TECHNICAL REPORT

TR 25.139
20 December 1974

A FORMAL DEFINITION OF A PL/I SUBSET

PART II

H. BEKIĆ
D. BJØRNER
W. HENHAPL
C. B. JONES
P. LUCAS

IBM LABORATORY VIENNA

IBM LABORATORY VIENNA, Austria

A FORMAL DEFINITION OF A PL/I SUBSET

PART II

by

H. Bekič
D. Bjørner
W. Henhapl
C. B. Jones
P. Lucas

ABSTRACT

This report provides a formal definition of large portions of the ECMA/ANSI proposed Standard PL/I language. The metalanguage used is described in the style of the "Mathematical Semantics". That is, the definition of PL/I is given by generating a function from a source program. A commentary is also provided to cover the less clear parts of the chosen model. For the convenience of the reader who wishes to have the commentary side by side with the formulae, the report is divided into two parts: Part I contains the description of the notation, the commentary and a cross-reference; Part II contains all the formulae.

NOTE

This document is not an official PL/I language specification. The language defined is based on the working documents (BASIS/1-9 to BASIS/1-11 [1]) of the joint ECMA/ANSI working group. It has not, however, been offered to them for review and has in no way been approved. Furthermore the subset chosen is not an indication of any IBM product plan.

TR 25.139

20 December 1974

A F O R M A L D E F I N I T I O N O F A P L / I S U B S E T

C O N T E N T S

Abstract Programs

Contents:

1.1 Rules

1.1.1 Programs

```
1     prog              =  proc-set
2     proc              :: id s-parms:id* s-ret-descr:[pdd] s-dcls:dcl-set proc-set
                            s-cprefs:cond-pref-set s-recity:[REC] ex-unit*
```

1.1.2 Data

1.1.2.1 Declarations

```
3     dcl               :: s-id:id s-dcl-tp:(prop-var|parm|based|def|BI|nmd-const)
4     prop-var          :: s-dd:dd s-stg-cl:(AUTO | static-cl)
5     static-cl         :: scope
6     parm              :: s-dd:dd
7     based             :: s-dd:dd s-dft-qual:[val-ref]
8     def               :: s-dd:dd s-base-item:var-ref s-pos:expr
9     nmd-const         =  file-const | ext-entry | LAB
10    ext-entry         :: entry
11    scope             =  INT | EXT
```

1.1.2.2 Descriptions

```
12    dd                =  array-dd | struct-dd | sc-dd
13    array-dd          :: s-unit-dd:(sc-dd | struct-dd) s-bpl:(bp | *)+
14    struct-dd         :: (s-f-nm:id s-f-dd:dd)+
15    bp                :: s-lb:extent s-ub:extent
16    extent            :: expr s-refer-opt:[id+]
17    sc-dd             :: dtp s-init:[init-elem+]
18    init-elem         =  simple-init | * | iterated-init
19    simple-init       :: expr
20    iterated-init     :: s-iter-fact:expr init-elem+
```

1.1.2.3 Types

```
21  dtp              =  comp-tp | non-comp-tp
22  comp-tp          =  arith | str
23  non-comp-tp      =  entry | LAB | PTR | FILE
24  arith            :: base scale prec
25  base             =  BIN | DEC
26  scale            =  FIX | FLT
27  prec             :: s-nod:intg s-scale-fact:[signed-intg]
28  str              :: str-tp s-maxl:(extent | *) varity
29  str-tp           =  CHAR | BIT
30  varity           =  VARYING | NONVARYING
31  entry            :: s-parm-descrs:[pdd*] s-ret-descr:[pdd]
                        s-opts:[/*impl.def.*/]
32  pdd              =  array-pdd | struct-pdd | sc-pdd
33  array-pdd        :: s-unit-dd:unit-pdd s-bpl:(bp|*)+
34  unit-pdd         =  sc-pdd  |  struct-pdd
35  struct-pdd       :: pdd+
36  sc-pdd           :: dtp
```

1.1.3 Statements

```
37  ex-unit          :: s-cprefs:cond-pref-set s-st-nms:id-set prop-st
38  prop-st          =  bl | iter-grp | non-iter-grp | on-st | if-st |
                        call-st | goto-st | NULL | ret-st | rev-st | sig-st |
                        io-st | ass-st | alloc-st | free-st
39  bl               :: s-dcls:dcl-set proc-set ex-unit*
40  iter-grp         =  ctld-grp | wh-only-grp
41  ctld-grp         :: targ-ref
                        (expr [s-by-opt:expr s-to-opt:[expr]] s-wh-opt:[expr])+
                        ex-unit*
42  wh-only-grp      :: s-wh-opt:expr ex-unit*
43  non-iter-grp     :: ex-unit*
44  on-st            :: [SNAP] cond-nm+ (proc | SYSTEM)
45  if-st            :: expr s-then-unit:ex-unit s-else-unit:[ex-unit]
46  call-st          :: proc-ref
47  proc-ref         :: val-ref arg*
48  goto-st          :: val-ref
49  ret-st           :: [expr]
50  rev-st           :: cond-nm+
51  sig-st           :: cond-nm
52  ass-st           :: targ-ref+ expr
53  alloc-st         :: (id s-set-opt:[var-ref])+
54  free-st          :: (s-locr-qual:[val-ref] id)+
```

1.1.4 Names

```
55  cond-nm            =  non-io-cond-nm | nmd-io-cond
56  non-io-cond-nm     =  comp-cond-nm | ERROR | FINISH | STG
57  cond-pref          :: comp-cond-nm (ENABLED | DISABLED)
58  comp-cond-nm       =  CONV | FOFL | OFL | SIZE | STRG | STRZ | SUBRG | UFL |
                          ZDIV
```

1.1.5 Expressions

```
59  expr               =  inf-expr | pref-expr | val-ref | const
60  inf-expr           :: expr inf-op expr
61  inf-op             =  OR | AND | GT | GE | EQ | LE | LT |
                          NE | CAT | ADD | SUBT | MULT | DIV
62  pref-expr          :: pref-op expr
63  pref-op            =  NOT | PLUS | MINUS
64  val-ref            =  var-ref | proc-fct-ref | bif-ref | nmd-const-ref
65  var-ref            :: s-locr-qual:[val-ref] s-main-id:id s-qual-ids:id* subscr*
66  subscr             =  expr | *
67  proc-fct-ref       :: val-ref arg*
68  arg                =  expr | by-ref-var
69  by-ref-var         :: var-ref
70  bif-ref            =  cond-bif-nm | distr-bif-ref | non-distr-bif-ref
71  cond-bif-nm        :: ONCHAR | ONCODE | ONCOUNT | ONFILE | ONSOURCE | ONKEY
72  distr-bif-ref      :: distr-bif-nm arg+
73  non-distr-bif-ref  :: ncn-distr-bif-nm arg*
74  distr-bif-nm       =  ABS | BIN | BIT | BOOL | CEIL | CHAR | DEC | FIX | FLT |
                          FLOOR | INDEX | LENGTH | MAX | MIN | MOD | PREC |
                          SIGN | SUBSTR | TRANSLATE | VERIFY
75  non-distr-bif-nm   =  ADDR | COLLATE | DATE | DIM | HBOUND | LBOUND |
                          NULL | STR | TIME
76  nmd-const-ref      =  lab-ref | entry-ref | file-ref
77  lab-ref            :: id
78  entry-ref          :: id
79  targ-ref           =  var-ref | pv-ref
80  stg-pv-ref         :: stg-pv-nm arg*
81  stg-pv-nm          =  STR | SUBSTR
82  pv-ref             =  cond-pv | stg-pv-ref
83  cond-pv            =  ONCHAR | ONSOURCE
84  const              :: scomp-tp symbol+
```

