A FORMAL DEFINITION OF ALGOL 60 AS DESCRIBED IN THE 1975 MODIFIED REPORT

Wolfgang Henhapl & Cliff B.Jones

Abstract:

This paper provides a formal definition of a version of the ALGOL 60 programming language. In particular the definition uses the denotational approach and the meta-language presented in this volume (-- known within the Vienna Laboratory as "*META-IV*"). As well as exemplifying the meta-language, (yet) another definition of ALGOL 60 is justified by the recent revision of the language which resolved most of the open points in the earlier "Revised Report".

CONTENTS

0. INTRODUCTION

For many years the official description of ALGOL 60 has been the "Revised Report" (Naur 63). Not only the language, but also its extremely precise description have been seen as a reference point. There were, however, a number of known unresolved problems and most of these have been eliminated by the recent modifications given in De Morgan 75. A number of formal definitions exist for the language of the revised report: this paper presents a denotational definition of the language as understood from the modified report (MAR).

Before making some introductory remarks on the definition, three points will be made about the language itself (as in De Morgan 75). Firstly the modifications have followed the earlier "*ECMA subset*" by making the language (almost) statically typed. Although all parameters must now be specified, there is still no way of fixing the dimensions of array parameters nor the required parameter types of procedure or function parameters (cf. ALGOL 68). In connection with this it could be observed that the parameter matching rules of section 4.7.5.5 are somewhat difficult. In particular the definition given below assumes that, for "*by name*" passing of arithmetic expressions, the types must match exactly!

The third observation is simply one of surprise. The decision to restrict the control variable of a *<for statement>* to be a *<variable identifier>* (i.e. not a subscripted variable) may or may not be wise: but the argument that *<for statement>* can now be defined by expansion within ALGOL is surely dangerous. The definition given here would have had no difficulty treating the more general case because the concept of location has anyway to be introduced for other purposes.

Two of the major points resolved by the modifications are the meaning of "*own*" variables and the provision of a basic set of input-output functions: particular attention has been given to these points in the formal definition below. In fact, the treatment of own given here is more detailed than that for PL/I static variables in Walk 69. Rather than perform name changes and generate dummy declarations in the outermost block, an extra environment component is used here to retain a mapping from (additional) unique names to their locations. This "*Own-env*" is used in generating the denotations for own variables for insertion in the local "*Env*". The input-output functions are defined to change the "Channel" components of the state (Σ).

Much of the definition which follows should be easy to read after a study of the example given in Jones 78a. The treatment of goto is similar to that given there (for discussion see Jones 75) except that the use of the "*tixe*" combinator has been held here to a minimum. Instead of the use in *i-block*, an argument can be made for localizing the effects of goto at the level of *comp-stmt*, *cond-stmt* and *stmt*. In a version which used "*tixe*" at all three levels it was found advantageous to merge the "*cue*" and "*i*" functions (cf. Jones 78b).

As has been discussed elsewhere in this volume, the definition of arbitrary order of evaluation has not been addressed: had it been, one would, for example, have to show that the elements of an expression can be evaluated in any order.

With the aid of the list of abbreviations given at the end of this introduction, the abstract syntax and context conditions should be straight forward. Notice that, although the abstract syntax itself is a "*context-free*" production system, context dependant typing (e.g. Array-name) is used and secured by the context conditions. (Notice by-value variables are, in fact, non-by-name - i.e. by-value includes non-parameters).

The semantic objects are the key to the definition. States contain two components, one of which stores scalar values for each current scalar location (the division to the sub-types of *Sc-loc* is not necessary, it is only made to fit the implementation viewpoint), the other of which contains an abstraction of the objects which can be accessed by the input-output statements. State transformations are of the type required by the "*exit treatment*" of goto.

The composite objects *Stmt-env* and *Expr-env* are introduced solely as abbreviations; *Own-env* has been mentioned above; the real interest lies in the denotations which can be stored in *Env*. *Type-dens* are obviously scalar locations. A function procedure which is activated sets up a value to be returned by assigning to the *Atv-proc-id*: again a scalar location is the appropriate denotation. Array denotations store the (one-one) mapping from all possible subscript values to scalar locations; notice that the constraint requires that the domain of an array denotation is "*dense*". Procedure denotations are functions which, for given arguments and a current set of activations, yield transformations. Notice that the *Act-parm-dens* carry type information with them for checking within the *Proc-den*. Switch denotations are similar.

The very general parameter passing "by name" permitted in ALGOL requires that the *By-name-dens* are rather like procedure denotations. Because formal parameter names can occur as Destinations for assignment statements it is also necessary to know whether a by-name parameter can be evaluated to a location or not (cf. *e-parm-expr*, *e-var-ref*, *e-var*). Furthermore, the question whether a parameter is to be passed by-name or by-value is not decidable at the point of call. Thus all parameters are passed by-name and the *Proc-den* has the task of creating the locations to store by-value parameters (cf. *e-proc-decl*, *e-val-parm*).

The classes *Loc* and *Val* are auxiliary and are used only in type clauses.

Given an understanding of the semantic objects the reader should be able to tackle section 4. Remember that for "splitting" rules only a type clause is given.

ACKNOWLEDGEMENT.

Returning the compliment to Peter Mosses, one of the authors would like to acknowledge that a part of the incentive to write this definition was the hope to provide an equally abstract but more readable definition than that in Mosses 74!

ABBREVIATIONS.

