IBM

IBM United Kingdom
Laboratories Limited

A Formal Definition of Algol 60

C.D. Allen
D.N. Chapman
C.B. Jones

August 1972

215-8052-0

Technical Report T.R.12.105

Unrestricted

A Formal Definition of Algol 60

C.D. Allen
D.N. Chapman
C.B. Jones

Unrestricted

August 1972 215-8052-0

IBM United Kingdom Laboratories Limited
Hursley Park
Winchester Hampshire

A FORMAL DEFINITION OF ALGOL 60

C.D. Allen
D.N. Chapman
C.B. Jones

Programming Technology Dept.
IBM Product Test Laboratory
Hursley
Winchester
England

ABSTRACT

This paper contains a formal definition of the semantics of the ECMA subset of Algol 60. The definition is written as a series of recursive functions. Although some of the ideas of the "Vienna Definition Language" are adopted, the use of the control mechanism is specifically avoided.

CONTENTS

1. BACKGROUND

This paper contains a formal definition of the ECMA subset of Algol 60. The definition is written as a family of recursive functions. Specific details of the notation used are given in Chapter 3 below. Before coming to this it is, perhaps, worth reviewing the motivation for this work.

First it must be clear that the hope is to illustrate language definition ideas rather than to provide a definition of an already widely understood language.

The authors were lead to undertake this definition as a result of an interest in correctness proofs for compilers. Some of the work on this topic (begun Ref. 8, see also ref. 9) has been based on "Vienna-style" definitions (generally referred to as VDL definitions). In spite of the relative success of this line of work, it was clear that certain difficulties were the result of the definition method. In particular the way in which the sequencing of operations was handled by the control complicates reasoning about the order of operations. This and other problems are discussed in the next Chapter. Ideas existed for circumventing some of these problems (e.g. ref. 7). The definition given below shows how these ideas are worked out in detail on a reasonable size language.

There is, of course, an existing formal definition of Algol 60 in the Vienna-style (ref. 6). The reader will find that differences exist between that and the current definition other than those caused by the discussed changes of method. Most of these can be explained by an attempt to come closer to the present authors' understanding of the "spirit of Algol". The best example of this is the use in the current definition of a fairly direct form of the copy-rule, whereas ref. 6 had exhibited the relation to the VDL definitions of PL/1 (ref. 5) by using the environment mechanism from that model. The existence of such options in the construction of a definition is a result of the use of the constructive definition method and the choices made in the current definition may not be universally liked.

The language defined in Ref. 6 is full Algol (i.e. ref. 1). The current definition is of the ECMA subset plus recursion (see ref. 2). This has two advantages. On the one hand it avoids some of the less clearly defined areas of Algol (e.g. own) and, on the other hand, it defines a language which is more oriented towards static "compilation".

The next Chapter contains a discussion of the main changes made in the current definition with respect to ref. 5. The remainder of the report is a self-contained definition. Chapter 3 describes the notation used in the formal definition. Chapter 4 describes some of the ideas built into the formal definition. Chapter 5 presents, on facing pages, prose and formal descriptions of the language following the structure of ref. 1. The prose description is an amalgam of refs. 1 and 2 with "comments" prompted by ref. 3. The Appendix contains a cross reference index of functions and predicates as an aid to the reader.

The major divisions of the paper have been called Chapters to avoid confusion with the sections of the Algol Report. References into the Algol Report are of the form A.R. followed by a section number. References consisting of only a section number are to the formal definition (i.e. the left hand pages of Chapter 5).

2. DISCUSSION

This section contains a discussion of the main changes to the definition style which have been made with respect to ref. 6.

The Control Component

The control component of a VDL style definition is described in, for example, ref. 4. It is basically a record of what remains to be done in the interpretation. This is achieved by storing instruction names and their arguments in the control. In the VDL definitions the control is made an actual state component. This has two interesting consequences: being a normal object it can be of a tree form; also explicit changes are possible.

The tree form of the control provides a good model for arbitrary ordering between elementary operations. This is achieved by providing a control function (LAMBDA) which selects the next operation from amongst the terminal nodes of the tree. This models in a fairly natural way the arbitrary order of sub-expression evaluation prescribed for Algol.

Without arbitrary ordering the control would exhibit an obvious stack type behaviour; the generalisation in the tree case is also not difficult to envisage. The ability to explicitly change the control complicates this picture.

The jump concept, present in most languages as a local goto (see below for discussion of non-local goto's), has the effect of cutting across the obvious recursive structure of evaluation instructions. This can be modelled by an explicit change to the control whose effect is to delete those entries which corresponded to instructions which looked as though they were to be executed. This explicit change obviously invalidates the simple generalised picture of the stack behaviour of the machine.

"Freezing" of the arbitrary order of evaluation (e.g. entry to a function referenced in expression evaluation) can be modelled by storing away an old control component and installing a new one whose last action will be to re-instate the stored control. Such a model, whilst adequate, causes further distortion of the intuitive stack behaviour since non-local goto commands may now discard stored control components. Another problem caused by storing the old control is that the control can no longer be used to return a value from a function reference (cf. ref. 6).

Perhaps it is unfair to describe the next item as a "consequence", but the possibility of explicitly changing the control can be over-utilized as a short cut. Such a short cut is used in ref. 6 to handle for statements: the result is one of the less clear parts of that definition.

Finally, the necessity to use two levels of function (i.e. the LAMBDA control function and the instructions) was unfortunate, and the description of this (see ref. 5) difficult to understand. It should be admitted that the mechanism now offered is also far from simple. The present authors put more emphasis on a "purely functional" definition and leave the decision on comprehensibility to the reader.

The problems with the control have long been understood (cf. ref. 10). Ideas for modelling the required language features without resort to a control have also been suggested: Ref. 7 shows how the abnormal termination of blocks can be modelled without explicitly changing the control. The basic idea is to anticipate the possibility of an "abnormal" return and to return an indicator which can be tested in the invoking routine. These tests then control the abnormal returns in an orderly fashion rather than goto's "taking the machine by surprise". The problem of the disappointingly large numbers of places where the abnormal value can be returned in the Algol definition is mitigated by the uniformity with which the condition is handled.

The related question of goto commands into structured statements, such as compound and conditional statements, is handled by "cue" functions which set up the appropriate continuation without interpreting those parts of the program which are jumped over.

The problem of representing the required degree of arbitrary ordering in expression evaluation is handled by having a function which accepts the whole expression tree and performs an arbitrary selection of the next operation. Of course, this mirrors the exact role played by the LAMBDA function. The difference, which the present authors consider relevant, is that the tree is part of an abstract program rather than a control containing function names. It is also important to observe that it is only the order of access of variables and function calls which is not defined, the order of evaluation of expressions is defined as left to right. This situation is modelled closely below by having separate functions which access values and which apply operators to values.

Use of the copy rule

Use is made of a form of the copy rule. This results in an extremely natural model of by name parameter passing.

Elimination of the state

VDL models have a state which is global and can be changed by side effect. A fairly mechanical rewriting into functions which have this state as part of their domain and range is possible. It is then clear which parts of the state are not required by or cannot be modified by each function: the domains/ranges can be reduced to only the required items.

A related point is the existence of values of local variables after their blocks cease to exist. Explicit delete operations appear in the current definition which remove the values left in limbo in ref. 6.

Error Detection

There is a class of defects which force a program to be considered invalid for which it is simple to devise static tests (e.g. undeclared variables). If a definition checks for these dynamically the question is left open as to whether a program is in error if such a defect occurs in an "unexecuted" portion. If formal definitions are to provide a reference point for implementations a suitable rule might be "any implementation should produce (one of) the result(s) of the definition unless the latter results in error - in which case the implementation is not further constrained". Clearly such a rule is untenable if compiler checkable (i.e. static) defects such as undeclared variables do not result in error. It is equally clear that one could never define all special cases of errors which are statically checkable. The break-point adopted below is basically to check those things which rely only on symbol matching and omit those checks which, in general, rely on values of symbols. Thus items not statically checked in the current definition are:

Upper bound vs lower bound of arrays
Validity of procedure body after by name substitution
Incompatibility of formal/actual parameters
Applicability of array bounds
Goto into for statement

However, declaration of variables and applicability of operators to their operands is checked by stating the properties of abstract programs which the translator will pass.

Array Values

There is no requirement in Algol to consider array values as structured (cf. ref. 6) . They are treated in the current definition as pairings of integer lists with simple values, where the integer lists correspond to the subscript list.

Abstract Syntax

The abstract program has been made to correspond more closely to the concrete program by leaving labels on the statements and by not using identifiers as selectors. Objects for which identifiers were used as selectors are now shown as sets. This, however, does lead to certain difficulties caused by the lack of text selectors.

Changes to the defined language

Apart from the change to the ECMA subset plus recursion, a few points of detail have been changed from the language defined in ref. 6.

Ref. 6 copies the <array list> onto each array identifier declared. Since this may result in a different number of invocations of any functions referenced from that expected, this decision has not been followed.

Since it is not entirely clear if it is possible to have formal parameters of type procedure passed by value it has been banned in the current definition (see A.R. 4.7.5.4).

There would appear to be no justification for specifying a left to right order of evaluation for subscript expressions occurring other than on the left of an assignment: the restriction of ref. 6 is relaxed in the current definition.

3. NOTATION

Introduction

The definition is a set of functions and predicates which together define the interpretation of any valid ALGOL 60 program. This Chapter describes the notation used to define functions and predicates.

The notation is that of refs. 4 and 11, with slight changes in typography.

In this section, an underlined word indicates that the meaning of that particular term is given at this point.

Objects and Selectors

The basic elements operated on throughout the definition are called objects. Objects may be elementary, that is having no structure, or composite. Composite objects have structure in that they are composed of parts which may be extracted or replaced by the appropriate functions.

The elementary objects used in this definition include members of the following classes:

ids: these are objects used in place of the identifiers of the ALGOL 60 source text,
values: these are the usual arithmetic and boolean values,
the empty list <>: see below under Lists,
sets: various kinds of sets are used; any set is an elementary object, including the empty set { }.

Several other specific elementary objects are used; they are denoted by names spelt in capitals. Thus REAL, INTG, BOOL are the elementary objects corresponding to the ALGOL 60 keywords real, integer, Boolean.

Construction of Composite Objects

Composite objects are constructed out of elementary objects and function names using the function μ_0 as follows. Let OBA, OBB, OBC be three elementary objects to be made parts of a composite object. To extract these three parts of the composite object we need three function names (functions intended for this purpose are given names beginning "s-", and are called

selectors). Let s-1, s-2, s-3 be the three selectors chosen to operate on the object being constructed. These are paired with the elementary object they are to select (notation <s-1:OBA>) and the pairs given to the function μ_0 as arguments, thus:

$$\mu_0(\langle s\text{-}1:OBA\rangle,\langle s\text{-}2:OBB\rangle,\langle s\text{-}3:OBC\rangle) \ .$$

The resulting object, say ob1, has three parts:

$$\begin{aligned} OBA &= s\text{-}1(ob1) \\ OBB &= s\text{-}2(ob1) \\ OBC &= s\text{-}3(ob1) \ . \end{aligned}$$

Note that this definition of ob1 has also defined new values of each of the functions s-1, s-2 and s-3, namely those given above. (Note also the use of colons as separators in the pairs within the μ_0 arguments. Elsewhere a comma is used for this purpose -- see below under Lists.)

Composite objects may also be used as the second member of an argument to μ_0. Thus if ob2 and ob3 are two further objects, either elementary or composite, then we might define ob4 as:

$$ob4 = \mu_0(\langle s\text{-}a:ob1\rangle,\langle s\text{-}b:ob2\rangle,\langle s\text{-}c:ob3\rangle) \ .$$

Then:

$$\begin{aligned} s\text{-}a(ob4) &= ob1 \\ s\text{-}b(ob4) &= ob2 \\ s\text{-}c(ob4) &= ob3 \ . \end{aligned}$$

The arguments to μ_0 may be specified as a set, provided the order in which they are taken does not affect the final object. Thus in the definition of tail (section 1, Lists) we use:

$$\mu_0(\{\langle elem(i):elem(i+1)(list)\rangle \mid 1 \le i \le length(list)\})$$

Here the arguments are the pairs <elem(i):elem(i+1)(list)> where i is in the appropriate range. (Note that the function μ_0 applied to an empty set in this way gives Ω, see below.)

Selector functions, and in fact any functions, may be composed by the functional composition operator, $\bullet$. The composition of two functions f1 and f2 is defined as follows:

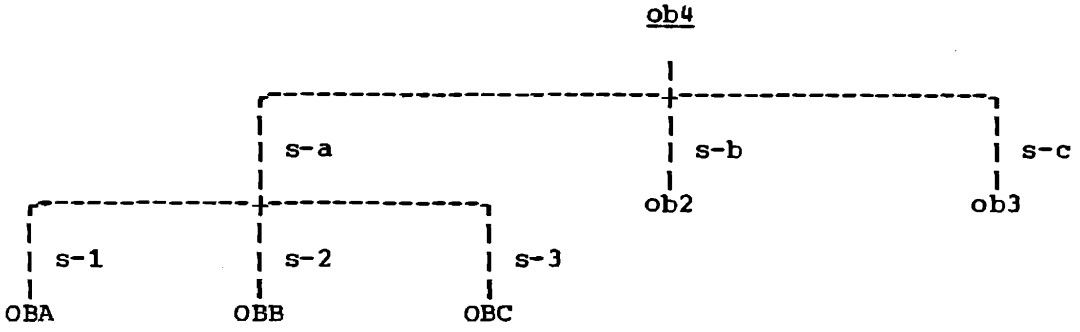
$$f2 \bullet f1(x) = f2(f1(x)) \ .$$

Note that in this use, f1 must be defined for argument x, and if it then returns y, f2 must be defined for this argument. Using this notation, we have from the definitions of ob4 and ob1:

$$s\text{-}1 \bullet s\text{-}a(ob4) = s\text{-}1(s\text{-}a(ob4)) = s\text{-}1(ob1) = OBA\ .$$

Composite selectors are functions formed by composition of selectors in this way. Selectors which are not composite are called elementary selectors.

Composite objects may be thought of as trees, having each branch from one node to another named by a selector, and having elementary objects at each of the terminal nodes. Thus ob4 may be shown diagramatically as a tree as follows.



If ob2 and ob3 are composite, then there will be further structure beneath these nodes.

Values of selector functions for specific arguments thus become defined by the definition of composite objects using them. If a selector is used with an argument for which no value has been defined in this way, then the value is taken to be a special object called Ω. Any selector applied to any elementary object gives Ω (with the sole exception of the identity selector, I, see below). With the example objects defined above:

$$s\text{-}1(ob4) = \Omega$$
$$s\text{-}a(ob1) = \Omega$$
$$s\text{-}a \bullet s\text{-}a(ob4) = \Omega\ .$$

Note that any selector applied to Ω gives Ω, by definition.

The selectors used in the arguments to μ_0 must all be different, otherwise a unique value for the selectors would not be defined if they were applied to the constructed object. For example

$$ob = \mu_0(\langle s\text{-}x:ob1\rangle, \langle s\text{-}x:ob2\rangle, \langle s\text{-}z:ob3\rangle)$$

does not give a defined value for s-x(ob) -- it might be ob1 or ob2. Hence this formula is invalid. However the same selector may be re-used at a different level of the tree, e.g.:

$$ob5 = \mu_0(\langle s\text{-}1:ob1\rangle, \langle s\text{-}2:ob2\rangle, \langle s\text{-}3:ob3\rangle)$$

is quite legitimate, giving:

$$s\text{-}1(ob5) = ob1$$
$$s\text{-}1 \bullet s\text{-}1(ob5) = OBA\ .$$

The identity function, I, is included in the selector functions, since it may be successfully applied to any object. It is defined by:

$$I(x) = x$$

-- where x is any argument whatever. When applied to an elementary object it returns that object, not Ω. (This is the only selector that does not give Ω when applied to an elementary object.)

Modification of Objects

Composite objects may be modified by replacing the sub-tree below one node (itself an object) by another object; additions may be made by replacing the empty sub-tree below one node by an object. The function used is μ, with arguments:

i) the object to be modified,
ii) a pair -- as for μ_0 -- consisting of the selector identifying the part of the object to be replaced and the object to be inserted at this point.

For example, with the objects defined above, if

$$ob6 = \mu(ob4; \langle s\text{-}b:ob7\rangle)$$

ob6 has the structure of ob4 with ob2 (= s-b(ob4)) replaced by ob7. Thus

$$s\text{-}b(ob6) = ob7\ .$$

(Note the semi-colon following the object to be modified.)

Predicates

Meanings are given to the operations & (and), and ∨ (or) of predicate calculus when some of their operands are undefined. In the notation used here (explained below under Definition of Functions) they may be defined as:

```
x & y =

     cases: x
     FALSE: FALSE
     TRUE:  y

x ∨ y =

     cases: x
     TRUE:  TRUE
     FALSE: y
```

The effect of the conditional definitions is that logical expressions become undefined if, working from left to right, an undefined operand is encountered before the value of the expression has been determined. Once a value has been determined, the fact that one or more of the remaining operands is undefined does not make the expression undefined.

The remaining operators are defined as usual in terms of & and ∨ where necessary, using these meanings of & and ∨. The operators are:

¬	not	(undefined if its operand is undefined)
&	and	
∨	or	
⊃	implies	(x ⊃ y = ¬x ∨ y)
≡	is equivalent to	(x ≡ y = (x & y) ∨ (¬x & ¬y))
∃	existential quantifier	
∀	universal quantifier	
i	(iota) see below .	

The i operator is used in the same way as a quantifier, e.g.:

```
(i x) (is-pred(x)) .
```

This expression denotes the unique x such that is-pred(x) is true, if one and only one such x exists; otherwise its value is undefined.

In quantified expressions, the bound variable (that following the quantifier symbol in the opening parentheses) is frequently a name that has a corresponding predicate, e.g. (∀sel)(...) with the corresponding predicate is-sel. In such cases the bound variable is assumed to take only values satisfying the predicate. Thus in the example given above, sel is assumed to take only values satisfying is-sel; this is equivalent to adding to the quantified expression inside the second pair of parentheses a term is-sel(sel) & ... If the quantified variable is i or j, then the corresponding predicate is understood to be is-intg-val.