1.1.6 Input/Output

```
85    io-st          =  file-ctl-st | record-st | stream-st
86    file-ctl-st    =  open-st | close-st
87    record-st      =  read-st | write-st | rewrite-st | delete-st | locate-st
88    stream-st      =  get-st | put-st

89    open-st        :: opng+
90    opng           :: val-ref s-title:[expr] file-descr layout-inf
91    layout-inf     :: s-lsz:[expr] s-psz:[expr] s-tabs:[expr+]
92    close-st       :: clng+
93    clng           :: val-ref [environment]
94    file-descr     :: s-org:[STR|REC] s-mode:[IN|OUT|UPD] s-order:[SEQ|DIR]
                        s-pr:[PR] s-keyed:[KEYED] [environment]

95    read-st        :: val-ref s-data-part:(into-inf|ptr-set-inf|ignore-inf)
                                s-key-part:[key-inf|keyto-inf]
96    write-st       :: val-ref s-from:var-ref [key-inf]
97    rewrite-st     :: val-ref s-from:[var-ref] [key-inf]
98    delete-st      :: val-ref [key-inf]
99    locate-st      :: val-ref s-based:id [ptr-set-inf] [key-inf]

100   get-st         :: val-ref s-skip:[expr] s-data-list:targ-ref+
101   put-st         :: val-ref s-skip:[expr] s-page:[PAGE] s-data-list:expr*

102   into-inf       :: var-ref
103   ptr-set-inf    :: var-ref
104   ignore-inf     :: expr
105   key-inf        :: expr
106   keyto-inf      :: targ-ref

107   environment    =                          /* implementation defined */

108   nmd-io-cond    :: io-cond val-ref
109   io-cond        =  ENDFILE | ENDPAGE | KEY | RECORD | TRANSMIT | UNDEFINEDFILE
110   file-ref       :: id
111   file-const     :: file-descr scope
```

1.1.7 Elementary Domains

```
intg
signed-intg
id
symbol
```

1.2 Context Conditions and Functions

1.2.1 Static Data Descriptions

```
112  sdd              =  array-sdd  |  struct-sdd  |  sc-sdd
113  array-sdd        :: s-unit-dd:(sc-sdd | struct-sdd) s-bpl:*+
114  struct-sdd       :: sdd+
115  sc-sdd           :: sdtp
116  sdtp             =  scomp-tp | snon-comp-tp
117  scomp-tp         =  arith  |  str-sdd
118  snon-comp-tp     =  sentry  |  LAB  |  PTR  |  FILE
119  sentry           :: s-parm-descrs:[sdd*] s-ret-descr:[sdd]
120  str-sdd          :: str-tp s-maxl:*


121  ENV-t            =  ID -> (dcl|proc)
```

1.2.2 Rule-by-Rule Conditions

```
1  is-wf-prog(procs) =
     p1,p2∈procs ∧ s-id(p1)=s-id(p2) ⊃ p1=p2 ∧
     (let main-pr = (Up∈procs)(s-id(p)=main-id);
        /*main-id given s.t. (∃p∈procs)(s-id(p)=main-id)*/
      s-parms(main-pr) = <> ∧
      s-ret-descr(main-pr) = nil) ∧
     /* dcls of ENTRY EXT "match" p∈procs */ ∧
     (let ext-dcls = {dcl∈comp-dcls(procs) | is-prop-var(s-dcl-tp(dcl)) ∧
                                             s-stg-cl(s-dcl-tp(dcl)) = EXT};
      d1,d2∈ext-dcls ∧ e1∈comp-exprs(d1) ∧ e2∈comp-exprs(d2) ⊃
              eval-restr-expr(e1) = eval-restr-expr(e2))  ∧
     (let env = [s-id(p) → p | p∈procs];
      p∈procs ⊃ is-wf-proc(p,env))
```

```
2  is-wf-proc(<nm,parm-l,rdd,dcls,procs,cprefs,recity,eu-l>,env) =
      is-unique-ids(parm-l)                                    ∧
      id∈Rparm-l ⊃ (∃dcl∈dcls)(dcl=mk-dcl(id,mk-parm()) ∧
      is-unique-cprefs(cprefs)                                 ∧
      is-wf-bl(mk-bl(dcls,procs,eu-l),env)                     ∧
      (let rets = ret-sts(eu-l);
       rdd=nil  ∧ rs∈rets ⊃ rs=nil                             ∧
       rdd≠nil ∧ rs∈rets ⊃ rs≠nil ∧ is-prom-conv(rdd,el-sdd(s-expr(rs),env)))   ∧
      /* last stmt (executed) must be RETURN */

3  is-wf-dcl(<id,dcl-tp>,env,r-env) =
      /* any contained iterated-init in dcl-tp is contained in an array-dd */   ∧
      (cases dcl-tp:
       mk-prop-var( ,AUTO) -> is-wf-prop-var(dcl-tp,r-env)
       mk-prop-var( ,)     -> is-wf-prop-var(dcl-tp,[ ])
       mk-parm( )          -> is-wf-parm(dcl-tp,r-env)
       mk-based( )         -> is-wf-based(dcl-tp,env)
       mk-def( )           -> is-wf-def(dcl-tp,r-env)
       BI                  -> id∈(distr-bif-nm ∪ non-distr-bif-nm))

4  is-wf-prop-var(<dd,stg-cl>,env) =
      is-starless(dd)      ∧
      conts-no-refers(dd)                    ∧
      is-static-cl(stg-cl) ⊃ conts-restr-exprs(dd)    ∧
      is-static-cl(stg-cl) ∧ sc-sdd∈comp-sc-dds(dd) ∧ s-dtp(sc-dd) ∈ {LAB} ∪ entry ⊃
               s-init(sc-dd)=nil

6  is-wf-parm(<dd>,env) =
      conts-no-refers(dd)  ∧
      conts-no-inits(dd)   ∧
      conts-restr-exprs(dd)

7  is-wf-based(<dd,dft-qual>,env) =
      is-starless(dd)      ∧
      dft-qual=nil ∨ el-sdd(dft-qual,env)=mk-sc-sdd(PTR)      ∧
      is-refer-geom(dd)    ∧
      mk-extent(e,nil) ∈comp-extents(dd) ⊃ is-const(e)
```

