return the appropriate values. The denotations for the other functions and procedures are all of the form:

```
xxx-denotation ≜ let xxx(arguments) in xxx
```

where the *xxx* is given below:

chr

```
chr: Int → Char
chr(x) = an implementation-defined character
```

dispose

```
dispose: Pointer-den[Value*] =>
dispose(pden,vl) =
   if vl=nil
      then let mk-Pointer-den(loc,,)=pden in deallocate({contents(loc)})
      else check-tags(pden,vl); dispose(pden,nil)

check-tags: Pointer-den × Value* → Bool
check-tags(p,vl) ≜
   let mk-Pointer-den(loc,type,) = p          in
   let rd = allocated-type(type,vl)           in
   def mk-Allocated-den(locs) : contents(loc);
   rd = domlocs
```

```
allocated-type: Type × Value* → Id-set
allocated-type(type,vl) ≜
   if vl=<>
      then ids-of(type)
      else let mk-Record-type(fp,vp) = type in
           ids-of(mk-Record-type(fp,nil)) ∪
           (let mk-Variant-part(id,,evp) = vp in
            if id=nil then {id}
            else {id} ∪ allocated-type(evp(hdvl),tlvl))
```

eof

```
eof: File-den => Bool
eof(mk-File-den(fid)) ≜
   if is-file-defined(fid)
      then s-rpart((c FILES)(fid))=<>
      else error
```

eoln

```
eoln: File-den => Bool
eoln(mk-File-den(fid)) ≜
   if ¬eof(mk-File-den(fid)) ∧ is-text(fid) ∧ is-file-defined(fid)
      then hds-rpart((c FILES)(fid))=end-of-line
      else error
```

get

```
get: File-den =>
get(mk-File-den(fid)) ≜
   if pre-get(fid)
      then def mk-File(buffer,lp,rp,mode) : (c FILES)(fid);
           def buffer'                    : re-allocate(buffer);
           FILES := c FILES + [fid ↦ mk-File(buffer',lp^<hd rp>,
                    tl rp,inspection];
           assign(buffer',hd rp)
      else error
```

```
pre-get: File-id => Bool
pre-get(fid) ≜
    is-file-defined(fid) ∧
    (def mk-File(,,rp,mode) : (c FILES)(fid);
     rp=<> ∧ mode=inspection )

is-file-defined: File-id => Bool
is-file-defined(fid) ≜
    def mk-File(,lp,rp,) : (c FILES)(fid);
    lp≠nil ∧ rp≠nil
```

new

```
new: Pointer-den × [Value*] =>
new(pden,vl) =
    let mk-Pointer-den(ploc,type,senv) = pden in
    def den : allocate-new(type,senv,vl);
    let rden = if vl=nil then den else mk-Allocated-den(den) in
    STORE := c STORE + [ploc ↦ rden]

allocate-new: Type × Store-env × Value* => Den
allocate-new(type,senv,vl) ≜
    if vl=nil
       then MD⟦type⟧(senv)
       else let mk-Record-type(fp,vp) = type in
            MD(fp)(senv)
            +(let mk-Variant-part(id,ttype,evp) = vp in
              if id=nil then [] else [id ↦ hdvl]
              + allocate-new(evp(hdvl),senv,tlvl))
```

ord

```
ord: Value → Integer
ord(v) ≜ v ∈ Integer    → v,
         v ∈ Boolean    → if v then 1 else 0
         v ∈ Char       → let i∈Integer be s.t. v=chr(i) in i
         v ∈ Enumerated → let mk-Enumerated(id,idl) = v in
                          (∆i∈indsidl)(idl[i]=id)-1
```

page

```
page: File-den =>
page(mk-File-den(fid)) ≜ if pre-put(fid) ∧ is-text(fid)
                         then def mk-File(,lp,,) : (c FILES)(fid);
                              if lp(len lp)≠end-of-line
                                 then put-char(fid,end-of-line);
                                 else I;
                              put-char(fid,end-of-page)
                         else error
```