*Abn*ormal component	*const*ant	*op*e*r*ator
*act*ual	*decl*aration	*par*a*m*eter
*act*ivation *id*entifier	*den*otation	*proc*edure
*activ*ated	*des*ignator	*ref*erence
*arith*metic	*descr*iptor	*sc*alar
*assign*ment	*dest*ination	*spec*ification
by-*n*ame	*elem*ent	*st*ate*m*en*t*
*Bool*ean	*env*ironment	*subscr*ipted
by-*v*alue	*expr*ession	*tr*ansformation
*char*acter	*funct*ion	*unlab*elled
*comp*ound	*id*entifier	*val*ue
*cond*itional	*int*eger	*var*iable

1. ABSTRACT SYNTAX

1.1 Definitions

```
Program      :: Block
Block        :: s-dp:Decl-set  s-sl:Stmt*
Stmt         :: s-lp:Id-set  s-sp:Unlab-stmt
Unlab-stmt =  Comp-stmt | Block | Assign-stmt | Goto-stmt |
              Dummy-stmt | Cond-stmt | For-stmt | Proc-stmt

Comp-stmt         :: Stmt*
Assign-stmt       :: s-lpl:Destin+  s-rp:Expr
Destin            :: s-tg:Left-part  s-tp:Type
Left-part         =  Var | Atv-proc-id
Atv-proc-id       :: Id
Goto-stmt         :: Expr
Dummy-stmt        :: DUMMY
Cond-stmt         :: s-dec:Expr  s-th:Stmt  s-el:Stmt
For-stmt          :: s-cv:Var  s-cvtp:Type  s-fl:For-list-elem+  s-b:Block
For-list-elem     =  Expr-elem | While-elem | Step-until-elem
Expr-elem         :: Expr
While-elem        :: s-in:Expr  s-wh:Expr
Step-until-elem   :: s-in:Expr  s-st:Expr  s-un:Expr
Proc-stmt         :: (Proc-des | Funct-des)
Proc-des          :: s-pn:Id  s-app:Act-parm*

Expr            =  Type-const | Var-ref | Label-const | Switch-des | Funct-des
                   Prefix-expr | Infix-expr | Cond-expr
Type-const      =  Bool-const | Arithm-const
Bool-const      :: Bool
Arithm-const    =  Real-const | Int-const
Real-const      :: Real
Int-const       :: Int
Var-ref         :: Var
Var             =  Simple-var | Subscr-var
Simple-var      =  Simple-var-bn | Simple-var-bv
Simple-var-bn   :: s-nm:Id
Simple-var-bv   :: s-nm:Id
Subscr-var      =  Subscr-var-bn | Subscr-var-bv
Subscr-var-bn   :: s-nm:Id  s-sscl:Expr+
Subscr-var-bv   :: s-nm:id  s-sscl:Expr+
```

```
Label-const    :: Id
Switch-des     :: s-id:Id  s-ssc:Expr

Funct-des      :: s-nm:Id  s-app:Act-parm*
Act-parm       :: s-v:Act-parmv  s-tp:Specifier
Act-parmv      =  Parm-expr | Array-name | Switch-name | Proc-name | String
Parm-expr      :: Expr
Array-name     :: Id
Switch-name    :: Id
Proc-name      :: Id
String         = Char*
Char           = Implementation defined set

Prefix-expr    :: s-opr:Prefix-opr  s-op:Expr
Prefix-opr     =  REAL-PLUS | REAL-MINUS |
                  INT-PLUS | INT-MINUS | NOT
Infix-expr     :: s-op1:Expr  s-opr:Infix-opr  s-op2:Expr
Infix-opr      =  REAL-ADD | REAL-SUB | REAL-MULT | REAL-DIV |
                  INT-ADD | INT-SUB | INT-MULT | INT-DIV |
                  REAL-EXP | REAL-INT-EXP | INT-EXP |
                  LT | LE | EQ | NE | GE | GT | IMPL | EQU | AND | OR
Cond-expr      :: s-dec:Expr  s-th:Expr  s-el:Expr
Decl           =  Type-decl | Array-decl | Switch-decl | Proc-decl
Type-decl      :: s-id:Id  s-oid:[Own-id]  s-desc:Type
Array-decl     :: s-id:Id  s-oid:[Own-id]  s-tp:Type  s-bdl:Bound-pair+
Bound-pair     :: s-lbd:Expr  s-ubd:Expr
Switch-decl    :: s-id:Id  s-el:Expr+
Proc-decl      :: s-id:Id
                  s-tp:(Type | PROC)
                  s-fpl:Id*
                  s-vids:Id-set
                  s-spm:(Id→Specifier)
                  s-body:(Block | Code)
Specifier      =  Type | Type-array | Type-proc | PROC | LABEL | STRING | SWITCH
Type-array     :: Type
Type-proc      :: Type
Type           =  Arithm | BOOL
Arithm         =  INT | REAL
Code           :: Tr
Tr                see "semantic objects"
```

Own-id		infinite set
Id		infinite set

Real	=	the set of rational numbers with the usual arithmetic
Int	=	the set of integers (embedded in *Real*)
Bool	=	TRUE | FALSE

Standard-proc-names	=	*Real-funct-names* | *Int-funct-names* | *Proc-names*
Real-funct-names	=	*"abs"* | *"sqrt"* | *"sin"* | *"cos"* | *"arctan"* | *"ln"* | *"exp"* | *"maxreal"* | *"minreal"* | *"epsilon"*
Int-funct-names	=	*"iabs"* | *"sign"* | *"entier"* | *"length"* | *"maxint"* |
Proc-names	=	*"inchar"* | *"outchar"* | *"outstring"* | *"outterminator"* *"stop"* | *"fault"* | *"ininteger"* | *"outinteger"* | *"inreal"* | *"outreal"*

Comment: The quotes around the standard-procedure names indicate the translated version of the identifiers.

1.2 Translator Notes.

Although neither the concrete syntax of ALGOL 60, nor its translation to objects of the abstract form are formally specified, a number of points should be borne in mind:

- Concrete delimiters, comments etc. are dropped.

- Within expressions, brackets and rules of operator precedence are used to choose the appropriate tree form of "*expr*".