BASIS-11: does not constrain non-refer bounds.

```
8  is-wf-def(<dd,base,pos>,env) =
       is-starless(dd)    ^
       conts-no-refers(dd)  ^
       conts-no-inits(dd)                    ^
       is-scomp-tp(s-sdtp(el-sdd(pos,env)))  ^
       /* relation twixt base desc. and dd */   ^
       (let <lq,id,id-l,ssc-l> = base;
        s-dcl-tp(env(id)) ∈ (prop-varuparm)  ^
        (∃tp)(is-all-str(tp,s-dd(s-dcl-tp(env(id))))))


14 is-wf-struct-dd(<f-l>,env) =
       is-unique-ids(<s-f-nm(f-l[i]) | 1≤i≤lf-l>)


16 is-wf-extent(<ex,ref-opt>,env) =
       is-scomp-tp(s-sdtp(el-sdd(ex,env)))   ^
       is-id-list(ref-opt) ⊃
          (let ref-obj-sdd = el-sdd(mk-var-ref(nil,hidl,tidl,<>),env);
           ref-obj-sdd = intg-tp())

   BASIS-11: only restricts to is-scomp-tp(ref-obj-sdd)


17 is-wf-sc-dd(<dtp,init>,env) =
       e∈comp-simple-inits(init) ⊃ /* el-sdd(e,env) "matches" dtp */


19 is-wf-simple-init(<e>,env) =
       is-sc-sdd(el-sdd(e,env))


20 is-wf-iterated-init(<ifct, >,env) =
       is-scomp-tp(s-sdtp(el-sdd(ifct,env)))


31 is-wf-entry(<pdd-l,rdd, >,env) =
       dtp∈comp-entries(pdd-l) ⊃ dtp=entry0()


32 is-wf-pdd(pdd,env) =
       conts-restr-exprs(pdd)  ^
       conts-no-refers(pdd)
```

```
37 is-wf-ex-unit(<cprefs, , >,env) =
      is-unique-cprefs(cprefs)


39 is-wf-bl(<dcls,procs,eu-l>,env) =
      dp1,dp2∈(dclsuprocs) ∧ s-id(dp1)=s-id(dp2) ⊃ dp1=dp2 ∧
      (let labl = labels(eu-l);
       is-unique-ids(labl) ∧
       id∈Rlabl ⊃ (∃dcl∈dcls)(dcl=mk-dcl(id,LAB))) ∧
      (let r-env' = env\{s-id(pd)|pd∈(dclsuprocs)};
       let env'   = r-env' u [s-id(pd) → pd | pd∈(dclsuprocs)];
       dcl∈dcls ⊃ is-wf-dcl(dcl,env',r-env') ∧
       proc∈procs ⊃ is-wf-proc(proc,env')   ∧
       is-wf-ex-unit-list(eu-l,env'))


41 is-wf-ctld-grp(<cv,sp-l,eu-l>,env) =
      let cv-sdd = el-sdd(cv,env);
      is-sc-sdd(cv-sdd)     ∧
      1≤i≤lsp-l ⊃
        (let <init,byto,while> = sp-l[i];
         is-prom-conv(cv-sdd,el-sdd(init,env))        ∧
         is-nil(byto)  ∨ is-scomp-tp(s-sdtp(cv-sdd)) ∧
                         (let <by,to> = byto;
                          is-scomp-tp(s-sdtp(el-sdd(by,env)))  ∧
                          is-nil(to)  ∨ is-scomp-tp(s-sdtp(el-sdd(to,env))))  ∧
         is-nil(while)  ∨ is-scomp-tp(s-sdtp(el-sdd(while,env))))


42 is-wf-wh-only-grp(<wh-opt, >,env) =
      is-scomp-tp(el-sdd(wh-opt,env))


44 is-wf-on-st(<snap,cnms,ps>,env) =
      is-proc(ps) ⊃ (∃cprefs,eul)(ps=<id,<>,nil,[q,{},cprefs,REC,eul>
                    /* id is dummy */


45 is-wf-if-st(<test, , >,env) =
      is-scomp-tp(s-sdtp(el-sdd(test,env)))
```

```
47 is-wf-proc-ref(<vlr,arg-l>,env) =
      let sdd = el-sdd(vlr,env);
      is-sentry(sdd)                          ^
      s-ret-descr(sdd) = nil                  ^
      ls-parm-descrs(sdd) = larg-l            ^
      1≤i≤larg-l ⊃
         if is-by-ref-var(arg-l[i])
            then match(s-parm-descrs(c-entry(vlr))[i], /*dd of arg-l[i] */)
            else is-prom-conv(s-parm-descrs(sdd)[i],el-sdd(arg-l[i],env))


48 is-wf-goto-st(<val-ref>,env) =
      el-sdd(val-ref) = <LAB>


52 is-wf-ass-st(<tr-l,expr>,env) =
      1≤i≤ltr-l ⊃ is-prom-conv(el-sdd(tr-l[i],env),el-sdd(expr,env))


53 is-wf-alloc-st(alloc-l,env) =
      1≤i≤lalloc-l ⊃
            (let <id,set-opt> = alloc-l[i];
             let <id,dcl-tp> = env(id);
             is-based(dcl-tp) ^
             set-opt=nil ⊃ is-var-ref(s-dft-qual(dcl-tp))     ^
             set-opt≠nil ⊃ el-sdd(set-opt,env)=<PTR>)


54 is-wf-free-st(free-l,env) =
      1≤i≤lfree-l ⊃
         (let <lq,id> = free-l[i];
          let < ,dcl-tp> = env(id);
          is-based(dcl-tp) ^
          lq=nil ⊃ s-dft-qual(dcl-tp)≠nil
          lq≠nil ⊃ el-sdd(lq,env) = <PTR>)


60 is-wf-inf-expr(<e1,op,e2>,env) =
      let sdd1 = el-sdd(e1,env),
          sdd2 = el-sdd(e2,env);
      is-compatible({sdd1,sdd2})              ^
      op∈{EQ,NE} ∨ cents-scomp-tps(sdd1)
```

```
62 is-wf-pref-expr(<op,e>,env) =
      let sdd = el-sdd(e,env);
      conts-scomp-tps(sdd)


65 is-wf-var-ref(<lq,id,idl,ssc-l>,env) =
      let dtp = s-dcl-tp(env(id));
      dtp∈(prop-var∪parm∪based∪def)          ∧
      is-valid-index(s-dd(dtp),<idl,ssc-l>)  ∧
      lq≠nil ⊃ is-based(dtp)                 ∧
      lq=nil ∨ el-sdd(lq,env)=<PTR>          ∧
      is-based(dtp) ∧ lq=nil ⊃ s-dft-qual(dtp)≠nil


66 is-wf-subscr(ssc,env) =
      is-expr(ssc) ⊃ is-scomp-tp(s-sdtp(el-sdd(ssc,env)))


67 is-wf-proc-fct-ref(<vlr,arg-l>,env) =
      /* like is-wf-proc-ref, except s-ret-descr(sdd) ≠ nil  */


72 is-wf-distr-bif-ref(<nm,arg-l>,env) =
      1≤i≤larg-l ⊃ ¬is-by-ref-var(arg-l[i])  ∧
      (let sdds = {el-sdd(arg-l[i],env) | i∈{1:larg-l}};
       is-compatible(sdds) ∧
       sdd∈sdds ⊃ conts-scomp-tps(sdd)  ∧
      /* see table of distr-el-sdd */


73 is-wf-non-distr-bif-ref(<nm,arg-l>,env) =
      1≤i≤larg-l ⊃ ¬is-by-ref-var(arg-l[i])  ∧
      /* see table of non-distr-el-sdd */


77 is-wf-lab-ref(<id>,env) =
      s-dcl-tp(env(id)) = LAB


78 is-wf-entry-ref(<id>,env) =
      is-proc(env(id))                  ∨
      is-ext-entry(s-dcl-tp(env(id)))
```

```
80 is-wf-st-pv-ref(pv-ref,env) =
      cases pv-ref:
      <STR,<arg-1>> -> (∃tp) (is-all-str(tp,el-sdd(arg-1,env)))
      <SUBSTR,arg-1> -> /* as for SUBSTR bif */


84 is-wf-const(<dtp,symbl>,env) =
      /* symbl "matches" dtp */


108 is-wf-nmd-io-cond(< ,val-ref>,env) =
      el-sdd(val-ref,env) = mk-sc-sdd(FILE)


110 is-wf-file-ref(<id>,env) =
      is-file-const(s-dcl-tp(env(id)))
```