pred

```
pred: Ordinal → Ordinal
pred(x) ≜ if mins(values(x))<x
          then (∆r∈values(x))(ord(r)=ord(x)-1)
          else undefined
```

put

```
put: File-den =>
put(mk-File-den(fid)) ≜
   if pre-put(fid)
      then def ch : contents(buffer-of(fid));
           put-char(fid,ch)
      else error

put-char: File-Id × Char =>
put-char(fid,ch) ≜
   def mk-File(buffer,lp,rp,mode) : (c FILES)(fid);
   def buffer'                    : re-allocate(buffer);
   FILES := c FILES + [fid ↦ mk-File(buffer',lp^<ch>,<>,generation)

pre-put: File-id => Bool
pre-put(fid) ≜
   def mk-File(,lp,rp,mode) : (c FILES)(fid);
   return(lp≠nil ∧ rp=<> ∧ mode=generation)
```

read

```
read: File-Id × Actual-read-parm =>
read(fid,arp) Δ
   let mk-Actual-read-parm(loc,type) = arp  in
   cases type:
      Integer → def v: get-value(fid,Integer);
                assign(loc,v)
      Char    → def buffer : buffer-of(fid);
                assign(loc,contents(buffer));
                get(fid)
      Real    → def v : get-value(fid,Real);
                assign(loc,v)

get-value: File-id × {Integer,Real} => Value
get-value(fid,type) =
   def mk-File(buffer,lp,rp,) : (c FILES)(fid);
   let i,j∈{0:lenrp} be s.t.
       i≥1 ∧
       check-syntax(subl(rp,i,j),type) ∧
       (¬∃k>j)(check-syntax(subl(rp,i,k),type) ∧
       (∀n∈{1:i-1})(rp(n)∈{BLANK,end-of-line) in
   if j≠0
      then def buffer' : re-allocate(buffer);
           FILES := (c FILES + [fid ↦ mk-File(buffer',
                                              lp^subl(rp,i,j),
                                              rest(rp,j+1),
                                              generation)];
           assign(buffer',rp(j+1));
           return(numeric-rep-of(subl(rp,i,j))
      else error

check-syntax: Char-list × {Integer,Real} → Bool
```

The result of *check-syntax(clist,type)* is *TRUE* if the characters in *clist* correspond to the syntax of a signed-integer, if *type* is Integer, or to a signed-number, if *type* is Real. Otherwise it is *FALSE*.

reset

```
reset: File-den =>
reset(mk-File-den(fid)) Δ
   if is-file-defined(fid)
      then def mk-File(buffer,lp,rp,) : (c FILES)(fid);
           let file-value = complete-file(lp^rp,s-type(buffer)) in
           def buffer'    : re-allocate(buffer);
           FILES := c FILES + [fid ↦ mk-File(buffer',<>,file-value,
                                                 inspection)]
           if file-value≠<> then assign(buffer',hd file-value) else I
      else I

complete-file: Value-list × Type → Value-list
complete-file(vl,type) Δ
   type≠Char ∨ vl=<>          → vl
   vl(len vl)=end-of-line     → vl
   T                          → vl^end-of-line
```

rewrite

```
rewrite: File-den =>
rewrite(mk-file-den(fid)) Δ
   def buffer  : buffer-of(mk-File-den(fid));
   def buffer' : re-allocate(buffer);
   FILES := (c FILES + [fid ↦ mk-File(buffer',<>,<>,generation)]
```

succ

```
succ: Ordinal → Ordinal
succ(x) Δ if x<maxs(values(x))
             then (Δr∈values(x))(ord(r)=ord(x)+1)
             else undefined
```

write

```
write: File-den × Actual-write-parameter =>
write(mk-File-den(fid),awp) ≜
   let val = s-value(awp) in
   let s∈String be s.t.
      (cases val:
          val ∈ Char    → is-character-rep(s,awp)
          val ∈ Integer → is-integer-rep(s,awp)
          val ∈ Real    → is-real-rep(s,awp)
          val ∈ String  → is-string-rep(s,awp)
          val ∈ Boolean → is-boolean-rep(s,awp)) in
   if pre-put(fid) ∧ is-text(fid)
      then def mk-File(buffer,lp,rp,mode) : c FILES;
           def buffer' : re-allocate(buffer);
           FILES := c FILES + [fid ↦ mk-File(buffer',lp^s,
                                             <>,generation)]
      else error
```