The quantifiers ∀ and ∃ are only used on expressions that are defined for all values of the quantified variable in the range of that variable. This range may be indicated by the name of the variable, as mentioned above; if not, the range is unrestricted.

Predicates are functions whose values are truth-values (either TRUE or FALSE). They may thus be relational expressions, such as x≤1, or expressions built of other predicates using the above operators. Generally they are given names beginning with "is-".

Predicate expressions are sometimes used; they have one of the following forms, defined as below.

(i) (is-pred1 ∨ is-pred2)(x) = is-pred1(x) ∨ is-pred2(x) ,
(ii) (is-pred1 & is-pred2)(x) = is-pred1(x) & is-pred2(x) .

Each elementary object whose name is spelt in capitals has an associated predicate whose name is the name of the object prefixed with is-. Thus

is-REAL(x) = (x = REAL)
is-INTG(x) = (x = INTG) , etc..

For composite objects, many predicates are defined in terms of their selectors and predicates applying to their parts. For example:

```
is-real-op(x) =

    is-REAL•s-type(x) & is-real-val•s-value(x) &
      (is-CONST ∨ is-Ω)•s-const(x) &
      ((sel ≠ s-type & sel ≠ s-value & sel ≠ s-const) ⊃
                                          is-Ω•sel(x))
```

defines is-real-op to be true of x if and only if x has two or three parts, one selected by s-type and satisfying is-REAL (i.e. it is the elementary object REAL), one selected by s-value and satisfying the predicate is-real-val, and a third which may not be present (indicated by the is-Ω

alternative of the predicate) but if it is it must satisfy the predicate is-CONST. Note that x may not have any other parts. This kind of definition is so frequently used that an abbreviated notation is used for it. The general form is:

```
is-x = (<s-a:is-a>,
        <s-b:is-b>,
        ...
        <s-n:is-n>) .
```

This defines is-x to be true of an object if and only if it has parts selected by s-a, s-b, ...,s-n satisfying the predicates is-a,is-b,...,is-n respectively. No other parts may be present; the parts mentioned may be missing if Ω satisfies the relevant predicate. Thus the definition of is-real-op given above takes the form:

```
is-real-op = (<s-type:is-REAL>,
              <s-value:is-real-val>,
              <s-const:is-CONST ∨ is-Ω>) .
```

(See Operands in section 1.) In this style we also use:

```
is-pred1 = (<is-pred2,is-pred3>)
```

to define is-pred1 as true only of pairs whose elements satisfy is-pred2 and is-pred3 respectively, and:

```
is-pred4 = ({is-pred5})
```

to define is-pred4 as true only of sets whose elements satisfy is-pred5.

Lists

Lists (strictly, non-empty lists) are objects constructed with the special selectors elem(1), elem(2),...,elem(n), where n is the number of elements in the list (the length of the list). They satisfy the predicate:

```
is-non-empty-list = (<elem(1):is-el>,<elem(2):is-el>,...
                                          <elem(n):is-el>)
```

for some n, where is-el is false of Ω, but true of any other object.

An ordering is defined on the elements of a list by the integer arguments to elem. The ith element of a list is

```
elem(i)(list) .
```

For i greater than the length of the list, elem(i)(list) = Ω; elem(i) may not be used with i ≤ 0.

The abbreviation:

elem(i,list) for elem(i)(list)

is sometimes used to economise on parentheses.

We also use the notation

<el1,el2,...,eln>

to denote the list with elements el1, el2,...,eln in that order, i.e.

$$\langle el1,el2,...,eln\rangle = \mu_0(\langle elem(1):el1\rangle, \langle elem(2):el2\rangle, ... \langle elem(n):eln\rangle) .$$

Functions head, giving the first element of a list; tail, giving the list with its first element removed -- and the remaining elements renumbered; length, giving the length of the list are defined (see section 1, Lists). The concatenation operator, ", is also defined between two lists. The empty list, <>, is defined to be an elementary object, since elem(i)(<>) = Ω for any positive i; it satisfies the predicate is-<>, which is false for any other object. It also satisfies the predicate is-list, thus

is-list = is-non-empty-list ∨ is-<> .

Any predicate is-pred has an associated predicate is-pred-list, which is true of <> and of any list all of whose elements satisfy is-pred.

Sets

Sets are elementary objects, since although they have elements no structure is defined on them. A set having specific elements a,b,c,...,n may be defined by:

set1 = {a,b,c,...,n}

-- in which the order of the arguments is immaterial. Thus

{a,b,c} = {b,c,a} = {b,a,c} .

The notation { } represents the empty set, having no elements. We may also define sets using the notation:

set2 = {f(x) | is-pred(x) } .

In this case the set is formed by taking all the objects satisfying is-pred, operating on them with the function f, and putting all the results in the set. (Frequently there will be no f, just x, in such a definition implying that the objects satisfying is-pred themselves are to be the members of the set.)

Any predicate is-pred has an associated predicate is-pred-set which is true of any set all of whose elements satisfy is-pred, and is also true of the empty set { }. The empty set satisfies the predicate is-{ }, which is false for any other object. The predicate is-set is true of any set, including the empty set.

The following operators on sets are used with their usual meaning

$\cup$	set union
-	set difference
$\in$	is a member of

Paths and Path-els

The existence of sets embedded in the abstract structure of a program, and the lack of selectors to select elements of a set is inconvenient in specifying some of the contextual constraints on a correct program. To overcome this difficulty, functions called path-els are assumed to exist. A path-el is either a selector or a function , like a selector, from a set to one of its elements. It is assumed that, in a given set in the abstract program, each element of the set may be obtained from the set itself by such a path-el function, distinct path-els giving distinct elements of the set. A path is a composition of path-els, so that any part of a given abstract program may be obtained by applying some path to it. Distinct paths give distinct parts of the program.

Definition of Functions

Definitions of functions are given in the normal way, and also by cases using the following basic form:

```
funct(args) =

     cases: f(args)
     is-pred1: def-1
     is-pred2: def-2
     ...
     is-predn: def-n
```

-- where args may be one or more arguments. This form is to be interpreted in the following way. If is-pred1 is true of f(args), then funct(args) is defined by def-1; if is-pred1 is false of f(args) and is-pred2 is true of f(args) then funct(args) is defined by def-2, etc.. If any of the predicates is-predi, i = 1,...,n, is undefined and none of the preceding predicates is true of f(args), then funct(args) is undefined. If all the predicates are false of f(args), then also funct(args) is undefined. Thus functions defined this way may be partial, i.e. they may not be defined for all values of their arguments. In this definition, the interpretation of a program never becomes undefined in this way. If argument values for which a function is not defined are presented to it, then the program must be in error. In such cases, we write a final case giving "error" as the definition. (Occasionally such a definition may appear as a case other than the last, with the same significance.) This indicates that it is an erroneous program being interpreted if this case of the definition is required.

Definitions of this form may be nested within each other. Thus in the above form, def-1 may again be such a definition.

The args given to the function f may be a subset of the arguments of the main function. If the case depends on the value of an argument itself, and not some value constructed from it, then the function f is omitted, and the predicates are then applied to the single argument. If the predicates are testing for equality with a particular elementary object, then we write the object, OB, rather than the corresponding predicate is-OB. The final predicate may be required to be satisfied in all cases not accepted by the previous predicates; in this case the constant predicate T, true for all arguments (and any number of arguments) may be used. Thus the erroneous case mentioned above is written:

```
T:error .
```

For example consider the definition of arithm-prefix-opr of section 3.3:

```
arithm-prefix-opr(op,opr) =

     cases: s-type(op)
     INTG: cases: opr
           PLUS: op
           MINUS: mk-op(INTG, - s-value(op))
     REAL: mk-op(REAL,real-prefix-value(s-value(op),opr))
```

This is constructed as follows. The definition deals with two basic cases, where s-type(op) satisfies is-INTG or is-REAL. Since these predicates test for equality with the elementary objects INTG or REAL, we write these objects in place of the predicates in the basic form. In the first case, i.e. if s-type(op) = INTG then there are two sub-cases. The first applies if the argument opr is the elementary object PLUS. In this case arithm-prefix-opr(op,opr) is defined to be just the object op. In the second sub-case, if the argument opr is the elementary object MINUS, then arithm-prefix-opr(op,opr) is defined to be the value of the function mk-op applied to arguments INTG and -s-value(op). If s-type(op) is REAL then arithm-prefix-op is defined to be the value of the function mk-op applied to arguments REAL and the value of real-prefix-value(s-value(op),opr). The majority of functions in the definition return more than one object; they may be thought of as returning a list of objects. Where such a composite returned value is referred to, we use the notation (el1,el2,...,eln) rather than the usual notation <el1,el2,...,eln> for such a list.

In many function definitions, the cases depend on the values of more than one object returned by another function, or some combination of predicates applied to more than one of the arguments. In such cases, a single predicate or elementary object determining the case may be replaced by a list of predicates or elementary objects. For example in the definition of cue-int-st-list in section 4.1:

```
cue-int-st-list(targ-sel,t,i,dn,vl) =

     cases: cue-int-st(targ-sel,elem(i,t),dn,vl)
     (vl¹,Ω): int-st-list(t,i+1,dn,vl¹)
     (vl¹,lab¹): cases: lab¹
                 local(lab¹,t): ...
```

the function cue-int-st returns a list of two objects. The first case of the definition is taken if the second of these is Ω; in this case we use the parameter name vl¹ to refer to the first of the returned objects (whatever it may be) in the following definition of the function. (Note that this use of the parameter vl¹ holds only within this case of the definition. This means of introducing a new name to represent something

occurring in the definition is a special case of the abbreviation by means of let clauses described immediately below.) Here we have used one elementary object (Ω) instead of a predicate is-Ω; the fact that the other name in the case definition is neither an elementary object nor a predicate indicates that the corresponding returned object plays no part in the case distinction.

In many function definitions, abbreviations are used to shorten the text. They are introduced by the word let; following this one or more names are defined as meaning some longer expression or name. Thus in the definition of convert-one-sub in section 3.1:

```
convert-one-sub(eb,op) =

    let: v¹ = convert(INTG,op)
    cases: s-lbd(eb)≤v¹≤s-ubd(eb)
    TRUE: v¹
    FALSE: error
```

the v^1 is used as an abbreviation for the expression on the right of the = in the let statement. Again such an abbreviation applies only within the case in which it is defined. Thus in the definition of iterate-for, section 4.6:

```
iterate-for(cvar,for-elem,t,dn,vl) =

    cases: for-elem
    is-arithm-expr:
        cases: eval-expr(for-elem,dn,vl)
        (op¹,vl¹,Ω): let: v = convert(s-type(cvar),op¹)
                          vl² = assign(cvar,v,vl¹)
                     int-st(t,dn,vl²)
        (op¹,vl¹,lab¹): (vl¹,lab¹)
    is-step-until-elem:
        cases: eval-expr(s-init-expr(for-elem),dn,vl)
        (op¹,vl¹,Ω): let: v = convert(s-type(cvar),op¹)
                          vl² = assign(cvar,v,vl¹)
                     iterate-step-until-elem(cvar,
                                             s-step-expr(for-elem),
                                  s-until-expr(for-elem),t,dn,vl²)
        (op¹,vl¹,lab¹): (vl¹,lab¹)
    is-while-elem: iterate-while(cvar,for-elem,t,dn,vl)
```

the abbreviations op^1 and vl^1 are defined differently in the first two cases of for-elem.

In some function definitions, an operation is required to be performed on an element of a set, but which element of the set is arbitrary. (This is necessary if, for instance, the elements of a set are to be operated on in an arbitrary order, not determined by the definition.) Thus the function defined becomes non-deterministic, in that it does not return a uniquely defined result, but rather one of many possible results. A special notation is used to indicate this. For example in the definition of mk-pairs, section 4:

```
mk-pairs(id-set,t,dn) =

     cases: id-set
     { }: { }
     ¬is-{ }: for some id¹ ∈ id-set
              {mk-pair(id¹,t,dn) } ∪ mk-pairs(id-set - {id¹},t,dn)
```

the definition of the second case, where id-set is not empty, requires that some element of id-set is to be used as id^1 in the case definition, but which one is not defined -- any choice will satisfy the definition. In cases such as this, where the set is already given, we use the phrase for some, followed by a requirement on the object denoted by an abbreviation -- in this case that it be an element of the set decl-set.

Each function definition is followed by a clause giving the type of arguments for which it is defined, and the type of values it returns. These types are specified using predicates, which in this use stand for the set of objects satisfying them. The notation

```
type: is-pred1 X is-pred2 → is-pred3 X is-pred4
```

indicates that the function accepts two arguments satisfying is-pred1 and is-pred2 respectively, and returns two objects satisfying is-pred3 and is-pred4 respectively.

The Sequencing Mechanism

The order in which the functions of the definition are evaluated is determined solely by their nesting. Thus int-program (section 4.1) is defined as an application of int-block. In the evaluation of int-block, the function eval-array-decls is evaluated first, since this determines whether or not the values of the other functions will be required. Following this, the functions intr-ids, mk-pairs, change-block, augment-dn, mod-set, int-block-body, seconds, epilogue are evaluated, necessarily in that order since each has as argument the result (or part of the result) of that preceding; the only exception is the pair augment-dn and mod-set, which may be evaluated in either order, since their results are both arguments to int-block-body. The purely functional nature of the definition ensures that the result is the same whichever order is used, since the results of a function evaluation depend only on its arguments, all of which are written explicitly.

Thus the ordering required by the language on the various defined operations must be reflected in the structure of the function definitions.

Two particular styles of definition are used to specify an operation performed on elements of a list in sequence. The simpler one is to define a function - say f1(el,res) - to operate on a single element, and combine this with the results from the preceding elements of the list. This is then used in a recursive definition of a function, say f2, defined as:

```
f2(list,res) =

     cases: list
     <>   : res
     ¬is-<>: let: res' = f1(head(list),res)
             f2(tail(list),res')   .
```

(For example, see iterate-for-list,section 4.6.)

This mechanism does not permit the ordering to depend on the results of any of the functions. In the case of statement sequencing, where exceptions may occur on executing a goto statement, a different mechanism is needed. Here we use an iteration driven by an index to the next statement in the list, as exemplified in int-st-list (section 4.1), and described below.

Normal sequencing through successive statements of a statement list is modelled by int-st-list, with the list and an index, initially 1, as its first two arguments. The effect of execution of the statements is to change the values in vl; int-st-list returns the changed vl as well as an abn component (explained below). The denotations, dn, affect the execution, so dn and the initial values, vl are given to int-st-list as its third and fourth arguments.

Generally, int-st-list has arguments t -- a statement list, i -- the index of a statement in t, dn -- the denotations appropriate to the execution of t, and vl -- the values of known variables after execution of the first i-1 statements of t. The values to be returned are obtained as the result of applying int-st-list to t, i+1, dn, and vl', where vl' is vl with any changes caused by the execution of the ith statement of t; these are obtained by applying int-st to elem(i,t) -- the ith statement, dn, and vl. The recursion of int-st-list in this way produces a sequence of vl's, giving the changes to values resulting from the execution of the statements of t in normal sequence. The recursion terminates when the new value of i is greater than length(t), and the final vl is returned by int-st-list.

If a goto is executed at some point in the sequence, the appropriate application of int-st will return a label in the abn component of its returned value. The vl resulting from further execution is no longer that obtained by interpreting the remainder of the list. If the label is on a statement within the list, an appropriate vl is obtained from int-st-list with an index to the labelled statement; this is obtained via cue-int-st-list and cue-int-st, which ensure that the mechanism to continue normal sequencing from the labelled statement is set up (the labelled statement may be within a nest of compound statements). If the label is not on a statement of the list, execution of this list is terminated with the current vl, and the label is returned to the functions interpreting the surrounding statement list.

The Evaluation of Expressions

In the evaluation of expressions, the order in which arithmetic and boolean operations are performed is well defined. It is reflected, together with any modifications introduced by parentheses, in the form of the abstract text of the expression. However the order in which subexpressions, e.g. in subscript or actual parameter lists, should be evaluated, and values of variables accessed is not defined.

The formal definition of this situation depends on the following two facts. Firstly, since arithmetic operations do not have any side effects, the order in which they are done relative to the other accessing and evaluation operations cannot affect the final result. Secondly, the resolution of

conditional expressions to the then or else expression, and the resolution of switch designators to the appropriate expression from their switch list, make available further text of the expression; to allow the maximum freedom of choice of ordering, these resolutions should be done as early as possible.

In the definition therefore the choice is made to delay arithmetic and boolean operations as long as possible. The function apply, which deals with these, is thereby given a text in which all operands are ops -- i.e. they have been reduced to values. The function reduce-cond-switch deals with conditionals and switch designators; it is used recursively, since the text introduced by one application of it may contain other conditionals or switch designators which may be resolved immediately. The function access deals with the remaining operations in arbitrary order. These comprise evaluation of the following components of the expression: function references, all of whose by name actual parameters have been fully accessed: simple variables: subscripted variables, all of whose subscripts have been fully accessed: array names (appearing in actual parameter lists). Fully accessed means here that their values can be obtained by apply; this function is used where appropriate to obtain the final value of these parts of the expression.

Static Type Checking

The translation makes use of a function desc (section 1, Miscellaneous), which returns a descriptor from the declarations or specifier from the parameter specifications within whose scope a particular appearance of an identifier lies. The first argument to desc is a path to that appearance of the id; the second is the text which contains this appearance and the declaration or formal parameter specification whose scope contains it. Thus the path applied to the text gives the id. The result depends on the context of an appearance of the id, thus a path to the id within the text is required as an argument rather than the id itself. (The same id may be declared or specified more than once within the text, and different descriptors or specifiers may then be appropriate to different appearances of the same id within the same text).

Procedure Invocations and Function Values

Procedure statements are interpreted by int-proc-st (section 4.7). This function uses access to evaluate the by value actual parameters, and proc-access (section 3.2) to achieve the call. The access mechanism (see the Accessing of Variables above) also uses proc-access (section 3.2).

The function proc-access finishes the evaluation of parameters with a final apply, and passes them to activate-proc. This makes various tests on the

matching of actual and formal parameters, and forms a set of pairs of formal parameter identifiers and their replacements. For by value parameters, the replacement is an identifier - different from the formal parameter identifier if this is already known as a non-local variable - to which the value of the actual parameter is given in vl, and the description of the formal parameter is given in dn. (This is achieved in the test of the actual parameter).