1.2.3 Auxiliary Functions

1.2.3.1 Static Data Description Functions

1.2.3.1.1 Constructors

```
122 intg-sdd() =
      mk-sc-sdd(intg-tp())

    type:   -> sc-sdd


123 bin-fix-sdd() =
      mk-sc-sdd(mk-arith(BIN,FIX,mk-prec(nn(BIN,FIX),0)))

    type:   -> sc-sdd


124 bit-str-sdd() =
      mk-sc-sdd(mk-str-sdd(BIT,*))

    type:   -> sc-sdd
```

```
125 char-str-sdd() =
      mk-sc-sdd(mk-str-sdd(CHAR,*))

    type:   -> sc-sdd


126 entry0() =
      mk-entry(nil,nil,nil)

    type:   -> entry


127 dd-to-sdd(dd) =
      /* inserts * for all elements of s-bpl, s-maxl,
         deletes components not required for sdd and changes constructor names  */

    type: dd -> sdd


128 pdd-to-sdd(pdd) =
      /* similar  */

    type: pdd -> sdd
```

```
129 el-sdd(e) =
      cases e:
      mk-const(dtp, ) ->
         mk-sc-sdd(dtp)
      mk-nmd-const-ref(id) ->
         let dcl = env(id);
         mk-sc-sdd(s-dcl-tp(dcl))
      mk-var-ref( ,id,idl,ssc-l) ->
         let dd = s-dd(s-dcl-tp(env(id)));
         let dd' = select-field(dd,idl);
         let sdd' = dd-to-sdd(dd');
         if (∀i∈{1:lssc-l})(is-expr(ssc-l[i]))
            then sdd'
            else (let st-l = asterisk-l(ssc-l);
                  cases sdd':
                  mk-array-sdd(u-sdd',bpl') ->
                     mk-array-sdd(u-sdd',st-l^bpl')
                  T                         ->
                     mk-array-sdd(sdd',st-l))
      mk-cond-bif-nm(nm) ->
         nm∈{ONCHAR,ONCODE,ONCOUNT} -> intg-sdd()
         nm∈{ONSOURCE,ONFILE,ONKEY} -> char-str-sdd()
      mk-non-distr-bif-ref(nm,arg-l) ->
         non-distr-el-sdd(nm,arg-l)
      mk-proc-fct-ref(val-ref,arg-l) ->
         let mk-sentry( ,srdd) = el-sdd(val-ref,env);
         srdd
      T ->
         let <op,sdd-l> =
            cases e:
            mk-inf-expr(e1,op,e2) -> <op,<el-sdd(e1),el-sdd(e2)>>
            mk-pref-expr(op,e1)   -> <op,<el-sdd(e1)>>
            mk-distr-bif-ref(op,arg-l) -> <op,<el-sdd(arg-l[i])|i∈{1:larg-l}>>;
         distr-el-sdd(op,sdd-l,if op∈{FIX,FLT,BIN,DEC,PREC}
                               then ts-arg-list(e)
                               else nil)
```

type: expr ENV-t -> sdd

```
130 distr-el-sdd(op,sdd-l,prec) =
      let l = {1:lsdd-l};
      (∃i∈l)(is-array-sdd(sdd-l[i])) ->
         let bpl be s.t. (∃i∈l)(is-array-sdd(sdd-l[i]) ∧ s-bpl(sdd-l[i])=bpl)
         mk-array-sdd(distr-el-sdd(op,<(if is-array-sdd(sdd-l[i])
                                        then s-unit-dd(sdd-l[i])
                                        else sdd-l[i]) | i∈l>,prec),bpl)
      (∃i∈l)(is-struct-sdd(sdd-l[i])) ->
         let length be s.t. (∃i∈l)(is-struct-sdd(sdd-l[i]) ∧ lsdd-l[i]=length);
         mk-struct-sdd(<distr-el-sdd(op,<(if is-struct-sdd(sdd-l[i])
                                           then sdd-l[i,j]
                                           else sdd-l[i] | i∈l>,prec)
                                           | j ∈ {1:length}>)
      T ->
         let sdtp-l = if ¬op ∈ {BIN,DEC,FIX,FLT,PREC}
                        then <s-sdtp(sdd-l[i]) | i∈l>
                        else <eval-prec-bifs(op,s-sdtp(sdd-l[1]),prec)>
         sc-el-sdd(op,sdtp-l)

   type: (inf-op | pref-op | distr-bif-nm) sdd* [const*] -> sdd
```

```
131 non-distr-el-sdd(op,arg-l) =
      let <arg1,...,argn> = arg-l;
      result is(/* defined in Table 1 */)

   type: non-distr-bif-nm expr* -> sc-sdd
```

Table 1:

op	constraint of arg_1 abstract syntax el-sdd	constraint of arg_2 abstract syntax el-sdd	el-sdd
ADDR	is-var-ref	----------	mk-sc-sdd(PTR)
COLLATE | DATE | TIME	----------	----------	char-str-sdd()
DIM | HBOUND | LBOUND	is-var-ref is-array-sdd	is-expr is-sc-sdd ∧ is-scomp-tp°s-sdtp	intg-sdd()
NULL	----------	----------	mk-sc-sdd(PTR)
STR	is-expr conts-scomp-tps	----------	if comp-str-tps(arg_1) = {BIT} then bit-str-sdd() else char-str-sdd()

```
132 eval-prec-bifs(op,sdtp,arg-l) =
      let <b1,s1> = der-bs({mk-sc-sdd(sdtp)});
      let base  = if op≠DEC ∧ b1=BIN then BIN else DEC,
          scale = if op≠FLT ∧ s1=FIX then FIX else FLT;
      let mk-prec(p1,q1) = conv-prec(sdtp,base,scale);
      let num-l = <s-num(eval-comp-restr-expr(intg-tp(),arg-l[i]))|i∈{1:larg-l}>;
      result is(mk-sc-sdd(mk-arith(base,scale,mk-prec(p,q))))
         /* where p,q given in Table 2 */

   type: (BIN | DEC | FIX | FLT | PREC) sdtp const* -> sc-sdd
```

Table 2:

op	scale	arg-l	p	q
BIN | DEC	FIX	0	p1	q1
		1	num-l[1]	0
		2	num-l[1]	num-l[2]
	FLT	0	p1	nil
		1	num-l[1]	
FIX		1	num-l[1]	0
		2	num-l[1]	num-l[2]
FLT		1	num-l[1]	nil
PREC	FIX	1	num-l[1]	0
		2	num-l[1]	num-l[2]
	FLT	1	num-l[1]	nil

```
133 sc-el-sdd(op,sdtp-l) =
      op ∈ {BIN,DEC,FLT,FIX,PREC}   -> hsdtp-l
      op ∈ arith-op  ->
        let <b,s> = der-bs(Rsdtp-l);
        let <<p1,q1>,...,<pn,qn>> = <conv-prec(sdtp-l[i],b,s) | 1≤i≤sdtp-l>;
        result is(mk-sc-sdd(mk-arith(b,s,mk-prec(p,q))))
           /* where p, q are defined in Table 3 */
      T   -> result is(el-sdd)
           /* where el-sdd is defined in Table 4 */

  type: (inf-op | pref-op | distr-bif-nm) scomp-tp* -> sc-sdd
```

Table 3:

op	lsdtp-l		p	q
MAX MIN	1 - n	s = FIX	min(nn(b,s), max($p_1-q_1,\ldots,p_n-q_n$) + max($q_1,\ldots,q_n$))	max($q_1,\ldots,q_n$)
		s = FLT	max($p_1,\ldots,p_n$)	nil
ADD SUBT	2	s = FIX	min(nn(b,s), max(p_1-q_1, p_2-q_2) + max(q_1,q_2) + 1)	max(q_1,q_2)
		s = FLT	max(p_1,p_2)	nil
DIV	2	s = FIX	nn(b,s)	nn(b,s) - $p_1+q_1-q_2$
		s = FLT	max(p_1,p_2)	nil
MOD	2	s = FIX	min(nn(b,s), $p_2 - q_2$ + max(q_1,q_2))	max(q_1,q_2)
		s = FLT	max(p_1,p_2)	nil
MULT	2	s = FIX	min(nn(b,s), $p_1 + p_2 + 1$)	$q_1 + q_2$
		s = FLT	max(p_1,p_2)	nil
MINUS PLUS SIGN	1		p_1	q_1
CEIL FLOOR	1	s = FIX	min(nn(b,s), max($p_1 - q_1 + 1, 1$))	0
		s = FLT	p_1	nil