The functions: *is-character-rep*, *is-integer-rep*, *is-real-rep*, *is-string-rep*, *is-boolean-rep*, *is-fixed-rep*, and *is-floating-rep*
all have type: *String* × *Actual-write-parm* → *Bool*

```
is-character-rep(s,awp) ≜
   let field-width = check-width(awp)        in
   let value       = s-val(awp)              in
   let lead        = max(0,field-width - 1)  in
   let spaces      = {1:lead}                in
   lens=field-width ∧ (∀i∈spaces)(s[i]=BLANK) ∧ s[lens]=value

is-integer-rep(s,awp) ≜
   let field-width = check-width(awp)                          in
   let value       = s-val(awp)                                in
   let lead        = max(0,field-width - (intsize(value) + 1)) in
   let zeros       = {1:lead}                                  in
   lens=field-width ∧ (∀i∈zeros)(s[i]=BLANK) ∧
   (if value≥0 then s[lead + 1]=BLANK else s[lead + 1]=MINUS) ∧
   numeric-rep-of(rest(s,lead + 2))=abs(value)
```

```
is-real-rep(s,awp) Δ
    s-frac(awp)=nil → is-floating-rep(s,awp),T → is-fixed-rep(s,awp)

is-string-rep(s,awp) Δ
    let field-width = check-width(awp)                    in
    let value       = s-val(awp)                          in
    let lead        = max(0,field-width - lenvalue)       in
    let spaces      = {1:lead}                            in
    lens=field-width ∧ (∀i∈spaces)(s[i]=BLANK) ∧
    rest(s,lead+1)=subl(value,1,min(lenvalue,lens - lead))

is-floating-rep(s,awp) Δ
    let field-width = check-width(awp)                    in
    let value       = s-val(awp)                          in
    let dec-places  = field-width - expdigits - 5         in
    let x,y ∈ Int be s:t: if value=0 then x=0 ∧ y=0
                                     else 10>x≥1 ∧
                                          dec-places-of(x)=dec-places ∧
                                          is-approximation(value,x,y) in
    lens=field-width ∧
    (if value>0 then s[1]=BLANK else s[1]=MINUS) ∧
    numeric-rep-of(s[2])=trunc(x) ∧
    s[3]=POINT ∧
    numeric-rep-of(subl(s,4,dec-places))=x-trunc(x) ∧
    s[4 + dec-places]=EXPON ∧
    s[5 + dec-places]=(y≥0 → PLUS,T → USCORE) ∧
    numeric-rep-of(rest(s,6 + dec-places))=y
```