The set of pairs is passed to non-type-proc or type-proc as appropriate. The function type-proc establishes a new identifier to receive the returned value (with type of the procedure being called) in dn and uses insert-ret to modify the relevant appearances of the procedure identifier within its body. It then applies non-type-proc (section 4.7) to proceed with the call.

The function non-type-proc uses the change-text function to insert the replacements for the formal parameters in the procedure body, and passes the modified body to int-proc-body. This function then executes the modified procedure body in the usual way.

Note that the formal parameters are local to the procedure; thus they, or any identifiers by which they were replaced, will be deleted from vl at the epilogue of the procedure. However, the identifier installed for the returned value will not be deleted, since it was not included in the list of such identifiers passed to epilogue by activate-proc. Thus type-proc can extract the type and value of this identifier, and return it through activate-proc and proc-access to fn-access, which will return it into the evaluation of the expression in which the function call occurred.

The Copy Rule

Entry to a procedure involves replacing the by name formal parameters by the text of the corresponding actual parameters. If this text contains identifiers that are declared within the procedure, those declared within the procedure are changed to a different, distinct, identifier. In the definition on entry to any block, the declared identifiers are changed to ids different from those already known (i.e. having entries in the current dn) and from all those declared within any internal blocks or procedure descriptors. (Note that label constants are given entries in dn solely to prevent re-use of their id.) This ensures that no conflicts can occur when these new identifiers appear in by name actual parameters passed to the declared procedures. It should be noted that such substitution for by name formal parameters may give rise to syntactically incorrect text; if this is the case the program is in error.

1 DESCRIPTION OF THE REFERENCE LANGUAGE

1.1 FORMALISM FOR THE DESCRIPTION

Introduction

The syntax of <program> is as given in the Algol Report. A string satisfying this syntax is translated into an object which satisfies is-program as defined under the 'Abstract Syntax' headings below. (The notation used is described in the Notation section.) For the main part the translation is obvious, but the Translator also performs some additional checking. This checking is described under the "Translation" headings by stating properties (in the form of implications) which will hold of any object created by the Translator. As a result of this some programs which satisfy the concrete syntax are rejected and their semantics are not defined.

Some extra predicates are defined in the 'Auxiliary Predicate' sections, and are used as abbreviations for collections of abstract syntax predicates. Note that all predicates defined in the abstract syntax and auxiliary predicate sections begin with is-.

The interpretation of an abstract program is then defined in the 'Interpretation' sections as the application of the function int-program to the translated program. The actual definition given of int-program is a trivial one reflecting the fact that strict Algol 60 programs can only cause any observable effects by calling code procedures. This interpretation corresponds to the semantic description of the Algol 60 language in the sense that it defines the result of interpretation (termination or looping and values available to code procedures), not the way of computing this result.

Appended to the function definitions are the following: a list of references to any predicates or functions not contained in the relevant section or in section 1; a description of any error cases; any explanatory notes required; the type of the defined function shown as a domain and range separated by "→" and stated in terms of cartesian products of (sets satisfying) predicates.

The remainder of this section contains the definitions of some predicates and functions used throughout the formal definition.

Objects

(1) is-object =

This predicate is true only of elementary objects (is-set, is-basic-symbol, is-real-val, is-intg-val, is-id, is-<>, and anything of the form is-CAPS. It is false of Ω.

(2) is-ob =

This predicate is true only of elementary or composite objects.

(3) is-selector =

This predicate is true only of elementary selectors. (It is false for I.)

(4) is-sel =

This predicate is true only of elementary or composite selectors (including I).

(5) is-path-el =

A path-el is either a selector or a function from a set to an element of that set. There is some path-el from any set in the abstract form of the program being interpreted to any element of that set.

(6) is-path =

This predicate is true of objects satisfying is-path-el or compositions thereof.

(7) is-set =

This predicate is true of any set, including the empty set, { }.

(8) is-[pred]-set(set) =

is-set(set) & (∀el)(el ∈ set ⊃ is-[pred](el))

note: Any defined predicate name, with the initial is- deleted, may be substituted for [pred] in the above definition. For example

is-ob-set(set) = is-set(set) & (∀el)(el ∈ set ⊃ is-ob(el))

is obtained in this way, using the defined predicate is-ob, 1(2).

type: is-ob → is-bool-val

Pairs

(9) is-pr = (<is-ob,is-ob>)

(10) is-idpr = (<is-id,is-id>)

refs: is-id 2.4

(11) s(e,pr-set) =

cases: e
(∃e-1)(<e,e-1> ∈ pr-set): (i e-1)(<e,e-1> ∈ pr-set)
T: error

error: e is not first element of a pair in pr-set. (Only occurs for uninitialised values.)
type: is-ob X is-pr-set → is-ob

(12) del-set(pr-set,ob-set) =

{<e-1,e-2> | <e-1,e-2> ∈ pr-set & ¬(e-1 ∈ ob-set)}

type: is-pr-set X is-ob-set → is-pr-set

(13) mod-set(pr-set-1,pr-set-2) =

{<e-1,e-2> | (<e-1,e-2> ∈ pr-set-2 ∨
(<e-1,e-2> ∈ pr-set-1 & ¬(<e-1,ob> ∈ pr-set-2)))}

type: is-pr-set X is-pr-set → is-pr-set

(14) firsts(pr-set) =

{ob-1 | <ob-1,ob-2> ∈ pr-set}

type: is-pr-set → is-ob-set

(15) seconds(pr-set) =

{ob-2 | <ob-1,ob-2> ∈ pr-set}

type: is-pr-set → is-ob-set

Denotations

(16) is-type-den = (<s-type:is-type>)

refs: is-type 5.1

(17) is-eb = (<s-lbd:is-intg-val>,
<s-ubd:is-intg-val>)

refs: is-intg-val 2.5

(18) is-array-den = (<s-type:is-type-array>,
<s-bounds:is-eb-list>)

refs: is-type-array 5.2

(19) is-label-den = (<s-type:is-LABEL>,
<s-value:is-id ∨ is-Ω>)

refs: is-id 2.4
note: Label denotations contain a value only for by value parameters and the identifier is that of the actual parameter.

(20) is-switch-den = (<s-switch-list:is-des-expr-list>,
<s-type:is-SWITCH>)

refs: is-des-expr 3.5

(21) is-proc-den = (<s-type:is-type-proc ∨ is-PROC>,
<s-form-par-list:is-id-list>,
<s-spec-pt:is-spec-set>,
<s-value-pt:is-id-set>,
<s-body:is-block ∨ is-code>)

refs: is-type-proc 5.4, is-id 2.4, is-spec 5.4, is-block 4.1, is-code 5.4

(22) is-den = is-type-den ∨ is-array-den ∨ is-proc-den ∨ is-label-den ∨ is-switch-den

(23) is-dn = ({<is-id,is-den>})

refs: is-id 2.4

Values

(24) is-arithm-val = is-real-val ∨ is-intg-val

refs: is-real-val 2.5, is-intg-val 2.5

(25) is-simple-val = is-bool-val ∨ is-arithm-val

refs: is-bool-val 2.2

(26) is-arithm-array-val = ({<is-intg-val-list,is-arithm-val>})

refs: is-intg-val 2.5

(27) is-bool-array-val = ({<is-intg-val-list,is-bool-val>})

refs: is-intg-val 2.5,is-bool-val 2.2

(28) is-array-val = is-arithm-array-val ∨ is-bool-array-val

(29) is-val = is-simple-val ∨ is-array-val

(30) is-vl = ({<is-id,is-val>})

refs: is-id 2.4

Operands

```
(31)   is-arithm-array-op = (<s-type:is-INTG-ARRAY ∨ is-REAL-ARRAY>,
                             <s-bounds:is-eb-list>,
                             <s-value:is-arithm-array-val>)

(32)   is-bool-array-op = (<s-type:is-BOOL-ARRAY>,
                           <s-bounds:is-eb-list>,
                           <s-value:is-bool-array-val>)

(33)   is-array-op = is-arithm-array-op ∨ is-bool-array-op

(34)   is-real-op = (<s-type:is-REAL>,
                     <s-value:is-real-val>,
                     <s-const:is-CONST ∨ is-Ω>)

          refs: is-real-val 2.5


(35)   is-intg-op = (<s-type:is-INTG>,
                     <s-value:is-intg-val>,
                     <s-const:is-CONST ∨ is-Ω>)

          refs: is-intg-val 2.5


(36)   is-bool-op = (<s-type:is-BOOL>,
                     <s-value:is-bool-val>,
                     <s-const:is-CONST ∨ is-Ω>)

          refs: is-bool-val 2.2


(37)   is-arithm-op = is-real-op ∨ is-intg-op

(38)   is-type-op = is-arithm-op ∨ is-bool-op

(39)   is-label-op = (<s-type:is-LABEL>,
                      <s-value:is-id>,
                      <s-const:is-CONST ∨ is-Ω>)

          refs:is-id 2.4


(40)   is-op = is-type-op ∨ is-array-op ∨ is-label-op
```

(41) mk-op(type,v) =

μ_0(<s-type:type>,<s-value:v>)

type: (is-type ∨ is-LABEL) X (is-simple-val ∨ is-id) → is-op

Lists

(42) is-list = is-<> ∨ (<elem(1):is-el>,<elem(2):is-el>,...,<elem(n):is-el>)

note: is-el is true of any object except Ω.

(43) head(list) = elem(1)(list)

type: is-list → is-ob

(44) tail(list) = μ_0({<elem(i):elem(i+1)(list)>|1≤i<length(list)})

type: is-list → is-list

(45) length(list) =

cases: list
<>: 0
¬is-<>: (i j)(elem(j)(list) ≠ Ω & is-Ω(elem(j+1,list)))

type: is-list → is-intg-val

(46) list-1 " list-2 =

μ(list-1;{<elem(length(list-1)+i):elem(i,list-2)> | 1≤i≤length(list-2)})

type: is-list X is-list → is-list

(47) is-[pred]-list(list) =

is-list(list) & (∀i)(1≤i≤length(list) ⊃ is-[pred]•elem(i)(list))

note: Any defined predicate name, with the initial is- deleted, may be substituted for [pred] in the definition above. For example

is-eb-list(list) =

is-list(list) & (∀i)(1≤i≤length(list) ⊃ is-eb•elem(i)(list))

is obtained in this way, using the defined predicate is-eb, 1(17).

type: is-ob → is-bool-val

Builtin functions

(48) sign(x) =

cases: x
x>0: 1
x=0: 0
x<0: -1

type: is-real-val → is-intg-val

(49) entier(x) =

(ι i)(is-intg-val(i) & i≤x<i+1)

refs: is-intg-val 2.5
type: is-real-val → is-intg-val

(50) abs(x) = x * sign(x)

type: is-real-val → is-real-val

Miscellaneous

(51) desc(path,t) =

desc-1(path(t),path,t)

note: is-id(path(t))
type: is-path X is-text → (is-specifier ∨ is-desc ∨ is-label-desc)

(52) desc-1(id,path-el•path,t) =

cases: path-el•path(t)
is-proc-desc: desc-proc(id,path-el•path,t)
is-block: desc-block(id,path-el•path,t)
T: desc-1(id,path,t)

refs: is-proc-desc 5.4, desc-proc 5.4, is-block 4.1, desc-block 4.1
type: is-id X is-path X is-program → (is-specifier ∨ is-desc ∨ is-label-desc)

(53) is-abn = is-id ∨ is-Ω

refs: is-id 2.4

(54) is-intr = is-decl ∨ is-label-decl

refs: is-decl 5

(55) is-lab-selector =

One of the elementary selectors s-st-pt, elem(i), s-then-st, s-else-st, or s-st.

(56) is-lab-sel =

Compositions of lab-selectors (including I).

(57) is-label-decl = (<s-id-set:is-id-set>,
<s-desc:is-label-desc>)

refs: is-id 2.4, is-label-desc 5

(58) is-text(t) = (∃p,path)(is-program(p) & is-path(path) & t=path(p))

refs: is-program 4.1

(59) disj(set-1,set-2) =

¬(∃el)(el ∈ set-1 & el ∈ set-2)

type: is-ob-set X is-ob-set → is-bool-val

(60) main-pt(sel) =

(i sel-1)(is-selector(sel-1) & (∃sel-2)(sel-2•sel-1 =sel))

type: is-sel → is-selector

(61) rest-pt(sel) =

(i sel-1)(sel-1•main-pt(sel) =sel)

type: is-sel → is-sel

(62) is-opt-[pred] = is-[pred] ∨ is-Ω

note: Any defined predicate name, with the initial is- deleted, may be substituted for [pred] in the definition above. For example

is-opt-op = is-op ∨ is-Ω

is obtained in this way, using the defined predicate is-op, 1(40).

Translation

(1) All <labels> other than label parameters are translated into label constants.

Abstract syntax

(2) is-basic-symbol =

The set of symbols which are members of the concrete syntax classes <letter>, <digit>, <logical value>, <delimiters>.

(3) is-type-const = is-arithm-const ∨ is-bool-const

refs: is-arithm-const 2.5, is-bool-const 2.2

(4) is-const = is-type-const ∨ is-label-const

refs: is-label-const 3.5

2.2 DIGITS/LOGICAL VALUES

Abstract syntax

(1) is-bool-val = is-TRUE ᵛ is-FALSE

(2) is-bool-const = (<s-type:is-BOOL>,
<s-value:is-bool-val>,
<s-const:is-CONST>)

2.3 DELIMITERS

Translation

(1) The Translator drops delimiters and comments.

2.4 IDENTIFIERS

Abstract syntax

(1) is-id =

Infinite class of Algol identifiers.

2.5 NUMBERS

Abstract syntax

(1) is-real-val =

Real values are as defined in A.R.

(2) is-intg-val =

Integer values are as defined in A.R.

note: It is assumed that is-intg-val(x) ⊃ is-real-val(x).

(3) is-non-neg-intg-val(x) = is-intg-val(x) & x≥0

(4) is-real-const = (<s-type:is-REAL>,
<s-value:is-real-val>,
<s-const:is-CONST>)

(5) is-intg-const = (<s-type:is-INTG>,
<s-value:is-intg-val>,
<s-const:is-CONST>)

(6) is-non-neg-intg-const = (<s-type:is-INTG>,
<s-value:is-non-neg-intg-val>,
<s-const:is-CONST>)

(7) is-arithm-const = is-real-const ∨ is-intg-const

2.6 STRINGS

Abstract syntax

(1) is-string-elem = is-basic-symbol ∨ is-string

refs: is-basic-symbol 2

(2) is-string = is-string-elem-list

Notes

In order to avoid the necessity of defining a separate class of objects to cover the changed expressions generated by interpretation, the definition of is-expr permits operands to occur as primitive elements.

Translation

(1) Only constants (a subset of operands) can occur:

is-expr(e) & is-sel(sel) & is-op•sel(e) ⊃ is-const•sel(e)

refs: is-const 2

Abstract syntax

(2) is-expr = is-arithm-expr ∨ is-bool-expr ∨ is-des-expr

refs: is-arithm-expr 3.3, is-bool-expr 3.4, is-des-expr 3.5

Auxiliary predicates

(3) is-op-expr(t) =

is-expr(t) & (¬(∃sel)((is-funct-ref ∨ is-var ∨ is-switch-des)(sel(t))) &
(is-cond-expr•sel(t) ⊃ is-op•s-decision•sel(t) &
(is-TRUE•s-value•s-decision•sel(t) ⊃ is-op-expr•s-then-expr•sel(t)) &
(is-FALSE•s-value•s-decision•sel(t) ⊃ is-op-expr•s-else-expr•sel(t))))

refs: is-funct-ref 3.2, is-var 3.1, is-switch-des 3.5, is-cond-expr 3.3

Interpretation

(4) eval-expr(t,dn,vl) =

```
cases: t
is-funct-ref:  cases: access(t,dn,vl)
                      (t¹,vl¹,Ω): fn-access(t¹,dn,vl¹)
                      (t¹,vl¹,lab¹): (Ω,vl¹,lab¹)
¬is-funct-ref: cases: access(t,dn,vl)
                      (t²,vl²,Ω): (apply(t²),vl²,Ω)
                      (t²,vl²,lab²): (Ω,vl²,lab²)
```

refs: is-funct-ref 3.2, fn-access 3.2
note: The case distinction is made because access will not invoke the final function reference (cf. note on access).
type: is-expr X is-dn X is-vl → is-opt-op X is-vl X is-abn