Table 4:

op	lsdtp-l	el-sdd
BOOL	3	bit-str-sdd()
AND | EQ | GE | GT | LE | LT | NE | OR	2	
NOT	1	
BIT	1 - 2	
CHAR	1 - 2	char-str-sdd()
INDEX | VERIFY	2	intg-sdd()
LENGTH	1	
SUBSTR	2 - 3	der-str-sdd(<hsdtp-1>)
TRANSLATE	2 - 3	der-str-sdd(sdtp-1)
CAT	2	

1.2.3.1.2 Predicates

```
134 is-prom-conv(sdd1,sdd2) =
      let sdd-tp =
           /* like the class sdd (with corresponding subclasses array-sdd-tp,
              struct-sdd-tp, sc-sdd-tp), except that contained scomp-tp's are
              replaced by the constant COMP, and contained sentry's are
              replaced by the constant ENTRY  */;
      let R =
           /* the smallest transitive relation R (hence: ordering) satisfying */
           sc∈sc-sdd-tp ⊃ R(sc,sc)                                        ∧
           <udd,bpl>∈array-sdd-tp ⊃ R(udd,<udd,bpl>)                      ∧
           <udd1,bpl>,<udd2,bpl>∈array-sdd-tp ∧ R(udd1,udd2) ⊃
                 R(<udd1,bpl>,<udd2,bpl>)                                 ∧
           sc∈sc-sdd-tp ⊃ R(sc,mk-struct-sdd-tp(<sc | 1≤i≤n>)) ∧
           (∀i∈{1:n})(R(sdd1[i],sdd2[i])) ⊃ R(mk-struct-sdd-tp(<sdd1[i] | 1≤i≤n>),
                                              mk-struct-sdd-tp(<sdd2[i] | 1≤i≤n>)))
                                                                          */;
      R(/*sdd-tp of*/ sdd1, /*sdd-tp of*/ sdd2)

    type: sdd sdd -> B


135 is-compatible(sdds) =
      let R =
           /* the smallest transitive and symmetric relation R (hence: equivalence)
              satisfying the conditions above  */;
      sdd1,sdd2∈sdds ⊃ R(/*sdd-tp of*/ sdd1, /*sdd-tp of*/ sdd2)

    type: sdd-set -> B




136 conts-scomp-tps(sdd) =
      (∀ dtp ∈ comp-sdtps(sdd))(is-scomp-tp(dtp))

   type: sdd -> B
```

```
137 match(dd1,dd2) =
      /* dd1 and dd2 are the same, except: INIT is ignored, entry matches entry,
         extents in dd1 which are (restricted) exprs match non-REFER extents in
         dd2 which are restricted exprs with the same value, * extents in dd1
         match any extents in dd2  */

    type: (pdd|parm) (pdd|dd) -> B
```

1.2.3.2 Base-Scale-Precision Functions

```
138  der-bs(dtps) =
      (let b = (if (∀e∈dtps)((is-arith(e) ∧ s-base(e)=BIN) ∨
                             (is-str(e) ∧ s-str-tp(e)=BIT))
                then BIN
                else DEC),
           s = (if (∃e∈dtps)(is-arith(e) ∧ s-scale(e)=FLT)
                then FLT
                else FIX);
      result is(<b,s>))

    type: comp-tp-set -> (base scale)
```

```
139 conv-prec(sdtp,b,s) =
      cases sdtp:
      char-str-sdd() -> conv-prec(mk-arith(DEC,FIX,mk-prec(nn(DEC,FIX),0)))
      bit-str-sdd()  -> conv-prec(mk-arith(BIN,FIX,mk-prec(BIN,FIX),0))
      mk-arith(b1,s1,mk-prec(p1,q1)) ->
         let m = nn(b,s);
         result is(mk-prec(p,q))
         /* where p,q are defined in Table 5  */

   type: comp-tp base scale -> prec
```

Table 5:

	b1 = FIX		b1 = FLT	
b	s	s1 = BIN	s1 = DEC	s1 = BIN	s1 = DEC
FIX	BIN	p = p1 q = q1	p = min(ceil(p1*3.32)+1,m) q = ceil(q1*3.32)		
	DEC	p = min(ceil(p1/3.32)+1,m) q = ceil(q1/3.32)	p = p1 q = q1		
FLT	BIN	p = min(p1,m)	p = min(ceil(p1*3.32),m)	p = p1	p = min(ceil(p1*3.32),m)
	DEC	p = min(ceil(p1/3.32),m)	p = min(p1,m)	p = min(ceil(p1/3.32),m)	p = p1

```
140 der-str-sdd(sdd-1) =
      mk-str-sdd(der-tp(sdd-1),*)
```

type: scomp-tp* -> str-sdd

```
141   nn(b,s) =
         /*** the maximum <number-of-digits> ***/
         /*** allowed for the target <base>  ***/
         /*** b, and <scale> s. Function is  ***/
         /*** implementation dependent.      ***/

   type:  base scale  ->  intg
```

```
142  der-arith(sdd1,sdd2) =
      (let <b,s> = der-bs({sdd1,sdd2});
       let <p,q> = conv-prec(sdd1,b,s);
       result is(mk-arith(b,s,mk-prec(p,q))))

   type: comp-tp comp-tp -> arith
   note: result is-edd!
```

1.2.3.3 Text Selection, Construction and Gathering Functions

1.2.3.3.1 Selectors

```
143 select-field(dd,id1) =
      cases dd:
      mk-sc-dd( ) -> dd
      mk-array-dd(u-dd, ) -> select-field(u-dd,id1)
      mk-struct-dd(f-l) ->
         if id1 = <>
            then dd
            else (let i = (Ui)(s-f-nm(f-l[i]) =h id1);
                  select-field(s-f-dd(f-l[i]),t id1)))

   type: dd id* -> dd
```

```
144 c-dcl-tp(id,env) =
      s-dcl-tp(env(id))

   type:  id ENV-t -> dcl-tp
   note: env argument omitted in uses.
```

```
145 c-entry(ref,env) =
      cases ref:
      mk-nmd-const-ref(mk-entry-ref(p-id)) ->
         if is-ext-entry(s-dcl-tp(env(p-id)))
            then s-entry(s-dcl-tp(env(p-id)))
            else /* entry matching proc of env(p-id) */
      mk-var-ref( ,id,idl,sscl) ->
         let dd = s-dd(s-dcl-tp(env(id)));
         let dd' = select-field(dd,idl);
         s-dtp(dd')
      mk-proc-fct-ref(vlr,argl) ->
         let mk-entry( ,rdd, ) = c-entry(vlr);
         rdd

    type: val-ref ENV-t -> entry
    note: env argument omitted in uses.
```

1.2.3.3.2 Constructors

```
146  asterisk-1(sl)=
      (lsl=0   ->  <>,
       hsl=*   ->  <*>^asterisk-1(tsl),
       T       ->  asterisk-1(tsl))

   type: subscr* -> **
```