- If the (concrete) outer block was labelled, the translator embeds it in another (unlabelled) block.

- The body of a procedure (which is not code) is always a block in the abstract form; the translator generates this block if it is not present in the concrete form.

- The body of a procedure which is code is translated into the appropriate state transformation.

- Constants are, similarly, translated to (abstract) values.

- The outermost block (a created one, if necessary) contains the standard functions and procedures: where these cannot be expressed in ALGOL 60, meta-language descriptions of the transformations are given.

- The body of the abstract form of a for statement is always a block; if not present in the concrete form it is generated by the translator.

- The use of one *<bound pair list>* to define several *<array identifiers>* is expanded by the translator. Notice that this can not be justified from *MAR* and, with side-effect producing function references in the bound pair list, is strictly wrong.

2. CONTEXT CONDITIONS.

An environment is used to record statically known type information:

$$Static\text{-}env = Id \underset{m}{\rightarrow} Specifier$$

With the exception of *is-wf-program*, all context conditions are, for a phrase class Θ, of type:

$$is\text{-}wf\text{-}\Theta\text{: } \Theta \quad Static\text{-}env \rightarrow Bool$$

As well as the splitting ("routing") rules, certain other obvious steps have been taken to shorten the functions given below, e.g. if

$$\Theta :: \Theta_1 \; \Theta_2 \; \ldots \; \Theta_n$$

then a rule (or part thereof) of the form:

$$\begin{aligned} & if\text{-}wf\text{-}\Theta(mk\text{-}\Theta(\Theta_1,\Theta_2,\ldots,\Theta_n),env) = \\ & \qquad is\text{-}wf\text{-}\Theta_1(\Theta_1,env) \; \& \\ & \qquad is\text{-}wf\text{-}\Theta_2(\Theta_2,env) \; \& \; \ldots \\ & \qquad is\text{-}wf\text{-}\Theta_n(\Theta_n,env) \end{aligned}$$

will be omitted.

2.1 *is-wf* Rules.

```
is-wf-program(mk-program(b)) =
  /* for all type-decl, array-decl's within b, their s-oid is unique */ &
  (let oads={d|within(d,b) &  is-array-decl(d) & s-oid(d) ≠ NIL}
  /* all expressions in s-bdl of elements of oads are integer constants */) &
  (let env = [n↦mk-type-proc(INT)|n∈Int-funct-names] ∪
             [n↦mk-type-proc(REAL)|n∈Real-funct-names] ∪
             [n↦PROC|n∈Proc-names]
   is-wf-block(b,env))
type: Program→Bool


is-wf-block(mk-block(dcls,stl),env) =
  let labl = /* list of all labels contained in stl without an intervening
                                                                block */
  is-uniquel(labl) &
  is-disjoint(<elems labl,{s-id(d)|d∈dcls}>) &
  (let renv = env\{s-id(d)|d∈dcls}
   let lenv = [s-id(d)↦(cases d:
                          mk-type-decl(,,tp)        → tp
                          mk-array-decl(,,tp,)      → mk-type-array(tp)
                          mk-switch-decl(,)         → SWITCH
                          mk-proc-decl(,PROC,,,,)→ PROC
                          mk-proc-decl(,tp,,,,)    → mk-type-proc(tp))
                               |d∈dcls] ∪
              [lab↦LABEL|lab∈elems labl]
   let nenv = renv  ∪ lenv
    d∈dcls ⇒
           ((is-array-decl(d)  ⇒ is-wf-array-decl(d,renv)) &
            (is-proc-decl(d)   ⇒ is-wf-proc-decl(d,nenv)) &
            (is-switch-decl(d) ⇒ is-wf-switch-decl(d,nenv))) &
   (1≤i≤len stl ⇒ is-wf-unlab-stmt(s-sp(stl(i)),nenv)))
```

```
is-wf-assign-stmt(mk-assign-stmt(dl,e),env) =
  1≤i≤len dl ⇒
     (let mk-destin(lp,tp)=dl(i)
      compat-tps(tp,expr-tp(e,env)) &
      (is-var(lp) ⇒ (is-scalar(lp,env) &
                     tp=var-tp(lp,env))) &
      (is-atv-proc-id(lp) ⇒ tp=s-type(env(lp))))


is-wf-goto-stmt(mk-goto-stmt(e),env) = expr-tp(e,env) = LABEL


is-wf-cond-stmt(mk-cond-stmt(dec,th,el),env) = expr-tp(dec,env) = BOOL


is-wf-for-stmt(mk-for-stmt(cv,cvtp,flel,b),env) =
  is-simple-var(cv) & is-scalar(cv,env) &
  is-arithm(cvtp) & cvtp=var-tp(cv,env)


is-wf-expr-elem(mk-expr-elem(e),env) = is-arithm(expr-tp(e,env))


is-wf-while-elem(mk-while-elem(in,wh),env) =
  is-arithm(expr-tp(in,env)) & is-BOOL(expr-tp(wh,env))


is-wf-step-until-elem(mk-step-until-elem(in,st,un),env) =
  is-arithm(expr-tp(in,env)) &
  is-arithm(expr-tp(st,env)) &
  is-arithm(expr-tp(un,env))


is-wf-proc-des(mk-proc-des(id,apl),env) =
  is-PROC(env(id)) &
  (1≤i≤len apl ⇒ s-tp(apl(i))=act-parm-tp(s-v(apl(i)),env)


is-wf-var(v,env) =
  if is-simple-var(v) then is-type(env(s-nm(v)))
  else is-type-array(env(s-nm(v))) &
       (1≤i≤len s-sscl(v) ⇒ is-arithm(expr-tp(s-sscl(v)(i),env)))
```

is-wf-simple-var-bn/bv }
is-wf-subscr-var-bn/bv } former iff refers to by name formal parameter