(5) access(t,dn,vl) =

```
let: t¹ = reduce-cond-switch(t,dn)
cases: (t¹,ready-set(t¹,dn))
((is-funct-ref ∨ is-proc-st),{I}):
     (t¹,vl,Ω)
(¬(is-funct-ref ∨ is-proc-st),{ }):
     (t¹,vl,Ω)
T:   for some sel¹ ∈ ready-set(t¹,dn)
     cases: one-access(sel¹(t¹),dn,vl)
     (t²,vl²,Ω): access(μ(t¹;<sel¹:t²>),dn,vl²)
     (t²,vl²,lab²): (Ω,vl²,lab²)
```

refs: is-funct-ref 3.2, is-proc-st 4.7
note: Since this function handles procedure statements the final access is handled as a special case in order to avoid the check for no returned value.
type: is-text X is-dn X is-vl → is-opt-text X is-vl X is-abn

```
(6)   reduce-cond-switch(t,dn) =

          cases: t
          is-op ∨ is-object ∨ is-simple-var: t
          is-funct-ref:
               cases: s-act-par-list(t)
               Ω: t
               ¬is-Ω: μ(t;{<elem(i)•s-act-par-list:
                              reduce-cond-switch(elem(i)•s-act-par-list(t),dn)> |
                              1≤i≤length•s-act-par-list(t) & is-value-parm(i,s(s-id(t),dn))})
          is-switch-des:
               let: sub-list¹ = reduce-cond-switch(s-subscr-list(t),dn)
               cases: sub-list¹
               ¬is-op-expr-list: μ(t;<s-subscr-list:sub-list¹>)
               is-op-expr-list:
                    let: v¹ = convert(INTG,apply•elem(1,sub-list¹))
                         list¹ = s-switch-list•s(s-id(t),dn)
                    cases: v¹
                    1≤v¹≤length(list¹):
                         reduce-cond-switch(elem(v¹,list¹),dn)
                    T: error
          is-cond-expr:
               let: dec² = reduce-cond-switch(s-decision(t),dn)
               cases: dec²
               ¬is-op-expr: μ(t;<s-decision:dec²>)
               is-op-expr:
                    let: op² = apply(dec²)
                    cases: s-value(op²)
                    TRUE: μ(t;<s-decision:op²>,
                              <s-then-expr:reduce-cond-switch(s-then-expr(t),dn)>)
                    FALSE: μ(t;<s-decision:op²>,
                              <s-else-expr:reduce-cond-switch(s-else-expr(t),dn)>)
          is-ob: μ₀({<sel:reduce-cond-switch(sel(t),dn)> | is-selector(sel) & ¬is-Ω•sel(t)})

     refs: is-simple-var 3.1, is-funct-ref 3.2, is-switch-des 3.5, convert 4.2,
           is-cond-expr 3.3
     type: is-text X is-dn → is-text
```

```
(7)   ready-set(t,dn) =

          cases: t
          is-object ∨ is-op: { }
          is-simple-var ∨ is-array-name: {I}
          is-subscr-var:
               let: set¹ = ready-set(s-subscr-list(t),dn)
               cases: set¹
               { }: {I}
               ¬is-{ }: {sel•s-subscr-list | sel ∈ set¹}
          is-funct-ref:
               cases:s-act-par-list(t)
               Ω: {I}
               ¬is-Ω: let: set² = {sel•elem(i) | sel∈ready-set(elem(i)•s-act-par-list(t),dn) &
                                                  1≤i≤length•s-act-par-list(t) &
                                                  is-value-parm(i,s(s-id(t),dn))}
                      cases: set²
                      { }: {I}
                      ¬is-{ }: {sel•s-act-par-list | sel ∈ set²}
          is-cond-expr:
               cases: s-decision(t)
               ¬is-op: {sel•s-decision | sel ∈ ready-set(s-decision(t),dn)}
               is-op: cases: s-value(s-decision(t))
                      TRUE: {sel•s-then-expr | sel ∈ ready-set(s-then-expr(t),dn)}
                      FALSE: {sel•s-else-expr | sel ∈ ready-set(s-else-expr(t),dn)}
          is-ob: {sel•sel-1 | ¬is-Ω•sel-1(t) & is-selector(sel-1) &
                              sel ∈ ready-set(sel-1(t),dn)}

     refs: is-simple-var 3.1, is-array-name 3.2, is-subscr-var 3.1, is-funct-ref 3.2,
           is-cond-expr 3.3
     type: is-text X is-dn → is-sel-set


(8)   one-access(t,dn,vl) =

          cases: t
          is-funct-ref: fn-access(t,dn,vl)
          is-simple-var: (simp-var-access(t,dn,vl),vl,Ω)
          is-subscr-var: (subscr-var-access(t,dn,vl),vl,Ω)
          is-array-name: (array-access(t,dn,vl),vl,Ω)

     refs: is-funct-ref 3.2, fn-access 3.2, is-simple-var 3.1, simp-var-access 3.1,
           is-subscr-var 3.1, subscr-var-access 3.1, is-array-name 3.2, array-access 3.1
     type: is-text X is-dn X is-vl → is-opt-op X is-vl X is-abn
```

(9) apply(e) =

```
        cases: e
        is-op: e
        is-arithm-expr: apply-arithm-opr(e)
        is-bool-expr: apply-bool-opr(e)
        is-des-expr: apply-des-opr(e)

   refs: is-arithm-expr 3.3, apply-arithm-opr 3.3, is-bool-expr 3.4, apply-bool-opr 3.4,
         is-des-expr 3.5, apply-des-opr 3.5
   type: (is-expr ∨ is-array-op) → is-op
```

(10) is-value-parm(i,den) =

```
        elem(i)•s-form-par-list(den) ∈ s-value-pt(den)

   type: is-intg-val X is-proc-den → is-bool-val
```

3.1 VARIABLES

Translation

All variable identifiers must be declared in declarations or specifications:

(1) is-path(path) & is-real-simple-var•path(prog) ⊃ is-REAL•desc(s-id•path,prog)

(2) is-path(path) & is-intg-simple-var•path(prog) ⊃ is-INTG•desc(s-id•path,prog)

(3) is-path(path) & is-bool-simple-var•path(prog) ⊃ is-BOOL•desc(s-id•path,prog)

(4) is-path(path) & is-real-subscr-var•path(prog) ⊃ (is-REAL-ARRAY•s-type•desc(s-id•path,prog) ∨
is-REAL-ARRAY•desc(s-id•path,prog))

(5) is-path(path) & is-intg-subscr-var•path(prog) ⊃ (is-INTG-ARRAY•s-type•desc(s-id•path,prog) ∨
is-INTG-ARRAY•desc(s-id•path,prog))

(6) is-path(path) & is-bool-subscr-var•path(prog) ⊃ (is-BOOL-ARRAY•s-type•desc(s-id•path,prog) ∨
is-BOOL-ARRAY•desc(s-id•path,prog))

Abstract Syntax

(7) is-real-simple-var = (<s-id:is-id>,
<s-type:is-REAL>)

refs: is-id 2.4

(8) is-real-subscr-var = (<s-id:is-id>,
<s-subscr-list:(is-arithm-expr-list & ¬is-<>)>,
<s-type:is-REAL>)

refs: is-id 2.4, is-arithm-expr 3.3

(9) is-intg-simple-var = (<s-id:is-id>,
<s-type:is-INTG>)

refs:is-id 2.4

(10) is-intg-subscr-var = (<s-id:is-id>,
<s-subscr-list:(is-arithm-expr-list & ¬is-<>)>,
<s-type:is-INTG>)

refs: is-id 2.4, is-arithm-expr 3.3

(11) is-bool-simple-var = (<s-id:is-id>,
<s-type:is-BOOL>)

refs: is-id 2.4

(12) is-bool-subscr-var = (<s-id:is-id>,
<s-subscr-list:(is-arithm-expr-list & ¬is-<>)>,
<s-type:is-BOOL>)

refs: is-id 2.4, is-arithm-expr 3.3

(13) is-real-var = is-real-simple-var ∨ is-real-subscr-var

(14) is-intg-var = is-intg-simple-var ∨ is-intg-subscr-var

(15) is-arithm-var = is-real-var ∨ is-intg-var

(16) is-bool-var = is-bool-simple-var ∨ is-bool-subscr-var

Auxiliary Predicates

(17) is-simple-var = is-real-simple-var ∨ is-intg-simple-var ∨ is-bool-simple-var ∨ is-label-var

refs: is-label-var 3.5

(18) is-subscr-var = is-real-subscr-var ∨ is-intg-subscr-var ∨ is-bool-subscr-var

(19) is-var = is-simple-var ∨ is-subscr-var

Interpretation

(20) simp-var-access(t,dn,vl) =

cases: s-type(t)
LABEL: mk-op(LABEL,s-value•s(s-id(t),dn))
is-type: mk-op(s-type(t),s(s-id(t),vl))

refs: is-type 5.1
type: is-simple-var X is-dn X is-vl → (is-type-op ∨ is-label-op)

(21) subscr-var-access(t,dn,vl) =

let: $subl^1$ = μ_0({<elem(i):apply•elem(i)•s-subscr-list(t)> | 1≤i≤length•s-subscr-list(t)})
$subl^2$ = convert-subs(s-bounds•s(s-id(t),dn),$subl^1$)
mk-op(s-type(t),s($subl^2$,s(s-id(t),vl)))

refs: apply 3
type: is-subscr-var X is-dn X is-vl → is-type-op

(22) convert-subs(eb-list,sub-list) =

```
        cases: length(eb-list) = length(sub-list)
        TRUE: {<elem(i):convert-one-sub(elem(i,eb-list),elem(i,sub-list))> |
                                                          1≤i≤length(eb-list)}
        FALSE: error
```

error: If lengths of bound list and subscript list are unequal.
type: is-eb-list X is-arithm-op-list → is-intg-val-list

(23) convert-one-sub(eb,op) =

```
        let: v¹ = convert(INTG,op)
        cases: s-lbd(eb)≤v¹≤s-ubd(eb)
        TRUE: v¹
        FALSE: error
```

refs: convert 4.2
error: If the subscript value is outside the bounds.
type: is-eb X is-arithm-op → is-intg-val

(24) array-access(arr-name,dn,vl) =

```
        let: id = s-id(arr-name)
             ebl = s-bounds(s(id,dn))
             subl-set¹ = {<j(1),j(2),...,j(n)> | n = length(ebl) &
                  (1≤i≤n ⊃ is-intg-val•j(i) & s-lbd•elem(i)(ebl)≤j(i)≤s-ubd•elem(i)(ebl))}
        cases: (∀subl)(subl ∈ subl-set¹ ⊃ (∃val)(<subl,val> ∈ s(s-id(t),vl)))
        TRUE: μ0(<s-bounds:s-bounds•s(s-id(t),dn)>,
                 <s-value:s(s-id(t),vl)>,
                 <s-type:s-type(t)>)
        FALSE: error
```

refs: is-intg-val 2.5
error: If any element of the array is not initialised.
type: is-array-name X is-dn X is-vl → is-array-op

Translation

Function designators are characterised as follows:

(1) is-path(path) & (is-real-funct-ref(path(prog)) ⊃
is-REAL-PROC(s-type•desc(s-id•path,prog)) ∨ is-REAL-PROC(desc(s-id•path,prog))

(2) is-path(path) & (is-intg-funct-ref(path(prog)) ⊃
is-INTG-PROC(s-type•desc(s-id•path,prog)) ∨ is-INTG-PROC(desc(s-id•path,prog))

(3) is-path(path) & (is-bool-funct-ref(path(prog)) ⊃
is-BOOL-PROC(s-type•desc(s-id•path,prog)) ∨ is-BOOL-PROC(desc(s-id•path,prog))

(4) is-path(path) & is-non-type-proc-name•path(prog) ⊃
is-PROC•s-type•desc(s-id•path,prog) ∨ is-PROC•desc(s-id•path,prog)

(5) Unsubscripted array variables are only permitted in actual parameter lists:

is-path(path) & is-array-name•path(prog) ⊃
(∃path-1,i)(path = elem(i)•s-act-par-list•path-1) &
is-type-array•desc(s-id•path,prog) ∨ is-type-array•s-type•desc(s-id•path,prog)

refs: is-type-array 5.2

Abstract syntax

(6) is-non-type-proc-name = (<s-id:is-id>,
<s-type:is-PROC>)

refs: is-id 2.4

(7) is-array-name = (<s-id:is-id>,
<s-type:is-type-array>)

refs: is-id 2.4, is-type-array 5.2

(8) is-act-par = is-expr ∨ is-array-name ∨ is-non-type-proc-name ∨ is-string

refs: is-expr 3, is-string 2.6
note: Procedure identifiers referring to type procedures satisfy is-expr.

(9) is-real-funct-ref = (<s-id:is-id>,
<s-act-par-list:is-act-par-list ∨ is-Ω>,
<s-type:is-REAL-PROC>)

refs: is-id 2.4

(10) is-intg-funct-ref = (<s-id:is-id>,
<s-act-par-list:is-act-par-list ∨ is-Ω>,
<s-type:is-INTG-PROC>)

refs: is-id 2.4

(11) is-bool-funct-ref = (<s-id:is-id>,
<s-act-par-list:is-act-par-list ∨ is-Ω>,
<s-type:is-BOOL-PROC>)

refs: is-id 2.4

Auxiliary predicates

(12) is-funct-ref = is-real-funct-ref ∨ is-intg-funct-ref ∨ is-bool-funct-ref

(13) is-fn-ret = (<s-type:is-type>,
<s-value:is-simple-val ∨ is-Ω>)

refs: is-type 5.1

Interpretation

(14) fn-access(t,dn,vl) =

cases: proc-access(t,dn,vl)
(op¹,vl¹,Ω): cases: op¹
¬is-Ω: (op¹,vl¹,Ω)
Ω: error
(op¹,vl¹,lab¹): (Ω,vl¹,lab¹)

error: If a procedure (invoked by a function reference) terminates normally but does not return a value.
type: is-funct-ref X is-dn X is-vl → is-opt-op X is-vl X is-abn

```
(15)   proc-access(t,dn,vl) =

           let: act-par-list¹ = cases: s-act-par-list(t)
                                Ω: <>
                                ¬is-Ω: μ(s-act-par-list(t);
                                         {<elem(i):apply•elem(i)•s-act-par-list(t)> |
                                            1≤i≤length•s-act-par-list(t) &
                                            is-value-parm(i,s(s-id(t),dn))})
           activate-proc(s(s-id(t),dn),s-id(t),s-type(t),act-par-list¹,dn,vl)

       refs: apply 3, is-value-parm 3
       type: (is-funct-ref ∨ is-proc-st) X is-dn X is-vl → is-opt-op X is-vl X is-abn
```

```
(16)  activate-proc(proc,id,type,act-par-list,dn,vl) =

        let: form-par¹(i) = elem(i,s-form-par-list(proc))
             length¹ = length(s-form-par-list(proc))
        cases: length(act-par-list) = length¹
        FALSE: error
        TRUE:
             let: conv-act-par-list = eval-act-par-list(act-par-list,s-form-par-list(proc),
                                          s-spec-pt(proc),s-value-pt(proc))
                  act-par¹(i) = elem(i,conv-act-par-list)
                  name-repl = {<form-par¹(i),act-par¹(i)> |
                                    1≤i≤length¹ & ¬is-value-parm(i,proc)}
                  val-repl = mk-pairs({form-par¹(i) |
                                  1≤i≤length¹ & is-value-parm(i,proc)},proc,dn)
                  dn¹ = mod-set(dn,{<id²,val-den(act-par¹(i))> |
                                  1≤i≤length¹ & <id¹,id²> ∈ val-repl & id¹ = form-par¹(i)})
                  vl¹ = mod-set(vl,{<id²,s-value(act-par¹(i))> |
                                       1≤i≤length¹ & <id¹,id²> ∈ val-repl &
                                       ¬is-label-op(act-par¹(i)) & id¹ = form-par¹(i)})
             cases: type
             is-PROC: cases: non-type-proc(s-body(proc),val-repl ∪ name-repl,dn¹,vl¹)
                       (vl²,Ω): (Ω,epilogue(seconds(val-repl),vl²),Ω)
                       (vl²,lab²): (Ω,epilogue(seconds(val-repl),vl²),lab²)
             is-type-proc: cases: type-proc(proc,id,type,val-repl ∪ name-repl,dn¹,vl¹)
                            (op²,vl²,Ω): (op²,epilogue(seconds(val-repl),vl²),Ω)
                            (op²,vl²,lab²): (Ω,epilogue(seconds(val-repl),vl²),lab²)

     refs: is-value-parm 3, mk-pairs 4.1, non-type-proc 4.7, epilogue 4.1, is-type-proc 5.4
     error: If lengths of actual and formal parameter lists are unequal.
     note: This function called in both procedure and functional cases of procedure
           invocation.
     type: is-proc-den X is-id X is-type-proc X is-act-par-list X is-dn X is-vl →
                                                          is-opt-op X is-vl X is-abn


(17)  eval-act-par-list(act-par-list,form-par-list,spec-pt,val-pt) =

        let: spec¹(i) = (ι spec)(spec ∈ spec-pt & s-id(spec) = elem(i,form-par-list))
        μ₀({<elem(i):eval-act-par(elem(i,act-par-list),s-specifier(spec¹(i)),
                          (elem(i,form-par-list)∈val-pt))> | 1≤i≤length(form-par-list)})

     type: is-act-par-list X is-id-list X is-spec-set X is-id-set → is-act-par-list
```

```
(18)   eval-act-par(act-par,spec,flag) =

          cases: flag
          TRUE: cases: spec
                is-arithm: cases: act-par
                           is-arithm-op: let: val = convert(spec,act-par)
                                         mk-op(spec,val)
                           ¬is-arithm-op: error
                is-BOOL: cases: act-par
                         is-bool-op: act-par
                         ¬is-bool-op: error
                is-arithm-array:
                    cases: act-par
                    is-arithm-array-op: let: val = convert-array(spec,act-par)
                                        μ(act-par;<s-type:spec>,
                                                  <s-value:val>)
                    ¬is-arithm-array-op: error
                is-BOOL-ARRAY:
                    cases: act-par
                    is-bool-array-op: act-par
                    ¬is-bool-array-op: error
                is-LABEL: cases: act-par
                          is-label-op: act-par
                          ¬is-label-op: error
          FALSE: cases: match(spec,act-par)
                 TRUE: act-par
                 FALSE: error

      refs: is-arithm 5.1, convert 4.2, is-arithm-array 5.2
      error: If act-par and spec do not conform.
      note: flag is TRUE if and only if act-par is passed by value.
            See also 5.4(8) for constraints on by value specifications.
      type: is-act-par X is-specifier X is-bool-val → is-act-par


(19)   convert-array(spec,act-par) =

          {<int-list,val> | <int-list,v> ∈ s-value(act-par) &
                            val = convert-array-el(spec,v) }

      type: is-arithm-array X is-arithm-array-op → is-arithm-array-val
```

```
(20)   convert-array-el(arr-type,val) =

           cases: (arr-type,val)
           (REAL-ARRAY,is-intg-val): val
           (INTG-ARRAY,is-real-val): entier(val+0.5)
           T: val

       type: is-arithm-array X is-arithm-val → is-arithm-val


(21)   match(spec,act-par) =

           is-STRING(spec) & is-string(act-par) ∨
           is-INTG(spec) & is-intg-expr(act-par) ∨
           is-REAL(spec) & is-real-expr(act-par) ∨
           is-BOOL(spec) & is-bool-expr(act-apr) ∨
           is-type-array(spec) & spec = s-type(act-par) ∨
           is-LABEL(spec) & is-des-expr(act-par) ∨
           is-SWITCH(spec) & s-type(act-par) = spec ∨
           is-PROC(spec) & spec = s-type(act-par) ∨
           is-type-proc(spec) & spec = s-type(act-par)

       refs: is-string 2.6, is-intg-expr 3.3, is-real-expr 3.3, is-bool-expr 3.4,
             is-type-array 5.2, is-des-expr 3.5, is-type-proc 5.4
       type: is-specifier X is-act-par → is-bool-val


(22)   val-den(act-par) =

           cases: act-par
           is-type-op: μ0(<s-type:s-type(act-par)>)
           is-array-op: μ0(<s-type:s-type(act-par)>,
                           <s-bounds:s-bounds(act-par)>)
           is-label-op: act-par

       type: is-op → is-den
```

```
(23)  type-proc(den,id,type,pr-set,dn,vl) =

          let: {<id,id¹>} = mk-pairs({id},den,dn)
               type¹ = cases: type
                       REAL-PROC: REAL
                       INTG-PROC: INTG
                       BOOL-PROC: BOOL
               t¹ = insert-ret(id,id¹,type¹,s-body(den))
               dn¹ = mod-set(dn,{<id¹,μ0(<s-type:type¹>)>})
          cases: non-type-proc(t¹,pr-set,dn¹,vl)
          (vl¹,Ω): cases: (∃v)(<id¹,v> ∈ vl¹)
                   TRUE: (μ0(<s-type:type¹>,<s-value:s(id¹,vl¹)>),epilogue({id¹},vl¹),Ω)
                   FALSE: (μ0(<s-type:type¹>,<s-value:Ω>),epilogue({id¹},vl¹)Ω)
          (vl¹,lab¹): (Ω,epilogue({id¹},vl¹),lab¹)

     refs: mk-pairs 4.1, non-type-proc 4.7, epilogue 4.1
     note: func-id is made into a var with ret-id as its s-id component.
           Although no value is being returned from a type procedure, error should not be
           given unless invoked as a function reference (see fn-access).
     type: is-proc-den X is-id X is-type-proc X is-pr-set X is-dn X is-vl →
                                                  is-opt-fn-ret X is-vl X is-abn


(24)  insert-ret(func-id,ret-id,type,t) =

          cases: t
          is-code: t
          is-assign-st: μ(t;{<elem(i)•s-lp:μ0(<s-id:ret-id>,<s-type:type>)> |
                                  1≤i≤length(s-lp(t)) & is-activated-fn(elem(i,s-lp(t))) &
                                  s-id(elem(i,s-lp(t)))= func-id})
          is-set: {insert-ret(func-id,ret-id,type,el) | el ∈ t}
          is-object: t
          is-ob: μ0({<sel:insert-ret(func-id,ret-id,type,sel(t))> | is-selector(sel) &
                                                          ¬is-Ω(sel(t))})

     refs: is-code 5.4, is-assign-st 4.2, is-activated-fn 4.2
     type: is-id X is-id X is-type X is-text → is-text
```

Translation

(1) The precedence of operators and use of brackets is reflected in the abstract object representing the expression.