1.2.3.3.3 Collectors

```
147   ret-sts(t) =
        (is-ex-unit-list(t) ->
                union(ret-sts(t[i] | 1≤i≤lt),
         is-ex-unit(t)      ->
                (let ps = s-prop-st(t);
                 cases ps:
                    (is-bl(ps) ∨ is-iter-grp(ps) ∨ is-non-iter-grp(ps)
                           ->   ret-sts(s-ex-unit-list(ps)),
                     mk-if-st(e,eu1,eu2)
                           ->   ret-sts(eu1) ∪ ret-sts(eu2),
                     is-ret-st(ps)
                           ->   {ps},
                     T     ->   {})),
         t=nil             ->   {})

  type: (ex-unit* | [ex-unit]) -> ret-st-set


148   labels(t) =
        (is-proc(t) ∨ is-bl(t) ∨ is-iter-grp(t) ∨ is-non-iter-grp(t)
             ->    labels(s-ex-unit-list(t)),
         is-ex-unit-list(t)
             ->    conc(labels(t[i] | 1≤i≤lt)),
         is-ex-unit(t)
             ->    (let <cprefs,snms,ps> = t;
                    let snms' =
                         (is-iter-grp(ps)  ∨
                          is-non-iter-grp(ps)   -> labels(s-ex-unit-list(ps)),
                          is-if-st(ps)          -> (let <e,eu1,eu2> = ps;
                                                    labels(eu1) ^ labels(eu2)),
                          T                     -> <>),
                    result is(snms^ snms')),
         T   ->    <>)

  type: text -> id-list
```

```
149 rel-evd-io-cond-nms(prog) =
      {mk-evd-io-cond-nm(ioc,c-uid(id)) |
       (∃fd)(<id,<fd, >>∈comp-dcls(prog) ∧ is-file-descr(fd) ∧
             (ioc=ENDFILE ∧ (fd=<REC,UPD,SEQ,nil,nil, > ∨ fd=<,IN,,,,>)  ∨
              ioc=ENDPAGE ∧  fd=<STR,OUT,nil,PR ,nil, >                  ∨
              ioc=RECORD  ∧  fd=<REC,   ,   ,   ,   , >                  ∨
              ioc=KEY     ∧  fd=<REC,   ,DIR,   ,   , >                  ∨
              ioc=UDF                                                    ∨
              ioc=TMI                                                     ))}

    type: prog -> evd-ic-cond-nm-set
```

1.2.3.4 Text Predicates

```
150 is-unique-cprefs(cprefs) =
      cn1,cn2∈cprefs ∧ s-comp-cond-nm(cn1)=s-comp-cond-nm(cn2) ⊃ cn1=cn2

   type: cond-pref-set -> B


151 is-unique-ids(id1) =
      1≤i,j≤lid1 ∧ id1[i]=id1[j] ⊃ i=j

   type:  id* -> B
```

```
152  is-valid-index(dd,<id1,sl>) =
      (is-sc-dd(dd) ->
         (id1=<> ∧ sl=<>),
       is-array-dd(dd) ->
         (let <dd1,bpl> = dd;
         ((sl=<> ∧ id1=<>)                    ∨
          (sl≠<> ∧ lsl≥lbpl) ∧
          is-valid-index(dd1,<id1,<<sl[i]>| lbpl+1≤i≤lsl>>))),
       T ->
         (let <<id[j],dd[j]>| 1≤j≤ldd> = dd;
          if lid1=0
             then (lsl=0)
             else
                (let k = (εk'∈{1:ldd}) (id[k']=hid1);
                 is-valid-index(dd[k],<tid1,sl>))))
```

type: id (id* subscr*) -> B

```
153 is-restr-expr(e) =
      comp-val-refs(e) ⊆ (nmd-const-ref ∪ {mk-non-distr-bif-ref(NULL,<>)})   ∧
      comp-inf-ops(e)  ⊆ {ADD,SUBT,MULT,DIV,CAT}                             ∧
      /* evaluation of e does not raise conditions  */
```

type: expr -> B

```
154 conts-restr-exprs(dd) =
      /* all contained expressions are restricted */
```

type: (dd|pdd) -> B

```
155 is-refer-geom(dd) =
      /* all refer objects precede the refer subject referencing them */
```

type: dd -> B

```
156 conts-no-refers(dd) =
      ext∈comp-extents(dd) ⇒ s-refer-opt(ext)=nil
```

type: dd -> B

```
157 conts-no-inits(dd) =
      sc∈comp-sc-sdds(dd) ⇒ s-init(sc)=nil

   type: dd -> B


158 is-starless(dd) =
      str∈comp-strs(dd) ⇒ s-maxl(dd)≠* ∧
      add∈comp-array-dds(dd) ∧ 1≤i≤ls-bpl(add) ⇒ s-bpl(add)[i]≠*

   type: dd -> B
```

1.2.3.5 Evaluation of Restricted Expressions

```
159 eval-restr-expr(e) =
      /* similar to relevant parts of eval-expr, but pure function:
         no side-effects or interrupts */

   type: expr -> VAL
   pre:  is-restr-expr(e)


160 eval-comp-restr-expr(rdd,e) =
      /* analoguous to eval-comp-expr:
         eval-restr-expr(e) followed by conversion */

    type: sdd expr -> VAL
```

D2. States, Auxiliary Parameters

Contents

1 Σ = REF -> S|ES|FS|AID|OE| /* a few other maps and elementary objects */
constr: $(\forall r \in \underline{D}\sigma)(r \in \underline{ref} V \Rightarrow \sigma(r) \in V)$

The following five references

2 S ES FS AA PA

of types refS, refES, ...(i.e. with $\sigma(r)$ ranging over the sets of equal name defined in the following subsections) are assumed as global state components, e.g. by dcl in a fictitious outermost block.

```
3  initialise-state() =
      S  := [ ],    /*the empty map */
      AA := { },
      PA := { }

   type: =>
```

2.1 Activation Identifiers

```
4  AID = /* infinite set of unigue names */
```

```
5  AA  = AID-set
```

```
6  PA  = AID-set
```

```
7  extend-AA() =
      let new-aid be s.t.
                  new-aid ∈ AID ∧
                  ¬(new-aid ∈ (cAA ∪ cPA));
      AA := cAA ∪ {new-aid};
      return(new-aid)

   type: => AID
```

```
8  restrict-AA(aid) =
      AA := cAA \ {aid};
      PA := cPA ∪ {aid}

   type: AID =>
```

2.2 Values, Locations, and Storage

2.2.1 Evaluated Data Descriptions (edd's)

```
9  edd         =  array-edd | struct-edd | sc-edd
10 array-edd   :: s-unit-edd:(sc-edd|struct-edd) s-bpl:ebp+
11 ebp         :: s-lb:intg s-ub:intg     /*constr: s-lb(ebp) ≤ s-ub(ebp) */
12 struct-edd  :: edd+
13 sc-edd      =  arith | str-edd | ENTRY | LAB | PTR | FILE
14 str-edd     :: s-tp:str-tp s-maxl:intg s-vary:varity /*constr: s-maxl(edd) ≥ 0 */
```

Indices

```
15 index = intg | (intg | *)+
```

```
16 augm-indices(ebpl) =
      {il ∈ (intg|*)+ | ↓il = ↓ebpl ∧
                        (∀k≤↓il) (is-intg(il[k]) ⇒ s-lb(ebpl[k])≤il[k]≤s-ub(ebpl[k]))}

   type: ebp+ -> (intg|*)+-set
```

```
17 indices(ebpl) =
      augm-indices(ebpl) ∩ intg+

   type: ebp+ -> intg+-set
```

```
18 augm-index-lists(edd) =
      cases edd:
      mk-array-edd(u-edd,ebpl) ->
         {<ind>^indl' | ind∈augm-indices(ebpl) ∧ indl'∈augm-index-lists(u-edd)}
      mk-struct-edd(edd-l) ->
         {<i>^indl' | 1≤i≤edd-l ∧ indl'∈augm-index-lists(edd-l[i])}
      T -> {<>}

   type: edd -> index*-set
```