is-wf-label-const(mk-label-const(id),env) = env(id) = LABEL

is-wf-switch-des(mk-switch-des(id,e),env) =
env(id) = SWITCH &
is-arithm(expr-tp(e,env))

is-wf-funct-des(mk-funct-des(id,apl),env) =
is-type-proc(env(id)) &
(1≤i≤len apl ⇒ s-tp(apl(i))=act-parmv-tp(s-v(apl(i)),env))

is-wf-array-name(mk-array-name(id),env) = is-type-array(env(id))

is-wf-switch-name(mk-switch-name(id),env) = env(id) = SWITCH

is-wf-proc-name(mk-proc-name(id),env) = is-type-proc(env(id)) ∨ env(id) = PROC

is-wf-prefix-expr(mk-prefix-expr(opr,expr),env) =
let tp = expr-tp(expr,env)
(cases opr:
NOT → tp = BOOL
REAL-PLUS, REAL-MINUS → tp = REAL
INT-PLUS, INT-MINUS → tp = INT)

```
is-wf-infix-expr(mk-infix-expr(e1,opr,e2),env) =
  let tp1 = expr-tp(e1,env)
  let tp2 = expr-tp(e2,env)
  (cases opr:
     REAL-ADD, REAL-SUB, REAL-MULT → (tp1=REAL ∨ tp2=REAL) &
                                     is-arith(tp1) & is-arith(tp2)
     REAL-DIV                      → is-arith(tp1) & is-arith(tp2)
     REAL-EXP                      → is-arith(tp1) & tp2=REAL
     REAL-INT-EXP                  → tp1=REAL & tp2=INT
     INT-ADD, INT-SUB, INT-MULT
     INT-DIV, INT-EXP              → tp1=INT & tp2=INT
     LT, LE, EQ, NE, GE, GT        → is-arith(tp1) & is-arith(tp2)
     IMPL, EQU, AND, OR            → tp1=BOOL & tp2=BOOL)


is-wf-cond-expr(mk-cond-exp(b,t,e),env) =
  expr-tp(b,env) = BOOL &
  (let tp1 = expr-tp(t,env)
   let tp2 = expr-tp(e,env)
    compat-tps(tp1,tp2))


is-wf-array-decl(mk-array-decl(,,tp,bdl),env) =
  1≤i≤len bdl ⇒ is-arith(expr-tp(s-lbd(bdl(i)),env) &
                is-arith(expr-tp(s-ubd(bdl(i)),env)


is-wf-switch-decl(mk-switch-decl(exl),env) =
  1≤i≤len exl ⇒ expr-tp(exl(i),env) = LABEL


is-wf-proc-decl(mk-proc-decl(id,tp,fpl,vids,spm,b),env) =
  is-uniquel(fpl) &
  id ∉ elems fpl &
  vids ⊆ elems fpl &
  dom spm = elems fpl &
  (id∈vids ⇒ spm(id)∈(Type∪{LABEL}∪Type-array)) &
   is-wf-block(b,env+spm)
```

2.2 *tp* Determining Rules.

```
expr-tp(expr,env) =
   cases expr:
     mk-bool-const()                          → BOOL
     mk-int-const()                           → INT
     mk-real-const()                          → REAL
     mk-label-const()                         → LABEL
     mk-var-ref(mk-var(var))                  → env (s-nm(var))
     mk-funct-des(id,)                        → s-type (env(id))
     mk-switch-des()                          → LABEL
     mk-prefix-expr(opr,)
        cases opr:
          INT-PLUS, INT-MINUS                 → INT
          REAL-PLUS, REAL-MINUS               → REAL
          NOT                                 → BOOL
     mk-infix-expr(,opr,)
        cases opr:
          INT-ADD, INT-SUB, INT-MULT,
          INT-DIV, INT-EXP                    → INT
          REAL-ADD, REAL-SUB, REAL-MULT,
          REAL-DIV, REAL-EXP, REAL-INT-EXP    → REAL
          LT, LE, EQ, NE, GE, GT,
          AND, OR, IMPL, EQU                  → BOOL
     mk-cond-expr(,t,e)
        let tp1 = expr-tp(t,env)
        let tp2 = expr-tp(e,env)
        tp1 = tp2                             → tp1
        is-arith(tp1) & is-arith(tp2)         → REAL


var-tp(v,envs) =
   cases v:
     mk-simple-var(id,)       → envs(id)
     mk-subscr-var(id,,sscl)  → s-type(envs(id)
```

act-parmv-tp similar to *expr-tp*

2.3 Auxiliary Functions.

compat-tps(tp1,tp2) =
  tp1 = tp2 ∨
  is-arithm(tp1) & is-arithm(tp2)

type: (Type | LABEL) (Type | LABEL) → BOOL

is-scalar(v,env) =
  is-type(env(s-nm(v))) ∨
  is-array-type(env(s-nm(v))) & is-subscr-var(v)

3. SEMANTIC OBJECTS.

Tr	=	$\Sigma \tilde{\to} \Sigma$ *Abn*
Σ	=	(*R-STG* $\underset{m}{\to}$ *Storage*) $\cup$ (*R-CHANS* $\underset{m}{\to}$ *Channels*)
Storage	=	*Sc-loc* $\underset{m}{\to}$ *Sc-val*
Sc-loc	=	*Bool-loc* | *Real-loc* | *Int-loc*
Bool-loc, Real-loc, Int-loc		are disjoint, infinite, sets
Sc-val	=	*Int* | *Real* | *Bool*
Channels	=	*Int* $\underset{m}{\to}$ *Char**
Abn	=	[*Label-den*]
Stmt-env	=	(*Own-env Env Aid-set*)
Expr-env	=	(*Env Aid-set*)
Own-env	=	*Own-id* $\underset{m}{\to}$ (*Type-den* | *Array-den*)
Env	=	*Id* $\underset{m}{\to}$ *Den*
Den	=	*Type-den* | *Atv-proc-id-den* | *Array-den* | *Proc-den* | *Label-den* *Switch-den* | *By-name-den* | *String*
Type-den	=	*Sc-loc*
Atv-proc-id-den	=	*Sc-loc*
Array-den	=	$Int^{+} \underset{m}{\leftrightarrow}$ *Sc-loc*
Constraint	:	$(\exists ipl \in (Int\ Int)^{+})$(**dom** *a loc=rect(ipl)*)
Proc-den	=	*Act-parm-den** *Aid-set* → ($\Sigma \tilde{\to} \Sigma$ *Abn* [*Sc-val*])
Act-parm-den	::	*s-v:Den s-tp:Specifier*
Label-den	::	*Id Aid*
Switch-den	=	*Int Aid-set* → ($\Sigma \tilde{\to} \Sigma$ *Abn Label-den*)
Aid		infinite set