Abstract Syntax

(2) is-arithm-prefix-opr = is-PLUS ∨ is-MINUS

(3) is-real-prefix-expr = (<s-op:is-real-expr>,
<s-opr:is-arithm-prefix-opr>)

(4) is-intg-prefix-expr = (<s-op:is-intg-expr>,
<s-opr:is-arithm-prefix-opr>)

(5) is-real-infix-opr = is-PLUS ∨ is-MINUS ∨ is-MULT ∨ is-DIV ∨ is-POWER

(6) is-real-infix-expr-1 = (<s-op-1:is-real-expr>,
<s-op-2:is-real-expr>,
<s-opr:is-real-infix-opr>)

(7) is-real-infix-expr-2 = (<s-op-1:is-real-expr>,
<s-op-2:is-intg-expr>,
<s-opr:is-real-infix-opr>)

(8) is-real-infix-expr-3 = (<s-op-1:is-intg-expr>,
<s-op-2:is-real-expr>,
<s-opr:is-real-infix-opr>)

(9) is-real-infix-expr-4 = (<s-op-1:is-intg-expr>,
<s-op-2:(is-intg-expr & ¬is-non-neg-intg-const)>,
<s-opr:is-POWER>)

refs: is-non-neg-intg-const 2.5

(10) is-real-infix-expr-5 = (<s-op-1:is-intg-expr>,
<s-op-2:is-intg-expr>,
<s-opr:is-DIV>)

(11) is-real-infix-expr = is-real-infix-expr-1 ∨ is-real-infix-expr-2 ∨ is-real-infix-expr-3 ∨
is-real-infix-expr-4 ∨ is-real-infix-expr-5

(12) is-intg-infix-opr = is-PLUS ∨ is-MINUS ∨ is-MULT ∨ is-INTGDIV ∨ is-POWER

(13) is-intg-infix-expr-1 = (<s-op-1:is-intg-expr>,
<s-op-2:is-intg-expr>,
<s-opr:(is-intg-infix-opr & ¬is-POWER)>)

(14) is-intg-infix-expr-2 = (<s-op-1:is-intg-expr>,
<s-op-2:is-non-neg-intg-const>,
<s-opr:is-POWER>)

refs: is-non-neg-intg-const 2.5

(15) is-intg-infix-expr = is-intg-infix-expr-1 ∨ is-intg-infix-expr-2

(16) is-real-cond-expr-1 = (<s-decision:is-bool-expr>,
<s-then-expr:is-real-expr>,
<s-else-expr:is-real-expr>)

refs: is-bool-expr 3.4

(17) is-real-cond-expr-2 = (<s-decision:is-bool-expr>,
<s-then-expr:is-real-expr>,
<s-else-expr:is-intg-expr>)

refs: is-bool-expr 3.4

(18) is-real-cond-expr-3 = (<s-decision:is-bool-expr>,
<s-then-expr:is-intg-expr>,
<s-else-expr:is-real-expr>)

refs: is-bool-expr 3.4

(19) is-real-cond-expr = is-real-cond-expr-1 ∨ is-real-cond-expr-2 ∨ is-real-cond-expr-3

(20) is-intg-cond-expr = (<s-decision:is-bool-expr>,
<s-then-expr:is-intg-expr>,
<s-else-expr:is-intg-expr>)

refs: is-bool-expr 3.4

(21) is-real-expr = is-real-op ∨ is-real-var ∨ is-real-funct-ref ∨ is-real-prefix-expr ∨
is-real-infix-expr ∨ is-real-cond-expr

refs: is-real-var 3.1, is-real-funct-ref 3.2

(22) is-intg-expr = is-intg-op ∨ is-intg-var ∨ is-intg-funct-ref ∨ is-intg-prefix-expr ∨
is-intg-infix-expr ∨ is-intg-cond-expr

refs: is-intg-var 3.1, is-intg-funct-ref 3.2

(23) is-arithm-expr = is-real-expr ∨ is-intg-expr

Auxiliary predicates

(24) is-arithm-prefix-expr = is-real-prefix-expr ∨ is-intg-prefix-expr

(25) is-arithm-infix-expr = is-real-infix-expr ∨ is-intg-infix-expr

(26) is-arithm-cond-expr = is-real-cond-expr ∨ is-intg-cond-expr

(27) is-cond-expr = is-arithm-cond-expr ∨ is-bool-cond-expr ∨ is-des-cond-expr

refs: is-bool-cond-expr 3.4, is-des-cond-expr 3.5

(28) is-arithm-infix-opr = is-real-infix-opr ∨ is-intg-infix-opr

Interpretation

```
(29)   apply-arithm-opr(e) =

            cases: e
            is-arithm-prefix-expr: arithm-prefix-opr(apply•s-op(e),s-opr(e))
            is-arithm-infix-expr: arithm-infix-opr(apply•s-op-1(e),apply•s-op-2(e),s-opr(e))
            is-arithm-cond-expr: cases: s-value•s-decision(e)
                                 TRUE: cases: is-intg-expr•s-then-expr(e) & is-real-expr(e)
                                       TRUE: mk-op(REAL,s-value•apply(s-then-expr(e)))
                                       FALSE: apply•s-then-expr(e)
                                 FALSE: cases: is-intg-expr•s-else-expr(e) & is-real-expr(e)
                                        TRUE: mk-op(REAL,s-value•apply(s-else-expr(e)))
                                        FALSE: apply•s-else-expr(e)

       refs: apply 3
       type: is-arithm-expr → is-arithm-op


(30)   arithm-prefix-opr(op,opr) =

            cases: s-type(op)
            INTG: cases: opr
                  PLUS: op
                  MINUS: mk-op(INTG, - s-value(op))
            REAL: mk-op(REAL,real-prefix-value(s-value(op),opr))

       type: is-arithm-op X is-arithm-prefix-opr → is-arithm-op


(31)   real-prefix-value(v,opr) =

       This function is implementation-defined.

       type: is-real-val X is-arithm-prefix-opr → is-real-val
```

```
(32)   arithm-infix-opr(op-1,op-2,opr) =

           let: type¹ = s-type(op-1)
                type² = s-type(op-2)
                v¹ = s-value(op-1)
                v² = s-value(op-2)
           cases: (type¹,type²,opr)
           (is-arithm,INTG,POWER): intg-power-opr(op-1,op-2))
           (is-arithm,REAL,POWER): mk-op(REAL,real-power-value(v¹,v²))
           (INTG,INTG,is-intg-infix-opr): mk-op(INTG,intg-arithm-infix-value(v¹,v²,opr))
           (is-arithm,is-arithm,is-real-infix-opr):
                     mk-op(REAL,real-arithm-infix-value(v¹,v²,opr))

       refs: is-arithm 5.1
       type: is-arithm-op X is-arithm-op X is-arithm-infix-opr → is-arithm-op


(33)   intg-power-opr(op-1,op-2) =

           let: v¹ = intg-power-opr-val(op-1,s-value(op-2))
           cases: is-intg-op(op-1) & is-non-neg-intg-const(op-2)
           TRUE: mk-op(INTG,v¹)
           FALSE: mk-op(REAL,v¹)

       refs: is-non-neg-intg-const 2.5
       note: intg-op to power non-neg-intg-const yields an intg result.
       type: is-arithm-op X is-intg-op → is-arithm-op


(34)   intg-power-opr-val(op,i) =

           cases: i
           i>0: s-value•self-mult(op,i)
           i=0: cases: s-value(op) ≠ 0
                TRUE: mk-op(s-type(op),1)
                FALSE: error
           i<0: cases: s-value(op) ≠ 0
                TRUE: arithm-infix-opr(mk-op(REAL,1),self-mult(op,-i),DIV)
                FALSE: error

       error: If s-value(op) = 0 and i≤0.
       type: is-arithm-op X is-intg-val → is-arithm-val
```

(35) self-mult(op,i) =

```
cases: i
i>1: arithm-infix-opr(self-mult(op,i-1),op,MULT)
i=1: op
```

type: is-arithm-op X is-non-neg-intg-val → is-arithm-op

(36) real-power-value(v-1,v-2) =

```
cases: v-1
v-1>0: real-arithm-infix-value(v-1,v-2,POWER)
v-1=0: cases: v-2
       v-2>0: 0
       v-2≤0: error
v-1<0: error
```

error: If a zero value is raised to a negative or zero power, or a negative value is raised to any power.
type: is-arithm-val X is-real-val → is-real-val

(37) real-arithm-infix-value(v-1,v-2,opr) =

This function is implementation defined.

type: is-real-val X is-real-val X is-real-infix-opr → is-real-val

(38) intg-arithm-infix-value(v-1,v-2,opr) =

```
cases: opr
PLUS: v-1 + v-2
MINUS: v-1 - v-2
MULT: v-1 * v-2
INTGDIV: sign(v-1/v-2) * entier•abs(v-1/v-2)
```

type: is-intg-val X is-intg-val X is-intg-infix-opr → is-intg-val

3.4 BOOLEAN EXPRESSIONS

Translation

(1) The precedence of operators and use of brackets is reflected in the abstract object representing the expression.

Abstract Syntax

(2) is-bool-prefix-expr = (<s-opr:is-NOT>,<s-op:is-bool-expr>)

(3) is-bool-infix-opr = is-AND ∨ is-OR ∨ is-IMPL ∨ is-EQUIV

(4) is-bool-infix-expr = (<s-opr:is-bool-infix-opr>,
<s-op-1:is-bool-expr>,
<s-op-2:is-bool-expr>)

(5) is-bool-cond-expr = (<s-decision:is-bool-expr>,
<s-then-expr:is-bool-expr>,
<s-else-expr:is-bool-expr>)

(6) is-arithm-relat-opr = is-GT ∨ is-GE ∨ is-EQ ∨ is-LE ∨ is-LT ∨ is-NE

(7) is-arithm-relat-expr = (<s-opr:is-arithm-relat-opr>,
<s-op-1:is-arithm-expr>,
<s-op-2:is-arithm-expr>)

refs: is-arithm-expr 3.3

(8) is-bool-expr = is-bool-op ∨ is-bool-var ∨ is-bool-funct-ref ∨ is-bool-prefix-expr ∨
is-bool-infix-expr ∨ is-bool-cond-expr ∨ is-arithm-relat-expr

refs: is-bool-var 3.1, is-bool-funct-ref 3.2

Interpretation

```
(9)     apply-bool-opr(e) =

            cases: e
            is-bool-prefix-expr: mk-op(BOOL, ¬ s-value•apply•s-op(e))
            is-bool-infix-expr: bool-infix-opr(apply•s-op-1(e),apply•s-op-2(e),s-opr(e))
            is-arithm-relat-expr: arithm-relat-opr(apply•s-op-1(e),apply•s-op-2(e),s-opr(e))
            is-bool-cond-expr: cases: s-value•s-decision(e)
                               TRUE: apply•s-then-expr(e)
                               FALSE: apply•s-else-expr(e)

        refs: apply 3
        type: is-bool-expr → is-bool-op


(10)    bool-infix-opr(op-1,op-2,opr) =

            mk-op(BOOL,bool-infix-value(s-value(op-1),s-value(op-2),opr))

        type: is-bool-op X is-bool-op X is-bool-infix-opr → is-bool-op


(11)    bool-infix-value(v-1,v-2,opr) =

            cases: opr
            AND  : v-1 & v-2
            OR   : v-1 ∨ v-2
            IMPL : v-1 ⊃ v-2
            EQUIV: v-1 ≡ v-2

        type: is-bool-val X is-bool-val X is-bool-infix-opr → is-bool-val


(12)    arithm-relat-opr(op-1,op-2,opr) =

            mk-op(BOOL,arithm-relat-value(s-value(op-1),s-value(op-2),opr))

        type: is-arithm-op X is-arithm-op X is-arithm-relat-opr → is-bool-op
```

(13) arithm-relat-value(v-1,v-2,opr) =

cases: opr
GT: v-1 > v-2
GE: v-1 ≥ v-2
EQ: v-1 = v-2
LE: v-1 ≤ v-2
LT: v-1 < v-2
NE: v-1 ≠ v-2

type: is-arithm-val X is-arithm-val X is-relat-opr → is-bool-val

3.5 DESIGNATIONAL EXPRESSIONS

Translation

(1) is-path(path) & is-switch-des•path(prog) ⊃ (is-switch-desc•desc(s-id•path,prog) ∨ is-SWITCH•desc(s-id•path,prog))

refs: is-switch-desc 5.3

(2) is-path(path) & is-label-const(path(prog)) ⊃
is-label-desc(desc(s-id•path,prog))

refs: is-label-desc 5

(3) is-path(path) & is-label-var(path(prog)) ⊃
is-LABEL(desc(s-id•path,prog))

note: This only applies to the translation of label parameters.

Abstract Syntax

(4) is-switch-des = (<s-id:is-id>,
<s-subscr-list:(<elem(1):is-arithm-expr>)>,
<s-type:is-SWITCH>)

refs: is-id 2.4, is-arithm-expr 3.3

```
(5)     is-des-cond-expr = (<s-decision:is-bool-expr>,
                            <s-then-expr:is-des-expr>,
                            <s-else-expr:is-des-expr>)

           refs: is-bool-expr 3.4


(6)     is-label-var = (<s-id:is-id>,
                        <s-type:is-LABEL>)

           refs: is-id 2.4


(7)     is-label-const = (<s-type:is-LABEL>,
                          <s-value:is-id>,
                          <s-const:is-CONST>)

           refs: is-id 2.4


(8)     is-des-expr = is-label-op v is-label-var v is-switch-des v is-des-cond-expr
```

Interpretation

```
(9)     apply-des-opr(e) =

              cases: s-value•s-decision(e)
              TRUE: apply•s-then-expr(e)
              FALSE: apply•s-else-expr(e)

           refs: apply 3
           type: is-des-cond-expr → is-label-op
```

Translation

(1) The <labels> at the head of a <basic statement> etc. are collected into a set.

(2) The Translator rejects any <program> in which errors of duplication would be hidden by (1) (e.g. lab:lab:x:=1;).