Auxiliary Functions

```
19 width(edd) =
      cases edd:
      mk-array-edd(u-edd,ebpl) ->
               width(u-edd)* prod <s-ub(ebpl[i]) - s-lb(ebpl[i]) + 1 | 1≤i≤lebpl>
      mk-struct-edd(edd-l) -> sum <width(edd-l[i]) | 1≤i≤ledd-l>
      mk-str-edd(,maxl,) -> maxl
      T -> 1

   type: edd -> intg
```

```
20 is-all-str(tp,edd) =
      cases edd:
      mk-array-edd(u-edd,ebpl) -> is-all-str(tp,u-edd)
      mk-struct-edd(edd-l)     -> (∀i∈{1:ledd-l}) (is-all-str(tp,edd-l[i])
      mk-str-edd(tp',vy)       -> tp'=tp ∧ vy=NONVARYING
      T                        -> false

   type: str-tp edd -> B
```

2.2.2 Values

```
21 VAL          =  ARRAY-VAL | STRUCT-VAL | SC-VAL

22 ARRAY-VAL    :: (intg+ -> VAL)
                      constr: mk-ARRAY-VAL(m)∈ARRAY-VAL ⊃
                              Dm = indices(ebpl) ∧ (∀ind∈Dm)(m(ind) ∈ values(u-edd))
                              for some mk-array-edd(u-edd,ebpl)

23 STRUCT-VAL   :: VAL+
24 SC-VAL       =  STR-VAL | ELEM-VAL | ?
25 STR-VAL      =  CHAR-STR-VAL | BIT-STR-VAL
26 CHAR-STR-VAL:: s-val-l:CHAR-VAL*
27 BIT-STR-VAL  :: s-val-l:BIT-VAL*

28 mk-STR-VAL(tp)(vl) =
       (tp = CHAR -> mk-CHAR-STR-VAL, tp = BIT -> mk-BIT-STR-VAL)

   type: (CHAR -> (CHAR-VAL* -> CHAR-STR-VAL)) | (BIT -> (BIT-VAL* -> BIT-STR-VAL))

29 CHAR-VAL     =  symbol | ?
30 BIT-VAL      =  0-BIT | 1-BIT | ?
31 ELEM-VAL     =  NUM-VAL | PTR-VAL | ENTRY-VAL | LAB-VAL | FILE-VAL
32 NUM-VAL      :: s-tp:arith s-num:NUM
33 NUM          =  /***** (subset of rational) real numbers including intg *****/
34 PTR-VAL      =  NULL-PTR | PROP-PTR-VAL
35 ENTRY-VAL    :: s-enf:(entry LOC* OE CBIF => [VAL]) s-id:ID s-aid:AID
36 LAB-VAL      :: s-id:ID s-aid:AID
37 FILE-VAL     :: uid
38 CMP-VAL      =  ARRAY-VAL | STRUCT-VAL | STR-VAL
```

2.2.2.1 Undefined Values

```
39 udf-val(edd) =
      cases edd:
      mk-array-edd(u-edd,ebpl) ->
         mk-ARRAY-VAL([ il → udf-val(u-edd) | il ∈ indices(ebpl) ])
      mk-struct-edd(edd-l)      ->
         mk-STRUCT-VAL(<udf-val(edd-l[ i ]) | 1≤i≤ledd-l>)
      mk-str-edd(tp,lgth,NONVARYING) ->
         mk-STR-VAL(tp)(<? | 1≤i≤lgth>)
      T   -> ?

   type: edd -> VAL
```

```
40 is-defined-val(v) =
      cases v:
      mk-ARRAY-VAL(m) | mk-STRUCT-VAL(vl) ->
         (∀ind ∈ v-indices(v))(is-defined-val(comp-val(v,ind)))
      mk-STR-VAL(tp)(vl) -> (∀i≤lvl)(vl[i ]≠?)
      T -> v≠?

   type: VAL -> B
```

2.2.2.2 Value-set of an edd

```
41 values(edd) =
      cases edd:
      mk-arith(b,sc,pr) -> {<edd,num> | <edd,num> ∈ NUM-VAL} ∪ {?}
      PTR               -> PTR-VAL ∪ {?}
      ENTRY             -> ENTRY-VAL ∪ {?}
      LAB               -> LAB-VAL ∪ {?}
      FILE              -> FILE-VAL ∪ {?}
      mk-str-edd(tp,lgth,vy) ->
         (vy = NONVARYING -> {mk-STR-VAL()(vl) ∈ str-vals(tp) | lvl=lgth}
          vy = VARYING    -> {mk-STR-VAL()(vl) ∈ str-vals(tp) | 0≤lvl≤lgth} ∪ {?})
      mk-array-edd(u-edd,ebpl) ->
         {mk-ARRAY-VAL(m) | Dm = indices(ebpl) ∧ (∀ind ∈ Dm)(m(ind)∈values(u-edd))}
      mk-struct-edd(edd-l)     ->
         {mk-STRUCT-VAL(vl) | lvl = ledd-l ∧ (∀i≤ledd-l)(vl[i] ∈ values(edd-l[i]))}

   type: edd -> B VAL
```

```
42 str-vals(tp) =
      {mk-STR-VAL(tp)(vl) |
       (tp=CHAR ∧ is-CHAR-VAL-list(vl)) ∨ (tp=BIT ∧ is-BIT-VAL-list(vl))}

   type: (CHAR -> CHAR-STR-VAL) | (BIT -> BIT-STR-VAL)
```

2.2.2.3 Components of Strings and Aggregates

```
43 v-augm-indices(v) =
      cases v:
      mk-ARRAY-VAL(m) -> (let ebpl be s.t. indices(ebpl) = Dm;
                          augm-indices(ebpl))
      mk-STRUCT-VAL(vl) -> {i | 1≤i≤lvl}
      mk-STR-VAL()(vl) -> {<i,j> | 1≤i ∧ i-1≤j≤lvl}

   type: CMP-VAL -> index-set
```

```
44 v-indices(v) =
      cases v:
      mk-ARRAY-VAL(m) -> Dm
      mk-STRUCT-VAL(vl) -> {i | 1≤i≤lvl}
      mk-STR-VAL()(vl) -> {<i,i> | 1≤i≤lvl}

   type: CMP-VAL -> (intg|intg+)-set
```

```
45 v-ordered-indices(v) =
      /* same, except returning list; row-major order for arrays */

   type: CMP-VAL -> (intg|intg+)*
```

```
46 comp-val(v,ind) =
      cases v:
      mk-ARRAY-VAL(m) -> (is-intg-list(ind) -> m(ind), T -> v-cross-sect(m, ind))
      mk-STRUCT-VAL(vl) -> vl[ind]
      mk-STR-VAL(tp)(vl) -> (let <i,j> = ind;
                             mk-STR-VAL(tp)(<vl[k] | i≤k≤j>))

   type: CMP-VAL index -> VAL
   pre:  ind ∈ v-augm-indices(v)
```

```
47 v-cross-sect(m,ind) =
      let ebpl be s.t. indices(ebpl) = Dm;
      let ebpl1 = conc<(is-*(ind[i]) -> <ebpl[i]>, T -> <>) | 1≤i≤lind>;
      let m1 =
           [ind1 → m(<(is-intg(ind[i]) -> ind[i], ind[i] = k-th * in ind -> ind1[k])
                       | 1≤i≤lind>)
             | ind1 ∈ indices(ebpl)];
      mk-ARRAY-VAL(m1)

   type: (intg+->VAL) (intg|*)+ -> ARRAY-VAL
```