By-name-den = *By-name-loc-den* | *By-name-expr-den*
By-name-loc-den :: *Aid-set* → ($\Sigma \tilde{\to} \Sigma$ *Loc*)
By-name-expr-den :: *Aid-set* → ($\Sigma \tilde{\to} \Sigma$ *Sc-val*)
Loc = *Type-den* | *Array-den*
Abn = [*Label-den*]
Val = *Sc-val* | *Label-den*

Implementation-defined-const = MAXINT | MINREAL | MAXREAL | EPSILON

Comment: The constants are *Sc-vals*, but no check is made as to whether they are machine representable.

4. MEANING FUNCTIONS

4.1 Functions from Object Language

```
i-program(p)(chans) =
    let state_s = [R-STG↦[], R-CHANS↦chans]
    let state_f = i-program1(p)(state_s)
    state_f(R-CHANS)

type:  Program → (Channels → Channels)


i-program1(mk-program(b)) =
    let own-decls = {d|within(d,b) & (is-type-decl(d)vis-array-decl(d)) &
                          s-oid(d) ≠ NIL}
    let oenv: ([s-oid(d)↦e-own-type-decl(d)|d∈own-decls & is-type-decl(d)] ∪
               [s-oid(d)↦e-own-array-decl(d)|d∈own-decls & is-array-decl(d)]);
    (tixe[RET↦I]
     in i-block(b,<oenv,env,{}>));
     epilogue({s-id(d)|d∈own-decls},oenv)

type:  Program ⇒
```

```
e-own-type-decl(d) =
    let l:e-type-decl(d);
    if s-desc(d)=BOOL then assign(FALSE,l)
    else                   assign(0,l);
    return (l)

type:  Type-decl ⇒ Type-den


e-own-array-decl(d) =
    let l:e-array-decl(d,<[],{}>);
    if s-tp(d)=BOOL then
            for all scl∈rngl do assign(FALSE,scl)
    else
            for all scl∈rngl do assign(0,scl);
    return (l)

type:  Array-decl ⇒ Array-den
```

Standard Functions and Transput

It is assumed that the translation of the standard functions and procedures are contained in the ("fictitious") outer block. The interpretation of their *proc-decl* follows the normal interpretation rules (*e-proc-decl*) except in the cases where the body cannot be expressed in Algol. In these cases the state transition of the non-Algol part is explicitly listed below.

Note: Referencing the translated identifiers we use quotes (e.g. "*inreal*" for the translation of the identifier inreal).

In procedure *stop*:

"*goto* Ω" → *exit*(*RET*)

In procedure *inchar:* <body> →

```
let channel : contents(env("channel"));
let str     = env("str")
let int     : e-left-part(mk-simple-var-bn("int"),exenv)
if channel∉dom c R-CHANS
   then error
   else (let chan: (c R-CHANS)(channel);
         if chan = <> then error;
         let char: hdchan
         let ind = if (∃i∈{1:lenstr})(str(i)=char)
                     then (ιi∈{1:lenstr})(str(i)=char &
                              (∀k∈{1:i-1})(str(k)≠char))
                     else 0;
         R-CHANS := c R-CHANS +
                     [channel↦tlchan];
         assign(ind,int))
```

In procedure *outchar:* <statement> →

```
let channel : contents(env("channel"));
let str     = env("str")
let int     : contents(env("int"));
let char    = str(int)
if channel∉dom c R-CHANS then error;
R-CHANS    := c R-CHANS +
              [channel↦(c R-CHANS)(channel)^<char>]
```

In procedure *outterminator:* <body> →

```
let channel : contents(env("channel"));
if channel∉dom c R-CHANS then error;
R-CHANS     := (c R-CHANS) +
               [channel↦(c R-CHANS)(channel)^<implementation
                          defined symbol depending on the current
                          state of the channel>]
```