(3) local(id,t) =

(∃sel)(id∈s-label-pt•sel(t) & ¬(∃sel-1,sel-2)(sel=sel-2•sel-1 & is-block•sel-1(t)))

refs: is-block 4.1
note: It is implicit in the above definition that labels within procedure declarations are omitted because no selector will yield components of the declaration set.
type: is-id X is-text → is-bool-val

(4) make-st-sel(id,t) =

(i sel)(id∈s-label-pt•sel(t) & ¬(∃sel-1,sel-2)(sel=sel-2•sel-1 & is-block•sel-1(t)))

refs: is-block 4.1
note: Only used if local(id,t).
type: is-id X is-text → is-lab-sel

Abstract syntax

(5) is-unlab-st = is-comp-st ∨ is-block ∨ is-assign-st ∨ is-goto-st ∨ is-dummy-st ∨ is-cond-st ∨ is-for-st ∨ is-proc-st

refs: is-comp-st 4.1, is-block 4.1, is-assign-st 4.2, is-goto-st 4.3, is-dummy-st 4.4, is-cond-st 4.5, is-for-st 4.6, is-proc-st 4.7

(6) is-st = (<s-label-pt:is-id-set>, <s-st-pt:is-unlab-st>)

refs: is-id 2.4

Interpretation

```
(7)     int-st(st,dn,vl) =

            cases: int-unlab-st(s-st-pt(st),dn,vl)
            (vl¹,Ω): (vl¹,Ω)
            (vl¹,lab¹): cases: lab¹
                        local(lab¹,st): cue-int-st(make-st-sel(lab¹,st),st,dn,vl¹)
                        ¬local(lab¹,st): (vl¹,lab¹)

        type: is-st X is-dn X is-vl → is-vl X is-abn


(8)     int-unlab-st(t,dn,vl) =

            cases: t
            is-comp-st: int-st-list(t,1,dn,vl)
            is-block: int-block(t,dn,vl)
            is-assign-st: int-assign-st(t,dn,vl)
            is-goto-st: int-goto-st(t,dn,vl)
            is-dummy-st: (vl,Ω)
            is-cond-st: int-cond-st(t,dn,vl)
            is-for-st: int-for-st(t,dn,vl)
            is-proc-st: int-proc-st(t,dn,vl)

        refs: is-comp-st 4.1, int-st-list 4.1, is-block 4.1, int-block 4.1, is-assign-st 4.2,
              int-assign-st 4.2, is-goto-st 4.3, int-goto-st 4.3, is-dummy-st 4.4, is-cond-st 4.5,
              int-cond-st 4.5, is-for-st 4.6, int-for-st 4.6, is-proc-st 4.7, int-proc-st 4.7
        type: is-unlab-st X is-dn X is-vl → is-vl X is-abn


(9)     cue-int-st(targ-sel,st,dn,vl) =

            cases: targ-sel
            I: int-st(st,dn,vl)
            ¬is-I: cases: cue-int-unlab-st(rest-pt(targ-sel),s-st-pt(st),dn,vl)
                   (vl¹,Ω): (vl¹,Ω)
                   (vl¹,lab¹): cases: lab¹
                               local(lab¹,st): cue-int-st(make-st-sel(lab¹,st),st,dn,vl¹)
                               ¬local(lab¹,st): (vl¹,lab¹)

        type: is-lab-sel X is-st X is-dn X is-vl → is-vl X is-abn
```

(10) cue-int-unlab-st(targ-sel,t,dn,vl) =

```
cases: t
is-comp-st: let: i¹ = (i i)(main-pt(targ-sel) = elem(i))
            cue-int-st-list(rest-pt(targ-sel),t,i¹,dn,vl)
is-cond-st: cue-int-st(rest-pt(targ-sel),main-pt(targ-sel)(t),dn,vl)
is-for-st: error
```

refs: is-comp-st 4.1, cue-int-st-list 4.1, is-cond-st 4.5, is-for-st 4.6
error: goto into for not allowed (see A.R. 4.6.6).
type: is-lab-sel X is-unlab-st X is-dn X is-vl → is-vl X is-abn

4.1 COMPOUND STATEMENTS AND BLOCKS

Translation

(1) All <programs> are surrounded by an embracing block which introduces all of the standard functions (see AR 3.2.4). SIGN must be included since it is used in interpretation as though referenced in the text.

(2) All <procedure bodies> which are not <code> are converted to blocks.

(3) intr-ids(t) =

{id | (∃d)(d ∈ s-decl-pt(t) & id ∈ s-id-set(d))} ∪ local-labs(t)

type: is-block → is-id-set

(4) local-labs(t) =

{id | local(id,t)}

refs: local 4
type: is-block → is-id-set

```
(5)     desc-block(id,path-el•path,t) =

            cases: id
            (∃decl)(decl ∈ s-decl-pt•path-el•path(t) &  id ∈ s-id-set(decl)):
                 let: decl¹ = (ι decl)(decl∈s-decl-pt•path-el•path(t) & id ∈ s-id-set(decl))
                 s-desc(decl¹)
            id ∈ local-labs(path-el•path(t)): μ0(<s-type:LABEL>)
            T: desc-1(id,path,t)

        type: is-id X is-path X is-program → (is-specifier ∨ is-desc ∨ is-label-desc)
```

Abstract syntax

```
(6)     is-comp-st = is-st-list

        refs: is-st 4

(7)     is-block = (<s-decl-pt:is-decl-set>,
                    <s-st-list:is-st-list>)

        refs: is-decl 5, is-st 4

(8)     is-program = is-block
```

Interpretation

```
(9)     int-program(t) =

                cases: int-block(t,{ },{ })
                ({ },Ω): EMPTY

           note: Only possible case.
           type: is-program → EMPTY


(10)    int-block(t,dn,vl) =

                cases: eval-array-decls(s-decl-pt(t),dn,vl)
                (decl-set¹,vl¹,Ω):
                     let: pr-set¹ = mk-pairs(intr-ids(t),t,dn)
                          t¹ = change-block(μ(t;<s-decl-pt:decl-set¹>),pr-set¹)
```

```
                    labs¹ = μ₀(<s-id-set:{id | id∈local-labs(t¹)}>,
                              <s-desc:μ₀(<s-type:is-LABEL>)>)
                    dn¹ = augment-dn(s-decl-pt(t¹) ∪ {labs¹},dn)
                    vl² = mod-set(vl¹,{<id,{}>|decl∈s-decl-pt(t¹) &
                                              is-array-desc•s-desc(decl) & id ∈ s-id-set(decl)})
                    (vl³,abn³) = int-block-body(s-st-list(t¹),dn¹,vl²)
                    vl⁴ = epilogue(seconds(pr-set¹),vl³)
               (vl⁴,abn³)
          (decl-set¹,vl¹,lab¹): (vl¹,lab¹)

     refs: eval-array-decls 5.2, change-block 4.7, augment-dn 5, is-array-desc 5.2
     note: Array bounds are evaluated before the new dn is installed, no local refs are
           possible, nor are local gotos (see A.R. 5.2.4.2).
     type: is-block X is-dn X is-vl → is-vl X is-abn


(11) mk-pairs(id-set,t,dn) =

          let: used-set¹ = all-intrs(t) ∪ {id | pden ∈ seconds(dn) & is-proc-den(pden) &
                                                id ∈ all-intrs(pden)}
               avoid-set¹ = id-set ∪ used-set¹
          construct-pairs(id-set,used-set¹,avoid-set¹)

     type: is-id-set X (is-block ∨ is-proc-den) X is-dn → is-idpr-set


(12) construct-pairs(id-set,us-set,av-set) =

          cases: id-set
          {}: {}
          ¬is-{}: for some id ∈ id-set
                  let: id¹ = cases: id ∈ us-set
                             FALSE: id
                             TRUE: for some id² ∈ {id | is-id(id) & ¬(id ∈ av-set)}
                                   id²
                  {<id,id¹>} ∪ construct-pairs(id-set - {id},us-set,av-set ∪ {id¹})

     refs: is-id 2.4
     note: The arbitrary order does not affect the result.
           No name is chosen which either has been used or is bound in a piece of text
             which can become active.
     type: is-id-set X is-id-set X is-id-set → is-idpr-set
```

(13) all-intrs(t) =

```
{id | (∃path)(path≠I & (is-block(path(t)) & (id ∈ intr-ids(path(t)) ∨
        is-proc-desc(path(t)) & (∃i)(elem(i)•s-form-par-list(path(t))=id))) }
```

refs: is-proc-desc 5.4
note: This function collects all identifiers bound in nested blocks or procedure declarations. (Not those of the argument text.)
type: is-text → is-id-set

(14) change-block(t,pr-set) =

```
μ0(<s-decl-pt:{change-text(d,pr-set) | d∈s-decl-pt(t)}>,
   <s-st-list:change-text(s-st-list(t),pr-set)>)
```

refs: change-text 4.7
note: Split in this way in order to change the outer block with a non-deleted set. Always produces an object satisfying is-block because id-prs used.
type: is-block X is-id-pr-set → is-block

(15) int-block-body(t,dn,vl) =

```
int-st-list(t,1,dn,vl)
```

type: is-st-list X is-dn X is-vl → is-vl X is-abn

(16) int-st-list(t,i,dn,vl) =

```
cases: i
i > length(t): (vl,Ω)
i ≤ length(t):
     cases: int-st(elem(i,t),dn,vl)
     (vl¹,Ω): int-st-list(t,i+1,dn,vl¹)
     (vl¹,lab¹): cases: lab¹
                 local(lab¹,t): let: ext = make-st-sel(lab¹,t)
                                     i¹ = (i j)(elem(j) = main-pt(ext))
                                cue-int-st-list(rest-pt(ext),t,i¹,dn,vl¹)
                 ¬local(lab¹,t): (vl¹,lab¹)
```

refs: int-st 4, local 4, make-st-sel 4
type: is-st-list X is-intg-val X is-dn X is-vl → is-vl X is-abn

```
(17)   cue-int-st-list(targ-sel,t,i,dn,vl) =

           cases: cue-int-st(targ-sel,elem(i,t),dn,vl)
           (vl¹,Ω): int-st-list(t,i+1,dn,vl¹)
           (vl¹,lab¹): cases: lab¹
                       local(lab¹,t): let: ext = make-st-sel(lab¹,t)
                                           i¹ = (i j)(elem(j) = main-pt(ext))
                                      cue-int-st-list(rest-pt(ext),t,i¹,dn,vl¹)
                       ¬local(lab¹,t): (vl¹,lab¹)

       refs: cue-int-st 4, local 4, make-st-sel 4
       type: is-lab-sel X is-st-list X is-intg-val X is-dn X is-vl → is-vl X is-abn


(18)   epilogue(id-set,vl) =

           del-set(vl,id-set)

       type: is-id-set X is-vl → is-vl
```

Translation

A procedure identifier can only appear on the left of an assignment statement if it is a type procedure and the assignment statement is within the body of the procedure itself. In such cases the type of the procedure is noted in the abstract text.

(1) is-path(path) & is-real-activated-fn(path(prog)) ⊃
(∃ path-1,path-2)(is-proc-desc•path-1(prog) & path = path-2•path-1 &
path-1(prog) = desc(s-id•path,prog) &
is-REAL-PROC•s-type•desc(s-id•path,prog))

refs: is-proc-desc 5.4

(2) is-path(path) & is-intg-activated-fn(path(prog)) ⊃
(∃ path-1,path-2)(is-proc-desc•path-1(prog) & path = path-2•path-1 &
path-1(prog) = desc(s-id•path,prog) &
is-INTG-PROC•s-type•desc(s-id•path,prog))

refs: is-proc-desc 5.4

(3) is-path(path) & is-bool-activated-fn(path(prog)) ⊃
(∃ path-1,path-2)(is-proc-desc•path-1(prog) & path = path-2•path-1 &
path-1(prog) = desc(s-id•path,prog) &
is-BOOL-PROC•s-type•desc(s-id•path,prog))

refs: is-proc-desc 5.4

Abstract syntax

(4) is-real-activated-fn = (<s-id:is-id>,
<s-type:is-REAL-PROC>)

refs: is-id 2.4

(5) is-real-lp = is-real-var ∨ is-real-activated-fn

refs: is-real-var 3.1

(6) is-intg-activated-fn = (<s-id:is-id>,
<s-type:is-INTG-PROC>)

refs: is-id 2.4

(7) is-intg-lp = is-intg-var ∨ is-intg-activated-fn

refs: is-intg-var 3.1

(8) is-bool-activated-fn = (<s-id:is-id>,
<s-type:is-BOOL-PROC>)

refs: is-id 2.4

(9) is-bool-lp = is-bool-var ∨ is-bool-activated-fn

refs: is-bool-var 3.1

(10) is-real-assign-st = (<s-lp:(is-real-lp-list & ¬is-<>)>,
<s-rp:is-arithm-expr>)

refs: is-arithm-expr 3.3

(11) is-intg-assign-st = (<s-lp:(is-intg-lp-list & ¬is-<>)>,
<s-rp:is-arithm-expr>)

refs: is-arithm-expr 3.3

(12) is-bool-assign-st = (<s-lp:(is-bool-lp-list & ¬is-<>)>,
<s-rp:is-bool-expr>)

refs: is-bool-expr 3.4

(13) is-assign-st = is-intg-assign-st ∨ is-real-assign-st ∨ is-bool-assign-st

Auxiliary predicates

(14) is-activated-fn = is-real-activated-fn ∨ is-intg-activated-fn ∨ is-bool-activated-fn

(15) is-lp = is-real-lp ∨ is-intg-lp ∨ is-bool-lp

Interpretation

```
(16)  int-assign-st(t,dn,vl) =

          cases: eval-lp-list(s-lp(t),dn,vl)
          (lp-list¹,vl¹,Ω):
               cases: eval-expr(s-rp(t),dn,vl¹)
               (op²,vl²,Ω): (change-lp-vars(lp-list¹,op²,dn,vl²),Ω)
               (op²,vl²,lab²): (vl²,lab²)
          (lp-list¹,vl¹,lab¹): (vl¹,lab¹)

      ref: eval-expr 3
      note: All left parts now satisfy is-var (cf. insert-ret 3.2).
      type: is-assign-st X is-dn X is-vl → is-vl X is-abn


(17)  eval-lp-list(lp-list,dn,vl) =

          cases: lp-list
          <>: (<>,vl,Ω)
          ¬is-<>:
               cases: eval-lp(head(lp-list),dn,vl)
               (lp¹,vl¹,Ω):
                    cases: eval-lp-list(tail(lp-list),dn,vl¹)
                    (lp-list²,vl²,Ω): (<lp¹>"lp-list²,vl²,Ω)
                    (lp-list²,vl²,lab²): (Ω,vl²,lab²)
               (lp¹,vl¹,lab¹): (Ω,vl¹,lab¹)

      type: is-var-list X is-dn X is-vl → is-opt-var-list X is-vl X is-abn
```

```
(18)   eval-lp(lp,dn,vl) =

           cases: lp
           is-simple-var: (lp,vl,Ω)
           is-subscr-var:
                cases: eval-subs(s-subscr-list(lp),dn,vl)
                (op-list¹,vl¹,Ω):
                     let: e-sub-list = convert-subs(s-bounds•s(s-id(lp),dn),op-list¹)
                          var¹ = μ(lp;<s-subscr-list:e-sub-list>)
                     (var¹,vl¹,Ω)
                (op-list¹,vl¹,lab¹): (Ω,vl¹,lab¹)

       refs: is-simple-var 3.1, is-subscr-var 3.1, convert-subs 3.1
       type: is-var X is-dn X is-vl → is-opt-var X is-vl X is-abn


(19)   eval-subs(sub-list,dn,vl) =

           cases: sub-list
           <>: <>
           ¬is-<>:
                cases: eval-expr(head(sub-list),dn,vl)
                (e-hd,vl¹,Ω):
                     cases: eval-subs(tail(sub-list),dn,vl¹)
                     (e-tl,vl²,Ω): (<e-hd>"e-tl,vl²,Ω)
                     (e-tl,vl²,lab²): (Ω,vl²,lab²)
                (e-hd,vl¹,lab¹): (Ω,vl¹,lab¹)

       refs: eval-expr 3
       type: is-arithm-expr-list X is-dn X is-vl → is-opt-op-list X is-vl X is-abn


(20)   change-lp-vars(lp-list,op,dn,vl) =

           let: v¹ = cases: lp-list
                     is-real-lp-list: convert(REAL,op)
                     is-intg-lp-list: convert(INTG,op)
                     is-bool-lp-list: op
           assign-to-lp-list(lp-list,v¹,vl)

       type: is-var-list X is-op X is-dn X is-vl → is-vl
```

```
(21)   convert(type,op) =

           cases: (type,s-type(op))
           (REAL,INTG): s-value(op)
           (INTG,REAL): entier(s-value(op) + 0.5)
           T:           s-value(op)

       type: is-arithm X is-arithm-op → is-arithm-val


(22)   assign-to-lp-list(lp-list,val,vl) =

           cases: lp-list
           <>: vl
           ¬is-<>: let: vl¹ = assign(head(lp-list),val,vl)
                   assign-to-lp-list(tail(lp-list),val,vl¹)

       type: is-var-list X is-simple-val X is-vl → is-vl


(23)   assign(lp,v,vl) =

           cases: lp
           is-simple-var: mod-set(vl,{<s-id(lp),v>})
           is-subscr-var: mod-set(vl,{<s-id(lp),mod-set(s(s-id(lp),vl),
                                                        {<s-subscr-list(lp),v>})>})

       refs: is-simple-var 3.1, is-subscr-var 3.1
       type: is-var X is-simple-val X is-vl → is-vl
```

4.3 GOTO STATEMENTS

Abstract syntax

```
(1)    is-goto-st = (<s-des-expr:is-des-expr>)

       refs: is-des-expr 3.5
```

Interpretation

(2) int-goto-st(t,dn,vl) =

```
        cases: eval-expr(s-des-expr(t),dn,vl)
        (lab-den,vl¹,Ω): (vl¹,s-value(lab-den))
        (lab-den,vl¹,lab¹): (vl¹,lab¹)

    refs: eval-expr 3
    note: Switch designators whose value is undefined give error in
          eval-expr (cf. A.R. 4.3.5).
    type: is-goto-st X is-dn X is-vl → is-vl X is-abn
```

4.4 DUMMY STATEMENTS

Translation

(1) The elementary object DUMMY is inserted in place of the <dummy statement>.

Abstract syntax

(2) is-dummy-st = is-DUMMY

Interpretation

(3) see int-unlab-st :- is-dummy-st

refs: int-unlab-st 4

4.5 CONDITIONAL STATEMENTS

Translation

(1) A dummy-st is inserted as the s-else-st if none is present in the <conditional statement>.

Abstract syntax

(2) is-cond-st = (<s-decision:is-bool-expr>,
<s-then-st:is-st>,
<s-else-st:is-st>)

refs: is-bool-expr 3.4, is-st 4
note: The concrete syntax is such that: ¬is-cond-st(s-st-pt(s-then-st(t))).