Comment: *expdigits* is an implementation-defined integer value representing the number of digit characters written in an exponent.

```
is-fixed-rep(s,awp) Δ
    let (field-width,frac-width) = check-width(awp)                in
    let value                    = s-val(awp)                      in
    let sign                     = if value>0 then 0 else 1        in
    let min-width                = intsize(value)+frac-width+sign+1 in
    let x∈Int be s.t. if value=0 then x=0
                      else dec-place-of(x)=frac-width ∧
                           is-approximation(value,x,0)             in
    let lead                     = max(0,field-width - min-width)  in
    let zeros                    = {1:lead}                        in
```

```
    lens=field-with ∧ (∀i∈zeros)(s[i]=BLANK) ∧
    value<0 ⊃ s[lead + 1]=MINUS ∧
    numeric-rep-of(subl(s,lead+sign+1,intsize(x))=trunc(x) ∧
    s[lead+sign+intsize(x)+1))= POINT ∧
    numeric-rep-of(rest(s,lead+sign+intsize(x)+2))=x - trunc(x)

is-boolean-rep(s,awp) ≜
    let field-width = check-width(awp)                          in
    let value       = s-val(awp)                                in
    let result      = if value then <T,R,U,E> else <F,A,L,S,E>  in
    is-string-rep(s,mk-Actual-write-parm(result,field-width,nil))

check-width: Actual-write-parm → (Int | Int × Int)
check-width(mk-Actual-write-parm(v,tw,fw)) ≜
    v∈Char          → let aw = if tw=nil then 1 else tw in
                      if aw≥1 then aw
                      else undefined,
    v∈Integer       → let aw = if tw=nil then intw else tw in
                      if aw≥1 then max(intsize(v)+1,aw)
                      else undefined,
    v∈Real ∧ fw=nil → let aw = if tw=nil then realw else tw in
                      if aw≥1 then max(expdigits+6,aw)
                      else undefined,
    v∈Real ∧ fw≠nil → let sign = if v>0 then 0 else 1          in
                      let mchs = sign+intsize(v)+1+dec-digits-of(v) in
                      let aw   = if tw=nil then mchs else tw    in
                      if aw≥1 ∧ fw≥1 then (max(aw,mchs),fw)
                      else undefined,
    v∈Bool          → let aw = if tw=nil then boolw else tw in
                      if aw≥1 then aw else undefined
    v∈String        → let aw = if tw=nil then lenv else tw in
                      if aw≥1 then aw else undefined
```

Comment: *intw*, *realw*, and *boolw* are implementation-defined integer values representing the number of characters written out for *Integer*, *Real* or *Boolean*, respectively.

```
numeric-rep-of: Char* → Int | Real
```

numeric-rep-of converts a *Char-list* into a implementation defined value which the character string is a representation of.

```
intsize: Real → Int
intsize(x) ≜ if x=0 then 1
             else let r be s.t. 10**(r-1)≤abs(x)<10**r in r

is-an-approximation: Real × Real × Integer → Bool
is-an-approximation(x,y,z) ≜
   rep-of(x-0.5*10**-dec-places-of(x),y)
   ≤ abs(r) <
   rep-of(x+0.5*10**+dec-places-of(x),y)

rep-of: Real × Integer → Real
rep-of(x,y) ≜ x*10**y
```

writeln

```
writeln: File-den =>
writeln(mk-File-den(fid)) ≜
    if pre-put(fid) ∧ is-text(fid)
       then put-char(fid,end-of-line)
       else error
```