Procedures "*maxint*", "*minreal*", "*maxreal*" and "*epsilon*" have bodies which return the appropriate *Implementation-defined-const*.

```
i-block(mk-block(dcls,stl),<oenv,env,cas>) =
   let aid∈Aid be s.t. aid∉cas
   let nenv : env +
           ([s-id(d)↦oenv(s-oid(d))|d∈dcls & (is-type-decl(d) v
                              is-array-decl(d)) & s-oid(d) ≠ NIL] U
            [s-id(d)↦e-type-decl(d)|d∈dcls & is-type-decl(d) &
                              s-oid(d) = NIL] U
            [s-id(d)↦e-array-decl(d,<env,cas>)|d∈dcls & is-array-decl(d) &
                              s-oid(d) = NIL] U
            [s-id(d)↦e-switch-decl(d,nenv)|d∈dcls & is-switch-decl(d)] U
            [s-id(d)↦e-proc-decl(d,oenv,nenv)|d∈dcls & is-proc-decl(d)] U
            [lab↦mk-label-den(lab,aid)|is-contnd(lab,stl)]);
   let stenv = <oenv,nenv,casU{aid}>
   always epilogue({s-id(d)|d∈dcls & (is-type-decl(d) v is-array-decl(d)) &
                              s-oid(d) = NIL},nenv)
   in (tixe[mk-label-den(tlab,aid)→
                    cue-i-stmt-list(tlab,stl,st-env) |
                              is-contndl(tlab,stl)]
      in for i=1 to lenstl do i-unlab-stmt(s-sp(stl(i)),stenv)
     )

type:  Block Stmt-env ⇒


epilogue(ids,env) =
   let sclocs = {env(id)|id∈ids & is-sc-loc(env(id))} U
                union{rng(env(id))|id∈ids & is-array-den(env(id))}
   R-STG      := R-STG\sclocs

type:  Id-set Env ⇒


cue-i-stmt-list(lab,stl,stenv) =
   let i = index(lab,stl)
   cue-i-stmt(lab,stl(i),stenv);
   for j = i+1 to lenstl do i-unlab-stmt(s-sp(stl(i)),stenv)

type:  Id Stmt* Stmt-env ⇒
pre:   is-contndl(lab,stl)
```

```
cue-i-stmt(lab,mk-stmt(labs,sp),stenv) =
   if lab∈labs then i-unlab-stmt(sp,stenv)
   else cue-i-unlab-stmt(lab,sp,stenv)

type:  Id Stmt Stmt-env ⇒
pre:   is-contnd(lab,mk-stmt(labs,sp))


cue-i-unlab-stmt: Id Unlab-stmt Stmt-env ⇒


cue-i-cond-stmt(lab,mk-cond-stmt(,th,el),stenv) =
   if is-contnd(lab,th) then cue-i-stmt(lab,th,stenv)
   else                      cue-i-stmt(lab,el,stenv)

pre:   is-contnd(lab,th) ∨ is-contnd(lab,el)


cue-i-comp-stmt(lab,mk-comp-stmt(stl),stenv) =
   cue-i-stmt-list(lab,stl,stenv)


i-unlab-stmt: Unlab-stmt Stmt-env ⇒

i-comp-stmt(mk-comp-stmt(stl),stenv) =
   for i=1 to lenstl do i-unlab-stmt(s-sp(stl(i)),stenv)


i-assign-stmt(mk-assign-stmt(dl,e),<,env,cas>) =
   let dl:<e-left-part(s-tg(dl(i)),<env,cas>)|1≤i≤lendl>;
   let v:  e-expr(e,<env,cas>);
   for i=1 to lendl do
       (let vc:conv(v,s-tp(dl(i)));
        assign(vc,dl(i)))


e-left-part: Left-part Expr-env ⇒ Sc-loc


e-atv-proc-id(mk-atv-proc-id(id),<env,>) = env(id)
```

```
i-goto-stmt(mk-goto-stmt(e),<,env,cas>) =
   let ld:e-expr(e,<env,cas>);
   exit(ld)


i-dummy-stmt(t,stenv) = I


i-cond-stmt(mk-cond-stmt(dec,th,el),stenv) =
   let <,env,cas> = st-env
   let b:e-expr(dec,<env,cas>);
   if b then i-unlab-stmt(s-sp(th),stenv)
   else      i-unlab-stmt(s-sp(el),stenv)


i-for-stmt(mk-for-stmt(cv,cvtp,flel,b),stenv) =
   for i=1 to lenflel do i-for-list-elem(flel(i),cv,cvtp,b,stenv)


i-for-list-elem: For-list-elem Var Type Block Stmt-env ⇒


i-expr-elem(mk-expr-elem(e),cv,cvtp,b,stenv) =
   let <,env,cas> = stenv
   let v:e-expr(e,<env,cas>);
   let vc:conv(v,cvtp);
   let l:e-var(cv,<env,cas>);
   assign(vc,l);
   i-block(b,stenv)


i-while-elem(mk-while-elem(in,wh),cv,cvtp,b,stenv) =
   let <,env,cas> = stenv
   while (let v:e-expr(in,<env,cas>);
          let vc:conc(v,cvtp);
          let l:e-var(cv,<env,cas>);
          assign(vc,l);
          let b:e-expr(wh,<env,cas>);
          b    ) do    i-block(b,stenv)
```

```
i-step-until-elem(mk-step-until-elem(in,st,un),cv,cvtp,b,stenv) =
   let <,env,cas> = stenv
   let exenv = <env,cas>
   let vin:e-expr(in,exenv);
   let vinc:conv(vin,cvtp);
   let l:e-var(cv,exenv);
   assign(vinc,l);
   while (let vst:e-expr(st,exenv);
          let b:e-expr(untest,exenv);
          b ) do (i-block(b,stenv);
                 let vcur:contents(l)+vst;
                 let vcurc:conv(vcur,cvtp);
                 assign(vcurc,l))
```