Interpretation

```
(3)   int-cond-st(t,dn,vl) =

            cases:eval-expr(s-decision(t),dn,vl)
            (bool-op¹,vl¹,Ω): cases: s-value(bool-op¹)
                              TRUE: int-st(s-then-st(t),dn,vl¹)
                              FALSE: int-st(s-else-st(t),dn,vl¹)
            (bool-op¹,vl¹,lab¹): (vl¹,lab¹)

      refs: eval-expr 3, int-st 4
      type: is-cond-st X is-dn X is-vl → is-vl X is-abn
```

4.6 FOR STATEMENTS

Abstract syntax

```
(1)     is-while-elem = (<s-init-expr:is-arithm-expr>,
                         <s-while-expr:is-bool-expr>)

           refs: is-arithm-expr 3.3, is-bool-expr 3.4


(2)     is-step-until-elem = (<s-init-expr:is-arithm-expr>,
                              <s-step-expr:is-arithm-expr>,
                              <s-until-expr:is-arithm-expr>)

           refs: is-arithm-expr 3.3
```

```
(3)    is-for-elem = is-arithm-expr ∨ is-while-elem ∨ is-step-until-elem

          refs: is-arithm-expr 3.3


(4)    is-for-st = (<s-contr-var:is-arithm-var>,
                    <s-for-list:(is-for-elem-list & ¬is-<>)>,
                    <s-st:is-st>)

          refs: is-arithm-var 3.1, is-st 4
```

Interpretation

```
(5)    int-for-st(t,dn,vl) =

             cases: eval-lp(s-contr-var(t),dn,vl)
             (cvar,vl¹,Ω): iterate-for-list(cvar,s-for-list(t),s-st(t),dn,vl¹)
             (cvar,vl¹,lab¹): (vl¹,lab¹)

          refs: eval-lp 4.2
          note: goto from evaluation of a for list is considered as external to a for statement.
                The control variable is evaluated once only.
          type: is-for-st X is-dn X is-vl → is-vl X is-abn


(6)    iterate-for-list(cvar,for-list,t,dn,vl) =

             cases: for-list
             <>: cases: cvar
                 is-simple-var: (del-set(vl,s-id(cvar)),Ω)
                 is-subscr-var: (mod-set(vl,{<s-id(cvar),del-set(s(s-id(cvar),vl),
                                                                 s-subscr-list(cvar))>}),Ω)
             ¬is-<>: cases: iterate-for(cvar,head(for-list),t,dn,vl)
                     (vl¹,Ω): iterate-for-list(cvar,tail(for-list),t,dn,vl¹)
                     (vl¹,lab¹): (vl¹,lab¹)

        refs: is-simple-var 3.1, is-subscr-var 3.1
        type: is-arithm-var X is-for-elem-list X is-st X is-dn X is-vl → is-vl X is-abn
```

```
(7)   iterate-for(cvar,for-elem,t,dn,vl) =

          cases: for-elem
          is-arithm-expr:
                   cases: eval-expr(for-elem,dn,vl)
                   (op¹,vl¹,Ω): let: v = convert(s-type(cvar),op¹)
                                     vl² = assign(cvar,v,vl¹)
                                int-st(t,dn,vl²)
                   (op¹,vl¹,lab¹): (vl¹,lab¹)
          is-step-until-elem:
               cases: eval-expr(s-init-expr(for-elem),dn,vl)
               (op¹,vl¹,Ω): let: v = convert(s-type(cvar),op¹)
                                 vl² = assign(cvar,v,vl¹)
                            iterate-step-until-elem(cvar,s-step-expr(for-elem),
                                                    s-until-expr(for-elem),t,dn,vl²)
               (op¹,vl¹,lab¹): (vl¹,lab¹)
          is-while-elem: iterate-while(cvar,for-elem,t,dn,vl)

      refs: is-arithm-expr 3.3, eval-expr 3, convert 4.2, assign 4.2, int-st 4
      note: Init expr eval once only.
      type: is-arithm-var X is-for-elem X is-st X is-dn X is-vl → is-vl X is-abn


(8)   iterate-step-until-elem(cvar,step-expr,until-expr,t,dn,vl) =

          cases: eval-until(cvar,step-expr,until-expr,dn,vl)
          (bool-op¹,vl¹,Ω):
               cases: s-value(bool-op¹)
               TRUE: (vl¹,Ω)
               FALSE:
                    cases: int-st(t,dn,vl¹)
                    (vl²,Ω):
                         cases: eval-step(cvar,step-expr,dn,vl²)
                         (op³,vl³,Ω):
                              let: v = convert(s-type(cvar),op³)
                                   vl⁴ = assign(cvar,v,vl³)
                              iterate-step-until-elem(cvar,step-expr,until-expr,t,dn,vl⁴)
                         (op³,vl³,lab³): (vl³,lab³)
                    (vl²,lab²): (vl²,lab²)
          (bool-op¹,vl¹,lab¹): (vl¹,lab¹)

      refs: int-st 4, convert 4.2, assign 4.2
      note: Step-expr is evaluated twice per iteration,
            until-expr is evaluated once per iteration.
      type: is-arithm-var X is-arithm-expr X is-arithm-expr X is-st X is-dn X is-vl →
                                                                  is-vl X is-abn
```

```
(9)     eval-until(cvar,step-expr,until-expr,dn,vl) =

            eval-expr(μ0(<s-opr:GT>,
                        <s-op-1:μ0(<s-opr:MULT>,
                                   <s-op-1:μ0(<s-opr:MINUS>,
                                              <s-op-1:cvar>,
                                              <s-op-2:until-expr>)>,
                                   <s-op-2:μ0(<s-id:SIGN>,
                                              <s-type:INTG-PROC>,
                                              <s-arg-list:<step-expr>>)>)>,
                        <s-op-2:μ0(<s-type:INTG>,
                                   <s-value:0>)>)     ,dn,vl)

        refs: eval-expr 3
        type: is-arithm-var X is-arithm-expr X is-arithm-expr X is-dn X is-vl →
                    is-opt-bool-op X is-vl X is-abn


(10)    eval-step(cvar,step-expr,dn,vl) =

            eval-expr(μ0(<s-opr:PLUS>,
                        <s-op-1:cvar>,
                        <s-op-2:step-expr>),dn,vl)

        refs: eval-expr 3
        type: is-arithm-var X is-arithm-expr X is-dn X is-vl → is-opt-arithm-op X is-vl X is-abn


(11)    iterate-while(cvar,while-elem,t,dn,vl) =

            cases: eval-expr(s-init-expr(while-elem),dn,vl)
            (op¹,vl¹,Ω): let: v = convert(s-type(cvar),op¹)
                              vl² = assign(cvar,v,vl¹)
                         cases: eval-expr(s-while-expr(while-elem),dn,vl²)
                         (bool-op³,vl³,Ω):
                              cases: s-value(bool-op³)
                              FALSE: (vl³,Ω)
                              TRUE: cases: int-st(t,dn,vl³)
                                    (vl⁴,Ω): iterate-while(cvar,while-elem,t,dn,vl⁴)
                                    (vl⁴,lab⁴): (vl⁴,lab⁴)
                         (bool-op³,vl³,lab³): (vl³,lab³)
            (op¹,vl¹,lab¹): (vl¹,lab¹)

        refs: eval-expr 3, convert 4.2, assign 4.2, int-st 4
        type: is-arithm-var X is-while-elem X is-st X is-dn X is-vl → is-vl X is-abn
```

4.7 PROCEDURE STATEMENTS

Translation

(1) is-path(path) & is-proc-st•path(prog) & is-PROC•s-type•path(prog) ⊃
is-PROC•s-type•desc(s-id•path,prog) ∨ is-PROC•desc(s-id•path,prog)

(2) See also 3.2 for type procs in proc statements.

Abstract syntax

(3) is-proc-st = (<s-id:is-id>,
<s-act-par-list:is-act-par-list ∨ is-Ω>,
<s-type:is-type-proc ∨ is-PROC>)

refs: is-id 2.4, is-act-par 3.2, is-type-proc 5.4

Auxiliary Predicates

(4) is-changed-text =

This predicate is true of text satisfying is-text except that some identifiers may have been replaced by text satisfying is-act-par. Such text, obtained as a result of copying by name actual parameters, may not satisfy is-text.
(This leads to error in int-proc-body.)

Interpretation

(5) int-proc-st(t,dn,vl) =

```
cases: access(t,dn,vl)
(t¹,vl¹,Ω): cases: proc-access(t¹,dn,vl¹)
                   (op²,vl²,Ω): (vl²,Ω)
                   (op²,vl²,lab²): (vl²,lab²)
(t¹,vl¹,lab¹): (vl¹,lab¹)
```

refs: access 3, proc-access 3.2
type:is-proc-st X is-dn X is-vl → is-vl X is-abn

```
(6)   non-type-proc(t,pr-set,dn,vl) =

          int-proc-body(change-text(t,pr-set),dn,vl)

      note: Called from activate-proc 3.2, which is called from proc-access 3.2.
      type: (is-block ∨ is-code) X is-pr-set X is-dn X is-vl → is-vl X is-abn


(7)   change-text(t,pr-set) =

          cases: t
          is-code: t
          is-id: cases: t ∈ firsts(pr-set)
                 TRUE: s(t,pr-set)
                 FALSE: t
          is-block: let: red-set¹ = del-set(pr-set,intr-ids(t))
                    μ0(<s-st-list:change-text(s-st-list(t),red-set¹)>,
                       <s-decl-pt:{change-text(d,red-set¹) | d∈s-decl-pt(t)}>)
          is-proc-desc: let: id-set² = {id | (∃i)(id = elem(i,s-form-par-list(t)))}
                        μ(t;<s-body:change-text(s-body(t),del-set(pr-set,id-set²))>)
          is-set: {change-text(el,pr-set) | el ∈ t}
          is-object: t
          is-ob: μ0({<sel:change-text(sel(t),pr-set)> | is-selector(sel) & ¬is-Ω(sel(t))})

      refs: is-code 4, is-id 2.4, is-block 4.1, intr-ids 4.1, is-proc-desc 5.4
      type: is-text X is-pr-set → is-changed-text


(8)   int-proc-body(t,dn,vl) =

          cases: t
          is-unlab-st: int-unlab-st(t,dn,vl)
          is-code: int-code(t,dn,vl)
          T: error

      refs: is-unlab-st 4, int-unlab-st 4, is-code 5.4
      error: If the result of change-block is not a well formed prog.
      type: is-changed-text X is-dn X is-vl → is-vl X is-abn


(9)   int-code(t,dn,vl) =

      This function is implementation defined.

      type: is-code X is-dn X is-vl → is-vl X is-abn
```

Translation

(1) The <declarations> in a <block head> are collected into a set.

(2) The <identifiers> introduced in a single <declaration> are collected into a set.

(3) The Translator rejects any <programs> in which errors of duplication would be hidden by (1) and (2) (e.g. real x,x; real y; real y;).

(4) The Translator rejects abstract programs in which an identifier is declared (or used as a label) more than once in the same scope:-

```
is-block(t) & decl-1 ∈ s-decl-pt(t) & decl-2 ∈ s-decl-pt(t) & decl-1 ≠ decl-2 ⊃
   disj(s-id-set(decl-1),s-id-set(decl-2)) &
   disj(s-id-set(decl-1),local-labs(t)) &
   (id-1 ∈ local-labs(t) & id-2 ∈ local-labs(t) & id-1 ≠id-2 ⊃
        make-st-sel(id-1,t) ≠ make-st-sel(id-2,t))
```

refs: is-block 4.1, local-labs 4.1, make-st-sel 4

Abstract syntax

(5) is-desc = is-var-desc ∨ is-array-desc ∨ is-switch-desc ∨ is-proc-desc

refs: is-var-desc 5.1, is-array-desc 5.2, is-switch-desc 5.3, is-proc-desc 5.4

(6) is-decl = (<s-id-set:is-id-set>,
<s-desc:is-desc>)

refs: is-id 2.4
note: The s-id-set of switch or procedure declarations will always be a unit set.

Auxiliary predicates

(7) is-label-desc = (<s-type:is-LABEL>)

Interpretation

(8) augment-dn(decl-set,dn) =

mod-set(dn,{<id,make-den(desc)>|(∃decl)(decl∈decl-set & desc=s-desc(decl) & id∈s-id-set(decl))})

type: is-intr-set X is-dn → is-dn

(9) make-den(desc) =

cases: desc
is-var-desc: μ_0(<s-type:desc>)
is-array-desc: desc
is-switch-desc: μ_0(<s-switch-list:desc>,<s-type:SWITCH>)
is-label-desc: desc
is-proc-desc: desc

refs: is-var-desc 5.1, is-array-desc 5.2, is-switch-desc 5.3, is-proc-desc 5.4
type: (is-desc ∨ is-label-desc) → is-den

5.1 TYPE DECLARATIONS

Abstract syntax

(1) is-arithm = is-INTG ∨ is-REAL

(2) is-type = is-arithm ∨ is-BOOL

(3) is-var-desc = is-type

Interpretation

(4) see make-den :- is-var-desc

refs: make-den 5

5.2 ARRAY DECLARATIONS

Translation

(1) The Translator rejects any program in which array bound expressions use local names:-

```
is-block(block) ⊃ (∀decl)(decl ∈ s-decl-pt(block) & is-array-desc(s-desc(decl)) ⊃
                         (sel•s-bounds(s-desc(decl)) = id-1 ⊃ ¬(id-1 ∈ intr-ids(block))))
```

refs: is-block 4, intr-ids 4.1
note: Declarations of the form array are treated as real array (see A.R. 5.2.3.3).

Abstract syntax

(2) is-arithm-array = is-REAL-ARRAY ∨ is-INTG-ARRAY

(3) is-type-array = is-arithm-array ∨ is-BOOL-ARRAY

(4) is-bound-pair = (<s-lbd:is-arithm-expr>,
<s-ubd:is-arithm-expr>)

refs: is-arithm-expr 3.3

(5) is-array-desc = (<s-type:is-type-array>,
<s-bounds:(is-bound-pair-list & ¬is-<>)>)

Auxiliary predicates

(6) is-array-decl = (<s-id-set:is-id-set>,
<s-desc:is-array-desc>)

refs: is-id 2.4

Interpretation

(7) see make-den :- is-array-desc

refs: make-den 5

(8) eval-array-decls(decl-set,dn,vl) =

cases: decl-set
(∀decl¹)(decl¹ ∈ decl-set & is-array-decl(decl¹) ⊃ is-array-den(s-desc(decl¹))):
(decl-set,vl,Ω)
T: for some decl¹ ∈ decl-set & is-array-decl(decl¹) & ¬is-array-den(s-desc(decl¹))
cases: eval-array-bds(s-desc(decl¹),dn,vl)
(desc²,vl²,Ω): cases: eval-array-decls(decl-set - {decl¹},dn,vl²)
(decl-set³,vl³,Ω): (decl-set³ ∪ {μ(decl²;<s-desc:desc²>)},vl³,Ω)
(decl-set³,vl³,lab³): (Ω,vl³,lab³)
(desc²,vl²,lab²): (Ω,vl²,lab²)

type: is-decl-set X is-dn X is-vl → is-opt-decl-set X is-vl X is-abn

```
(9)     eval-array-bds(desc,dn,vl) =

            cases: access(s-bounds(desc),dn,vl)
            (abds,vl¹,Ω):
                 let: ebds = μ0({<sel-1•elem(i):apply(sel-1•elem(i)(abds))> |
                                        (sel-1 = s-lbd ∨ sel-1 = s-ubd) & 1≤i≤length(abds)})
                 cases: ebds
                 (∀i)(1≤i≤length(ebds) ⊃ s-lbd(elem(i,ebds)) ≤ s-ubd(elem(i,ebds))):
                      (μ(desc;<s-bounds:ebds>),vl¹,Ω)
                 T: error
            (abds,vl¹,lab¹): (Ω,vl¹,lab¹)

        refs: access 3, apply 3
        error: No array is defined if any upper bound is less than the corresponding lower bound.
        type: is-array-desc X is-dn X is-vl → is-opt-array-den X is-vl X is-abn
```

5.3 SWITCH DECLARATIONS

Abstract syntax

```
(1)     is-switch-desc = (is-des-expr-list & ¬is-<>)

           refs: is-des-expr 3.5
```

Interpretation

(2) see make-den :- is-switch-desc

refs: make-den 5

5.4 PROCEDURE DECLARATIONS

Translation

(1) The entries in the <specification part> are collected into a set which associates one copy of the appropriate <specifier> with each <identifier>.

(2) The <identifiers> in the <value part> are collected into a set.

(3) The Translator rejects any <program> in which errors of duplication would be hidden by (1) and (2) (e.g. procedure p(x); value x,x; real x; real x; ...).

(4) The <specifier> array is treated as real array.

(5) The body of a procedure (except the case of code) is formed into a block (see A.R. 5.4.3).

(6) The Translator rejects abstract programs in which an identifier appears in more than one spec:-

```
is-proc-desc(pd) & spec-1 ∈ s-spec-pt(pd) & spec-2 ∈ s-spec-pt(pd) ⊃
    (s-id(spec-1) = s-id(spec-2) ⊃ spec-1 = spec-2)
```

(7) The Translator rejects abstract programs in which the same identifier occurs in more than one position in the form-par-list:-

```
is-proc-desc(pd) & elem(i,s-form-par-list(pd)) = elem(j,s-form-par-list(pd)) ⊃ i=j
```

(8) The Translator rejects any programs in which a formal parameter does not have a corresponding specifier (see A.R. 5.4.5) :-

```
is-proc-desc(pd) & 1≤i≤length•s-form-par-list(pd) ⊃
    (∃spec)(spec∈s-spec-pt(pd) & s-id(spec) = elem(i)•s-form-par-list(pd))
```

(9) The Translator rejects any programs in which an identifier appears in the value part but not in the formal parameter list :-

```
is-proc-desc(pd) & id∈s-value-pt(pd) ⊃ (∃i)(elem(i)•s-form-par-list = id)
```

(10) The Translator rejects abstract programs in which procedure, string or switch parameters appear in the value-pt:-

```
is-proc-desc(pd) & spec ∈ s-spec-pt(pd) &
    (is-type-proc(s-specifier(spec)) ∨
     is-PROC(s-specifier(spec)) ∨
     is-STRING(s-specifier(spec)) ∨
     is-SWITCH(s-specifier(spec))) ⊃ ¬(s-id(spec) ∈ s-value-pt(pd))
```

(11) The following function is used to determine the type of references etc. :-

```
desc-proc(id,path-el•path,t) =

        cases: id
        (∃spec)(spec ∈ s-spec-pt•path-el•path(t) & id = s-id(spec)):
             let: spec¹ = (ι spec)(spec∈s-spec-pt•path-el•path(t) & id=s-id(spec))
             s-specifier(spec¹)
        T: desc-1(id,path,t)

   type: is-id X is-path X is-program → (is-specifier ∨ is-desc ∨ is-label-desc)
```

Abstract syntax

(12) is-code =

This predicate is implementation defined. The objects satisfying it are distinct elementary objects (thus is-sel(sel) & is-code(t) ⊃ is-Ω•sel(t)).