7.4.3 Auxiliary Functions

```
deallocate: Den-set =>
deallocate(ds) =
   def locs : union{sc-locs-of(d) | d∈ds} ;
   STORE := c STORE \ locs;
   def files : union{files-of(d) | d∈ds} ;
   FILES := c FILES \ files;
   def dids : {x | x∈Did∧(∃y∈rng((cDENV)(x)))(sc-locs-of(y)⊆locs)} ;
   DENV  := c DENV \ dids;
   VENV  := c VENV \ dids

sc-locs-of: Storage-loc => Sc-loc-set
sc-locs-of(x) ≜
   (x ∈ Sc-loc        → {x},
    x ∈ Array-den     → union{sc-locs-of(r) | r ∈ rng x},
    x ∈ Record-den    → union{sc-locs-of(r) | r ∈ rng x},
    x ∈ File-den      → def mk-File(buf,,,,) : (c FILE)(x);
                        sc-locs-of(buf),
```

```
      x ∈ Pointer-den   → let mk-Pointer-den(l,,) = x in {l},
      x ∈ Subrange-den  → let mk-Subrange-den(l,) = x in {l},
      x ∈ Tag-den       → let mk-Tag-den(l,,) = x in {l},
      x ∈ Did           → union{sc-locs-of(r) | r ∈ rng((c DENV)(x))},
      x = nil           → {})

files-of: Structural-den => Fil-id-set
files-of(x) ≜
   (x ∈ File-den    → {s-id(x)},
    x ∈ Array-den   → union{files-of(r) | r ∈ rng(x)},
    x ∈ Record-den  → union{files-of(r) | r ∈ rng(x)},
    x ∈ Did         → union{files-of(r) | r ∈ rng((c DENV)(x))},
    T               → {})

buffer-of: File-den => Den
buffer-of(mk-File-den(fid)) ≜
   def mk-File(buffer,,,) : (c FILES)(fid); s-loc(buffer)

labels-of: Statement* → Label-set
labels-of(stl) = {s-label(stl(i)) | i∈indsstl} - {nil}

used-storage: => Sc-loc-set
used-storage() ≜
   dom c STORE ∪ union{sc-locs-of(y) | y∈rng(c STORE) ∩ Storage-loc}

re-allocate: Buffer => Buffer
re-allocate(buffer) ≜
   let mk-Buffer(loc,type,senv) = buffer in
   if loc≠nil then de-allocate({loc}) else I;
   def loc' : MD[type](senv);
   mk-Buffer(loc',type,senv)

ids-of: Record-type → Id-set
ids-of(rt) ≜
   let mk-Record-type(fp,vp) = rc in
   if vp=nil
      then dom fp
      else let mk-Variant-part(tag,,vc) = vp in
           dom(fp)∪ (if tag=nil then {} else{tag})
           ∪ union{ids-of(rts) | rts∈rngvc}
```

```
is-dcont: Label × Statement-list → Bool
is-dcont(lab,sl) ≜ (∃i∈inds sl)(lab=s-label(sl[i]))

index: Label × Statement-list → Nat
index(lab,stl) ≜ (Δi∈inds stl)(lab = stl[i])

values: Ordinal → Ordinal-set
values(x) ≜ x∈Integer    → {i | -maxint≤i≤maxint },
            x∈Bool       → 𝒫 Bool,
            x∈Char       → 𝒫 Char,
            x∈Enumerated → let mk-Enumerated(,idl) = x in
                           {mk-Enumerated(e,idl) | e∈elemsidl}

type-of: Type × Type-env → Type
type-of(t,tp) ≜ if t∈Type-id then type-of(tp(s-id(t)),tp) else t

is-text: File-id => Bool
is-text(fid) ≜ def file : (c FILES)(fid);
               let mk-Buffer(,t,(tp,)) = s-buffer(file) in
               type-of(t,tp)=mk-File-type(Char)

maxs: Ordinal-set → Ordinal
maxs(s)     ≜ let e∈s in if cards=1 then e else max(e,maxs(s-{e}))
pre-maxs(s) ≜ s≠{}

mins: Ordinal-set → Ordinal
mins(s)     ≜ let e∈s in if cards=1 then e else min(e,mins(s-{e}))
pre-mins(s) ≜ s≠{}
```

Comment: *min* and *max* can be extended to *Ordinals* using the extended definition of ≥.

```
subl: X* × Int × Int → X*
subl(l,i,n)     ≜ l=<> ∨ n=0 → <>
                  i=1        → hdl^subl(tll,1,n-1)
                  T          → subl(tll, i-1, n
pre-subl(l,,n) ≜ i∈indsl ∧ n≥0

rest: X* × Int → X*
rest(l,i)     ≜ subl(l,i,lenl - i+1)
pre-rest(l,i) ≜ i∈indsl
```

```
is-same-loc-type: Loc × Loc → Bool
is-same-loc-type(l1,l2) ≜
    (l1,l2∈File-den)  ∨ (l1,l2∈Sc-loc) ∨ (l1,l2∈Array-den) ∨
    (l1,l2∈Record-den ∧ doml1=doml2)                        ∨
    (l1,l2∈Subrange-den ∧ s-range(l1)=s-range(l2)
```

7.5 INDEXES

7.5.1 Static Semantics Functions and Objects

7.5.2 Dynamic Semantics Functions and Objects