note: "*untest*" is an *Expr* corresponding to ⌜*(cv-un)×signvst≤0*⌝

```
i-proc-stmt(mk-proc-stmt(des),<,env,cas>) =
   cases des:
     mk-proc-des(id,apl) →
        (let denl = <e-act-parm(apl(i),env)|1≤i≤lenapl>
         let f = env(id)
         f(denl,cas))
     T → (let v:e-funct-des(des,<env,cas>);
          I)
```

```
e-expr: Expr Expr-env ⇒ Val
```

```
e-bool-const(mk-bool-const(b),) = return(b)

e-real-const(mk-real-const(r),) = represent(r)

e-int-const(mk-int-const(i),)   = test(i)
```

```
e-var-ref(mk-var-ref(v),<env,cas>) =
   if is-simple-var-bv(v) ∨ is-subscr-var-bv(v) ∨ is-by-name-loc-den(env(s-nm(v)))
   then (let l:e-var(v,<env,cas>);
         contents(l))
   else (let bned = env(s-nm(v))
         bned(cas))


e-var: Var Expr-env ⇒ Sc-loc


e-simple-var-bn(mk-simple-var-bn(id),<env,cas>) =
   let bnd = env(id)
   if is-by-name-loc-den(bnd) then bnd(cas)
   else error


e-simple-var-bv(mk-simple-var-bv(id),<env,>) = env(id)


e-subscr-var-bn(mk-subscr-var-bn(id,sscl),<env,cas>) =
   let esscl:e-subscrl(sscl,<env,cas>);
   let bnd = env(id)
   if is-by-name-loc-den(bnd) then
             (let aloc:bnd(cas);
              if esscl∈domaloc then return (aloc(esscl))
              else error)
   else error


e-subscr-var-bv(mk-subscr-var-bv(id,sscl),<env,cas>) =
   let esscl:e-subscrl(sscl,<env,cas>);
   let aloc = env(id)
   if esscl∈domaloc then return (aloc(esscl))
   else error
```

```
e-subscrl(sscl,exenv) =
   let esscl:<(let essc:e-expr(sscl(i),exenv);
               let i:conv(essc,INT);
               i                          )|1≤i≤lensscl>;
   return (esscl)

type:  Expr* Expr-env ⇒ Int*


e-label-const(mk-label-const(id),<env,>) = return(env(id,


e-switch-des(mk-switch-des(id,ssc),<env,cas>) =
   let ess:e-expr(ssc,<env,cas>);
   let i:conv(ess,INT);
   let f = env(id)
   let ld:f(i,cas);
   return (ld)


e-funct-des(mk-funct-des(id,apl),<env,cas>) =
   let denl = <e-act-parm(apl(i),env)|1≤i≤lenapl>
   let f    = env(id)
   let v:f(denl,cas);
   return (v)


e-act-parm(mk-act-parm(e,tp),env) =
   let d = e-act-parmv(e,env)
   mk-act-parm-den(d,tp)

type: Act-parm Env → Act-parm-den


e-act-parmv: Act-parmv Env → Den
```

```
e-parm-expr(mk-parm-expr(e),env) =
   if is-var-ref(e) then
      (let f(dcas) = e-var-ref(e,<env,dcas>)
       mk-by-name-loc-den(f)
      )
   else
      (let f(dcas) = e-expr(e,<env,dcas>)
       mk-by-name-expr-den(f)
      )


e-array-name(mk-array-name(id),env) = env(id)


e-switch-name(mk-switch-name(id),env) = env(id)


e-proc-name(mk-proc-name(id),env) = env(id)


e-string(s,env) = s


e-prefix-expr(mk-prefix-expr(opr,e),exenv) =
   let v:e-expr(e,exenv);
   apply-prefix-opr(opr,v)


e-infix-expr(mk-infix-expr(e1,opr,e2),exenv) =
   let v1:e-expr(e1,exenv);
   let v2:e-expr(e2,exenv);
   apply-infix-opr(opr,v1,v2)


e-cond-expr(mk-cond-expr(b,t,e),exenv) =
   if e-expr(b,exenv)
      then e-expr(t,exenv)
      else e-expr(e,exenv)
```

Comment: The evaluation of infix expressions is from left to right. Since *Int*⊆*Real* no explicit conversion from integer to real is necessary in infix and conditional expressions.

```
represent(r) =
   if -MAXREAL<r<-MINREAL v
       MINREAL<r< MAXREAL v
              r=0
     then return (an implementation defined
                      approximation of r)
     else error

type:  Real ⇒ Real


test(i) =
   if -MAXINT<r<MAXINT
     then return(i)
     else error

type:  Int ⇒ Int


apply-prefix-opr(opr,v) =
   cases opr:
     NOT                    → return(¬v)
     REAL-PLUS, INT-PLUS    → return(v)
     REAL-MINUS, INT-MINUS  → return(-v)

type:  Prefix-opr Sc-val ⇒ Sc-val
```

```
apply-infix-opr(opr,v,w) =
  cases opr:
    REAL-ADD      → represent(v+w)
    REAL-SUB      → represent(v-w)
    REAL-MULT     → represent(v*w)
    REAL-DIV      → if w=0
                       then fault1
                       else represent(v*represent(1/w))
    REAL-EXP      → if v>0
                       then value of the standard function exp
                            applied on represent(v*value of the
                                                  standard function
                                                  ln applied on w)
                       else fault2
    REAL-INT-EXP  → if v=0 & w=0
                       then fault3
                       else (let expn(n) = if n=0
                                              then 1
                                              else represent(expn(n-1)*n.
                             if w≥0
                                then expn(w)
                                else represent(1/expn(-w)))
    INT-ADD       → test(v+w)
    INT-SUB       → test(v-w)
    INT-MULT      → test(v*w)
    INT-DIV       → if w=0
                       then fault1
                       else test(v/w)
    INT-EXP       → if w<0 & v=0  ∨ w=0
                       then fault4
                       else (let expi(n) = if n=0
                                              then 1
                                              else test(expi(n-1)*v)
                             return(expi(w))
    LT            → return(v<w)
    LE            → return(v≤w)
    EQ            → return(v=w)
    NE            → return(v≠w)
    GE            → return(v≥w)
    GT            → return(v>w)
    IMPL          → return(v⇒w)
```

```
    EQU            → return(v⇔w)
    AND            → return(v&w)
    OR             → return(v∨w)

  type:  Infix-opr Sc-val Sc-val ⇒ Sc-val
```