(13) is-type-proc = is-REAL-PROC ∨ is-INTG-PROC ∨ is-BOOL-PROC

(14) is-specifier = is-type ∨ is-type-array ∨ is-type-proc ∨ is-PROC ∨ is-LABEL ∨ is-STRING ∨ is-SWITCH

refs: is-type 5.1, is-type-array 5.2

(15) is-spec = (<s-id:is-id>, <s-specifier:is-specifier>)

refs: is-id 2.4

(16) is-proc-desc = (<s-type:is-type-proc ∨ is-PROC>, <s-form-par-list:is-id-list>, <s-spec-pt:is-spec-set>, <s-value-pt:is-id-set>, <s-body:is-block ∨ is-code>)

refs: is-id 2.4, is-block 4
note: The body of a procedure, unless code, is made into a block.

Interpretation

(17) see mk-den :- is-proc-desc

refs: make-den 5

Acknowledgements

Thanks are due to IFIP W.G. 2.1 (The Algol working group) for kindly allowing the reproduction of the Revised Algol Report in the form presented. Also to members of the IBM Laboratory, Vienna for useful discussions on general problems of the definition method and on specifics of their Algol definition.

References

1. Naur, P. (Ed.) "Revised Report on the Algorithmic language Algol 60"
Comm ACM Vol. 6 No. 1 pp 1-17 (Jan 1963)

2. Duncan, F.G. "ECMA Subset of Algol 60"
Comm ACM Vol. 6 No. 10 pp 595-597 (Oct 1963)

3. Knuth, D.E. "The Remaining Trouble spots in Algol 60"
Comm ACM Vol. 10 No. 10 pp 611-618 (Oct 1967)

4. Lucas, P. and Walk, K. "On the Formal Description of PL/I"
Annual Review in Automatic Programming Vol. 6 Part 3 Pergamon Press (1969)

5. Lucas, P., Lauer, P. and Stigleitner, H.
"Method and Notation for the Formal Definition of Programming Languages"
IBM Lab. Vienna, Tech. Report TR25.087 (June 1968)

6. Lauer, P. "Formal Definition of Algol 60"
IBM Lab. Vienna, Tech. Report TR25.088 (Dec 1968)

7. Henhapl, W. and Jones, C.B. "On the Interpretation of GOTO Statements in the ULD"
IBM Lab. Vienna, Lab. note LN25.3.065 (March 1970)

8. Lucas, P. "Two Constructive Realisations of the Block Concept and Their equivalence"
IBM Lab. Vienna, Tech. Report TR25.085 (Jan 1968)

9. Jones C.B. and Lucas, P. "Proving Correctness of Implementation Techniques"
Symposium on Semantics of Algorithmic Languages (Ed. Engeler, E.)
Lecture notes in Mathematics 188, Springer-Verlag pp 178-211

10. Bekic, H. "On the Formal Definition of Programming Languages"
Proceedings of International Computing Symposium Bonn (1970)

11. Burstall,R.M. "Proving Properties of Programs by Structural Induction"
Comp Journal Vol. 12 No. 1 pp 41-48 (Feb 1969)

APPENDIX: Cross Reference Index

NAME	DCLN	USED IN
change-text	4.7(7)	4.1(14),4.7(6),4.7(7)
construct-pairs	4.1(12)	4.1(11),4.1(12)
convert	4.2(21)	3(6),3.1(23),3.2(18),4.2(20),4.6(7),4.6(8),4.6(11)
convert-array	3.2(19)	3.2(18)
convert-array-el	3.2(20)	3.2(19)
convert-one-sub	3.1(23)	3.1(22)
convert-subs	3.1(22)	3.1(21),4.2(18)
cue-int-st	4(9)	4(7),4(9),4(10),4.1(17)
cue-int-st-list	4.1(17)	4(10),4.1(16),4.1(17)
cue-int-unlab-st	4(10)	4(9)
del-set	1(12)	4.1(18),4.6(6),4.7(7)
desc	1(51)	
desc-block	4.1(5)	1(52)
desc-proc	5.4(11)	1(52)
desc-1	1(52)	1(51),1(52),4.1(5),5.4(11)
disj	1(59)	
entier	1(49)	3.2(20),3.3(38),4.2(21)
epilogue	4.1(18)	3.2(16),3.2(23),4.1(10)
eval-act-par	3.2(18)	3.2(17)
eval-act-par-list	3.2(17)	3.2(16)
eval-array-bds	5.2(9)	5.2(8)
eval-array-decls	5.2(8)	4.1(10),5.2(8)

NAME	DCLN	USED IN
eval-expr	3(4)	4.2(16),4.2(19),4.3(2),4.5(3),4.6(7),4.6(9),4.6(10) 4.6(11)
eval-lp	4.2(18)	4.2(17),4.6(5)
eval-lp-list	4.2(17)	4.2(16),4.2(17)
eval-step	4.6(10)	4.6(8)
eval-subs	4.2(19)	4.2(18),4.2(19)
eval-until	4.6(9)	4.6(8)
firsts	1(14)	4.7(7)
fn-access	3.2(14)	3(4),3(8)
head	1(43)	4.2(17),4.2(19),4.2(22),4.6(6)
insert-ret	3.2(24)	3.2(23),3.2(24)
int-assign-st	4.2(16)	4(8)
int-block	4.1(10)	4(8),4.1(9)
int-block-body	4.1(15)	4.1(10)
int-code	4.7(9)	4.7(8)
int-cond-st	4.5(3)	4(8)
int-for-st	4.6(5)	4(8)
int-goto-st	4.3(2)	4(8)
int-proc-body	4.7(8)	4.7(6)
int-proc-st	4.7(5)	4(8)
int-program	4.1(9)	
int-st	4(7)	4(9),4.1(16),4.5(3),4.6(7),4.6(8),4.6(11)
int-st-list	4.1(16)	4(8),4.1(15),4.1(16),4.1(17)

NAME	DCLN	USED IN
int-unlab-st	4(8)	4(7),4.7(8)
intg-arithm-infix-value	3.3(38)	3.3(32)
intg-power-opr	3.3(33)	3.3(32)
intg-power-opr-val	3.3(34)	3.3(33)
intr-ids	4.1(3)	4.1(10),4.1(13),4.7(7)
is-(pred)-list	1(47)	
is-(pred)-set	1(8)	
is-abn	1(53)	
is-act-par	3.2(8)	3.2(9),3.2(10),3.2(11),4.7(3)
is-activated-fn	4.2(14)	3.2(24)
is-arithm	5.1(1)	3.2(18),3.3(32),5.1(2)
is-arithm-array	5.2(2)	3.2(18),5.2(3)
is-arithm-array-op	1(31)	1(33),3.2(18)
is-arithm-array-val	1(26)	1(28),1(31)
is-arithm-cond-expr	3.3(26)	3.3(27),3.3(29)
is-arithm-const	2.5(7)	2(3)
is-arithm-expr	3.3(23)	3(2),3(9),3.1(8),3.1(10),3.1(12),3.4(7),3.5(4),4.2(10) 4.2(11),4.6(1),4.6(2),4.6(3),4.6(7),5.2(4)
is-arithm-infix-expr	3.3(25)	3.3(29)
is-arithm-infix-opr	3.3(28)	
is-arithm-op	1(37)	1(38),3.2(18)
is-arithm-prefix-expr	3.3(24)	3.3(29)
is-arithm-prefix-opr	3.3(2)	3.3(3),3.3(4)

NAME	DCLN	USED IN
is-arithm-relat-expr	3.4(7)	3.4(8),3.4(9)
is-arithm-relat-opr	3.4(6)	3.4(7)
is-arithm-val	1(24)	1(25),1(26)
is-arithm-var	3.1(15)	4.6(4)
is-array-decl	5.2(6)	5.2(8)
is-array-den	1(18)	1(22),5.2(8)
is-array-desc	5.2(5)	4.1(10),5(5),5(9),5.2(6)
is-array-name	3.2(7)	3(7),3(8),3.2(8)
is-array-op	1(33)	1(40),3.2(22)
is-array-val	1(28)	1(29)
is-assign-st	4.2(13)	3.2(24),4(5),4(8)
is-basic-symbol	2(2)	1(1),2.6(1)
is-block	4.1(7)	1(21),1(52),4(3),4(4),4(5),4(8),4.1(8),4.1(13),4.7(7) 5.4(16)
is-bool-activated-fn	4.2(8)	4.2(9),4.2(14)
is-bool-array-op	1(32)	1(33),3.2(18)
is-bool-array-val	1(27)	1(28),1(32)
is-bool-assign-st	4.2(12)	4.2(13)
is-bool-cond-expr	3.4(5)	3.3(27),3.4(8),3.4(9)
is-bool-const	2.2(2)	2(3)
is-bool-expr	3.4(8)	3(2),3(9),3.2(21),3.3(16),3.3(17),3.3(18),3.3(20),3.4(2) 3.4(4),3.4(5),3.5(5),4.2(12),4.5(2),4.6(1)
is-bool-funct-ref	3.2(11)	3.2(12),3.4(8)

NAME	DCLN	USED IN
is-bool-infix-expr	3.4(4)	3.4(8),3.4(9)
is-bool-infix-opr	3.4(3)	3.4(4)
is-bool-lp	4.2(9)	4.2(12),4.2(15),4.2(20)
is-bool-op	1(36)	1(38),3.2(18),3.4(8)
is-bool-prefix-expr	3.4(2)	3.4(8),3.4(9)
is-bool-simple-var	3.1(11)	3.1(16),3.1(17)
is-bool-subscr-var	3.1(12)	3.1(16),3.1(18)
is-bool-val	2.2(1)	1(25),1(27),1(36),2.2(2)
is-bool-var	3.1(16)	3.4(8),4.2(9)
is-bound-pair	5.2(4)	5.2(5)
is-changed-text	4.7(4)	
is-code	5.4(12)	1(21),3.2(24),4.7(7),4.7(8),5.4(16)
is-comp-st	4.1(6)	4(5),4(8),4(10)
is-cond-expr	3.3(27)	3(3),3(6),3(7)
is-cond-st	4.5(2)	4(5),4(8),4(10)
is-const	2(4)	
is-decl	5(6)	1(54),4.1(7)
is-den	1(22)	1(23)
is-des-cond-expr	3.5(5)	3.3(27),3.5(8)
is-des-expr	3.5(8)	1(20),3(2),3(9),3.2(21),3.5(5),4.3(1),5.3(1)
is-desc	5(5)	5(6)
is-dn	1(23)	

NAME	DCLN	USED IN
is-dummy-st	4.4(2)	4(5),4(8)
is-eb	1(17)	1(18),1(31),1(32)
is-expr	3(2)	3(3),3.2(8)
is-fn-ret	3.2(13)	
is-for-elem	4.6(3)	4.6(4)
is-for-st	4.6(4)	4(5),4(8),4(10)
is-funct-ref	3.2(12)	3(3),3(4),3(5),3(6),3(7),3(8)
is-goto-st	4.3(1)	4(5),4(8)
is-id	2.4(1)	1(1),1(10),1(19),1(21),1(23),1(30),1(39),1(53),1(57) 3.1(7),3.1(8),3.1(9),3.1(10),3.1(11),3.1(12),3.2(6) 3.2(7),3.2(9),3.2(10),3.2(11),3.5(4),3.5(6),3.5(7),4(6) 4.1(12),4.2(4),4.2(6),4.2(8),4.7(3),4.7(7),5(6),5.2(6) 5.4(15),5.4(16)
is-idpr	1(10)	
is-intg-activated-fn	4.2(6)	4.2(7),4.2(14)
is-intg-assign-st	4.2(11)	4.2(13)
is-intg-cond-expr	3.3(20)	3.3(22),3.3(26)
is-intg-const	2.5(5)	2.5(7)
is-intg-expr	3.3(22)	3.2(21),3.3(4),3.3(7),3.3(8),3.3(9),3.3(10),3.3(13) 3.3(14),3.3(17),3.3(18),3.3(20),3.3(23),3.3(29),3.3(33)
is-intg-funct-ref	3.2(10)	3.2(12),3.3(22)
is-intg-infix-expr	3.3(15)	3.3(22),3.3(25)
is-intg-infix-expr-1	3.3(13)	3.3(15)
is-intg-infix-expr-2	3.3(14)	3.3(15)
is-intg-infix-opr	3.3(12)	3.3(13),3.3(28),3.3(32)

NAME	DCLN	USED IN
is-intg-lp	4.2(7)	4.2(11),4.2(15),4.2(20)
is-intg-op	1(35)	1(37),3.3(22)
is-intg-prefix-expr	3.3(4)	3.3(22),3.3(24)
is-intg-simple-var	3.1(9)	3.1(14),3.1(17)
is-intg-subscr-var	3.1(10)	3.1(14),3.1(18)
is-intg-val	2.5(2)	1(1),1(17),1(24),1(26),1(27),1(35),1(49),2.5(3),2.5(5) 3.1(24)
is-intg-var	3.1(14)	3.1(15),3.3(22),4.2(7)
is-intr	1(54)	
is-lab-sel	1(56)	
is-lab-selector	1(55)	1(56)
is-label-const	3.5(7)	2(4)
is-label-decl	1(57)	1(54)
is-label-den	1(19)	1(22)
is-label-desc	5(7)	1(57),5(9)
is-label-op	1(39)	1(40),3.2(16),3.2(18),3.2(22),3.5(8)
is-label-var	3.5(6)	3.1(17),3.5(8)
is-list	1(42)	1(47)
is-lp	4.2(15)	
is-non-neg-intg-const	2.5(6)	3.3(9),3.3(14),3.3(33)
is-non-neg-intg-val	2.5(3)	2.5(6)
is-non-type-proc-name	3.2(6)	3.2(8)
is-ob	1(2)	1(9),3(6),3.2(24),4.7(7)

NAME	DCLN	USED IN
is-object	1(1)	3(6),3(7),3.2(24),4.7(7)
is-op	1(40)	3(3),3(6),3(7),3(9)
is-op-expr	3(3)	3(3),3(6)
is-opt-(pred)	1(62)	
is-path	1(6)	1(58)
is-path-el	1(5)	1(6)
is-pr	1(9)	
is-proc-den	1(21)	1(22),4.1(11)
is-proc-desc	5.4(16)	1(52),4.1(13),4.7(7),5(5),5(9)
is-proc-st	4.7(3)	3(5),4(5),4(8)
is-program	4.1(8)	1(58)
is-real-activated-fn	4.2(4)	4.2(5),4.2(14)
is-real-assign-st	4.2(10)	4.2(13)
is-real-cond-expr	3.3(19)	3.3(21),3.3(26)
is-real-cond-expr-1	3.3(16)	3.3(19)
is-real-cond-expr-2	3.3(17)	3.3(19)
is-real-cond-expr-3	3.3(18)	3.3(19)
is-real-const	2.5(4)	2.5(7)
is-real-expr	3.3(21)	3.2(21),3.3(3),3.3(6),3.3(7),3.3(8),3.3(16),3.3(17) 3.3(18),3.3(23),3.3(29)
is-real-funct-ref	3.2(9)	3.2(12),3.3(21)
is-real-infix-expr	3.3(11)	3.3(21),3.3(25)
is-real-infix-expr-1	3.3(6)	3.3(11)

NAME	DCLN	USED IN
is-real-infix-expr-2	3.3(7)	3.3(11)
is-real-infix-expr-3	3.3(8)	3.3(11)
is-real-infix-expr-4	3.3(9)	3.3(11)
is-real-infix-expr-5	3.3(10)	3.3(11)
is-real-infix-opr	3.3(5)	3.3(6),3.3(7),3.3(8),3.3(28),3.3(32)
is-real-lp	4.2(5)	4.2(10),4.2(15),4.2(20)
is-real-op	1(34)	1(37),3.3(21)
is-real-prefix-expr	3.3(3)	3.3(21),3.3(24)
is-real-simple-var	3.1(7)	3.1(13),3.1(17)
is-real-subscr-var	3.1(8)	3.1(13),3.1(18)
is-real-val	2.5(1)	1(1),1(24),1(34),2.5(4)
is-real-var	3.1(13)	3.1(15),3.3(21),4.2(5)
is-sel	1(4)	
is-selector	1(3)	1(5),1(60),3(6),3(7),3.2(24),4.7(7)
is-set	1(7)	1(1),1(5),1(8),3.2(24),4.7(7)
is-simple-val	1(25)	1(29),3.2(13)
is-simple-var	3.1(17)	3(6),3(7),3(8),3.1(19),4.2(18),4.2(23),4.6(6)
is-spec	5.4(15)	1(21),5.4(16)
is-specifier	5.4(14)	5.4(15)
is-st	4(6)	4.1(6),4.1(7),4.5(2),4.6(4)
is-step-until-elem	4.6(2)	4.6(3),4.6(7)
is-string	2.6(2)	2.6(1),3.2(8),3.2(21)

NAME	DCLN	USED IN
is-string-elem	2.6(1)	2.6(2)
is-subscr-var	3.1(18)	3(7),3(8),3.1(19),4.2(18),4.2(23),4.6(6)
is-switch-den	1(20)	1(22)
is-switch-des	3.5(4)	3(3),3(6),3.5(8)
is-switch-desc	5.3(1)	5(5),5(9)
is-text	1(58)	
is-type	5.1(2)	1(16),3.1(20),3.2(13),5.1(3),5.4(14)
is-type-array	5.2(3)	1(18),3.2(7),3.2(21),5.2(5),5.4(14)
is-type-const	2(3)	2(4)
is-type-den	1(16)	1(22)
is-type-op	1(38)	1(40),3.2(22)
is-type-proc	5.4(13)	1(21),3.2(16),3.2(21),4.7(3),5.4(14),5.4(16)
is-unlab-st	4(5)	4(6),4.7(8)
is-val	1(29)	1(30)
is-value-parm	3(10)	3(6),3(7),3.2(15),3.2(16)
is-var	3.1(19)	3(3)
is-var-desc	5.1(3)	5(5),5(9)
is-vl	1(30)	
is-while-elem	4.6(1)	4.6(3),4.6(7)
iterate-for	4.6(7)	4.6(6)
iterate-for-list	4.6(6)	4.6(5),4.6(6)
iterate-step-until-elem	4.6(8)	4.6(7),4.6(8)

NAME	DCLN	USED IN
seconds	1(15)	3.2(16),4.1(10),4.1(11)
self-mult	3.3(35)	3.3(34),3.3(35)
sign	1(48)	1(50),3.3(38)
simp-var-access	3.1(20)	3(8)
subscr-var-access	3.1(21)	3(8)
tail	1(44)	4.2(17),4.2(19),4.2(22),4.6(6)
type-proc	3.2(23)	3.2(16)
val-den	3.2(22)	3.2(16)