Comment: *fault1* represents the state transition, which corresponds to the call: *fault('div by zero',v)*
fault2 ~ *fault('expr undefined',v)*
fault3 ~ *fault('expn undefined',v)*
fault4 ~ *fault('expi undefined',w)*

```
e-type-decl(mk-type-decl(,,tp)) = gen-sc-den(tp)

type:  Type-decl ⇒ Sc-loc


e-array-decl(mk-array-decl(,,tp,bdl),exenv) =
   let ebds:<(let v:e-expr(s-lbd(bdl(i)),exenv);
              let lbd:conv(v,INT);
              let w:e-expr(s-ubd(bdl(i)),exenv);
              let ubd:conv(w,INT);
              if v>w then error;
                 <lbd,ubd>)|1≤i≤lenbdl>;
   let indes = rect(ebds)
   let array-den:gen-array-den(indes,tp);
   return(array-den)

type: Array-decl Expr-Env ⇒ Array-den
```

Comment: It is assumed that the generation of an array without elements is erroueous, rather than the access to such an array.

```
e-switch-decl(mk-switch-decl(,exl),env) =
   let f(ind,cas) =
         (if 1≤ind≤lenexl then e-expr(exl(i),<env,cas>)
          else error
         )
   f

type:  Switch-decl Env → Switch-den


e-proc-decl(mk-proc-decl(id,tp,fpl,vids,spm,b),oenv,env) =
   let f(denl,cas) =
        (if lendenl ≠ lenfpl v
            (∃i∈{1:lenfpl})(¬is-parm-ok(denl(i),spm(fpl(i)),fpl(i)∈vids))
         then error
         else (let nenv:env+
                         ([fpl(i)↦s-v(denl(i))|1≤i≤lenfpl & fpl(i)∉vids))] ∪
                          [fpl(i)↦e-val-parm(s-v(denl(i)),spm(fpl(i)),cas) |
                                        1≤i≤lenfpl & fpl(i)∈vids] ∪
                          (if tp=NIL then []
                           else [id↦gen-sc-den(tp)]));
               cases b:
                 mk-code(tr) → tr
                 T           → i-block(b,<oenv,nenv,cas>);
                 epilogue({fpl(i)|1≤i≤lenfpl & fpl(i)∈vids &
                                           spm(fpl(i))∈Type∪Type-array})
                 let rv: if tp=NIL then NIL else contents(env(id));
                 if tp≠NIL then epilogue(id,nenv);
                 return(rv)))
   f

type:  Proc-decl Own-env Env → Proc-den
```

```
is-parm-ok(<v,spa>,spf,bv) =
   if bv then
         ( spa=spf ∨ is-arithm(spf) & is-arithm(spa) ∨
           is-type-array(spf) & is-type-array(spa) & is-arithm(s-type(spa))
                                                   & is-arithm(s-type(spf)))
   else
         ( spa=spf ∨ spf=PROC & is-type-proc(spa)  ∨
           is-type-proc(spf) & is-type-proc(spa) & is-arithm(s-type(spa))
                                                 & is-arithm(s-type(spf)))

type:  Act-parm-den Specifier Bool → Bool


e-val-parm(den,sp,cas) =
   cases sp:
     mk-type-array(tp) →
         (let aloc:gen-array-den(domden,tp);
          for esscl∈domden do
              (let v:contents(den(esscl));
               assign(v,aloc(esscl)));
          return(aloc))
   LABEL → den(cas)
   T     →
         (let v:(if is-by-name-expr-den(den) then den(cas)
                 else (let l:den(cas);
                       contents(l)));
          let vc:conv(v,sp);
          let l:gen-sc-den(sp);
          assign(vc,l);
          return(l))

type:  Den Specifier Aid-set ⇒ Den
```

4.2 Auxiliary Functions

```
is-contnd:  Id Stmt  → Bool
is-contndl: Id Stmt* → Bool
```

Comment: Two obvious functions for checking whether the given identifier is contained in the label part of a (contained) statement which is not contained in an intervening block.

*index: Id Stmt** $\tilde{\rightarrow}$ *Nat*

<u>Comment</u>: For identifiers which satisfy "*is-contndl*" this function finds the index such that the indexed element of the statement list also contains the identifier.

within(so,o) =
/ checks if so is a sub-part of o */*

type: Object Object → Bool

is-uniquel: Object → Bool*

<u>Comment</u>: True iff no duplicates

is-disjoint: (Object-set) → Bool*

Comment: True iff sets are pairwise disjoint

rect(ipl) =
$\{il \mid \underline{len}\,il = \underline{len}\,ipl \;\&\; (1 \leq i \leq \underline{len}\,ipl \Rightarrow ipl(i)(1) \leq il(i) \leq ipl(i)(2))\}$

type: $(Int^2)^+ \rightarrow (Int^+)\text{-}set$
pre: $1 \leq i \leq \underline{len}\,ipl \Rightarrow ipl(i)(1) \leq ipl(i)(2)$

assign(v,l) =
$\underline{R\text{-}STG} := \underline{c}\ \underline{R\text{-}STG} + [l \rightarrow v]$

type: Sc-val Sc-loc ⇒

contents(l) =
$\underline{if}\ \underline{c}\ \underline{R\text{-}STG}(l) = \underline{?}\ \underline{then}\ \underline{error}$
$\underline{else}\ \underline{c}\ \underline{R\text{-}STG}(l)$

type: Sc-loc ⇒ Sc-val

```
conv(v,tp) =
   if tp=INT
      then test(rounded value of v)
      else return(v)

pre:  is-bool(v) ⇒ tp=BOOL
      is-real(v) ⇒ is-arith(tp)


gen-array-den(indls,tp) =
   let den:[indl→gen-sc-den(tp)|indl∈indls]
   return(den)

type:  Int+-set Type ⇒ Array-den


gen-sc-den(tp) =
   let loc∈(cases tp:
              BOOL → Bool-loc
              REAL → Real-loc
              INT  → Int-loc) be s.t. loc∉dom c R-STG
   R-STG := c R-STG ∪ [loc↦?];
   return(loc)

type:  Type ⇒ Sc-loc
```