2.2.3 Locations

```
48 LOC          =  ARRAY-LOC | STRUCT-LOC | SC-LOC

49 ARRAY-LOC    :: (intg* -> LOC)
                     constr: mk-ARRAY-LOC(m) ∈ ARRAY-LOC ⊃
                               Dm = indices(ebpl) ∧ (∀ind∈Dm)(l-edd(m(ind)) = u-edd)
                               for some mk-array-edd(u-edd,ebpl)
                     note:   locations violating this constraint arise via
                             substr-pv, see Comm.

50 STRUCT-LOC   :: LOC*
51 SC-LOC       =  STR-LOC | ELEM-LOC
52 STR-LOC      =  CHAR-STR-LOC | BIT-STR-LOC
53 CHAR-STR-LOC:: s-vary:varity s-loc-l:CHAR-LOC*
54 BIT-STR-LOC :: s-vary:varity s-loc-l:BIT-LOC*

55 mk-STR-LOC(tp) =
       (tp = CHAR -> mk-CHAR-STR-LOC, tp = BIT -> mk-BIT-STR-LOC)

   type: (CHAR -> (varity CHAR-LOC* -> CHAR-STR-LOC)) |
         (BIT -> (varity BIT-LOC* -> BIT-STR-LOC))
   note: mk-STR-LOC(tp)(vy,<>) assumed not unique for empty <>, see Comm.

56 ELEM-LOC     =  /* not further analyzed, but see l-edd(l)  */
57 CHAR-LOC     =  /* not further analyzed */
58 BIT-LOC      =  /* not further analyzed */
59 CMP-LOC      =  ARRAY-LOC | STRUCT-LOC | STR-LOC
60 ATM-LOC      =  ELEM-LOC | CHAR-LOC | BIT-LOC



61 l-edd(l) =
       cases l:
       mk-ARRAY-LOC(m) -> (⊍ mk-array-edd(u-edd,ebpl))
                           (indices(ebpl)=Dm ∧ (∀ind∈Dm)(l-edd(m(ind))=u-edd))
       mk-STRUCT-LOC(loc-l) -> mk-struct-edd(<l-edd(loc-l[i]) | 1≤i≤lloc-l>)
       mk-STR-LOC(tp)(vy,loc-l) -> mk-str-edd(tp,lloc-l,vy)
       el ∈ ELEM-LOC -> /* some sc-edd other than str-edd */

   type: LOC -> edd
```

```
62 l-indices(l) =
      cases l:
      mk-ARRAY-LOC(m)              -> Dm
      mk-STRUCT-LOC(loc-l)         -> {i | 1≤i≤lloc-l}
      mk-STR-LOC(tp)(vy,loc-l)     -> {<i,i> | 1≤i≤lloc-l}

   type: CMP-LOC -> index-set



63 l-ordered-indices(l) =
      /* same, except returning list */

   type: CMP-LOC -> index*



64 comp-loc(l,ind) =
      cases l:
      mk-ARRAY-LOC(m) -> (is-intg-list(ind) -> m(ind), T -> l-cross-sect(m,ind))
      mk-STRUCT-LOC(loc-l) -> loc-l[ind]
      mk-STR-LOC(tp)(vy,loc-l) ->
         (let <i,j> = ind;
          mk-STR-LOC(tp)(NONVARYING, <loc-l[k] | i≤k≤j>))

   type: CMP-LOC index -> LOC
   pre:  analogucus to pre of comp-val



65 l-cross-sect(m,ind) =
      let ebpl1,m1 = /* like in v-cross-sect */;
      mk-ARRAY-LOC(m1)

   type: (intg+ -> LOC) (intg|*)+ -> ARRAY-LOC
```

```
66 sub-loc(l,indl) =
      indl = <> -> l
      T         -> (let ind = hindl,
                        indl1 = tindl;
                    let l' = comp-loc(l,ind);
                    (is-ARRAY-LOC(l) ∧ ¬is-intg-list(ind) ->
                       (let m'=[ind' → sub-loc(comp-loc(l',ind'),indl1)
                                | ind' ∈ l-indices(l') ];
                        array-loc(m'))
                     T -> sub-loc(l',indl1)))
```

type: LOC index* -> LOC
pre: indl ∈ augm-index-lists(l-edd(l))

```
67 array-loc(m) =
      let <ebpl,u-edd> be s.t.
            (Dm = indices(ebpl) ∧ (∀ind∈Dm)(l-edd(m(ind)) = u-edd));
      (is-array-edd(u-edd) ->
        (let <u-edd',ebpl'> = u-edd;
         let mm' = [ind^ind' → comp-loc(m(ind),ind')
                    | ind∈Dm ∧ ind'∈indices(ebpl') ];
         mk-ARRAY-LOC(mm'))
       T -> mk-ARRAY-LOC(m))
```

type: (intg+ -> LOC) -> ARRAY-LOC
pre: <m> is an array-loc, except that u-edd may again be an array-edd.

```
68 ordered-sc-locs(l) =
      is-SC-LOC(l) -> <l>
      T -> (let indl = l-ordered-indices(l);
            conc<ordered-sc-locs(comp-loc(l,indl[i])) | 1≤i≤lindl>)
```

type: LOC -> SC-LOC*

```
69 sc-locs(l) = R(ordered-sc-locs(l))
```

type: LOC -> SC-LOC-set

```
70 ordered-atm-locs(l) =
      is-STR-LOC(l) ∧ ¬(s-vary(l)=VARYING ∧ l(s-loc-l(l))=0) -> s-loc-l(l)
      is-SC-LOC(l) -> <l>
      T -> (let indl = l-ordered-indices(l);
            conc<ordered-atm-locs(comp-loc(l,indl[i])) | 1≤i≤lindl>)

   type: LOC -> ATM-LOC*


71 atm-locs(l) = R(ordered-atm-locs(l))

   type: LOC -> ATM-LOC-set
```

2.2.3.1 Level-One Locations, Independence

```
72 1-LOC ⊆ LOC


73 1-LOC = AUTO-LOC | BASED-LOC
           constr: AUTO-LOC ∩ BASED-LOC = {}


74 is-indep(l,l') =
      atm-locs(l) ∩ atm-locs(l') = {}

   type: LOC LOC -> B


75 (∀l∈CMP-LOC)(∀ind,ind'∈l-indices(l))
      (ind≠ind' ⊃ is-indep(comp-loc(l,ind),comp-loc(l,ind')))


76 (∀l,l'∈1-LOC)(l≠l' ⊃ is-indep(l,l'))
```

2.2.3.2 Connected, Left-to-Right Equivalence.

```
77 is-conn(l) =
      (∃l'∈l-LOC,loc-l1,loc-l2)
         (ordered-atm-locs(l') = loc-l1^ordered-atm-locs(l)^loc-l2)

   type: LOC -> B
```

```
78 is-l-to-r-edd(edd,edd') =
      edd = edd' ∨
      (is-struct-edd(edd) ∧ is-struct-edd(edd') ∧
         (∃edd-l1)(edd' = edd^edd-l1)∨
         (∃edd-l1,edd1,edd1',edd1-l')
            (edd = edd-l1^<edd1> ∧ edd' = edd-l1^<edd1'>^edd1-l' ∧
             is-l-to-r-edd(edd1,edd1')))

   type: edd edd -> B
```

```
79 is-l-to-r-loc(l,l') =
      is-l-to-r-edd(l-edd(l),l-edd(l')) ∧
      (∃loc-l)(ordered-sc-locs(l') = ordered-sc-locs(l)^loc-l)

   type: LOC LOC -> B
```

2.2.3.3 Pointers

```
80 PROP-PTR-VAL ::  /*characterized implicitly by formulae 81 to 86 */
```

```
81 addr(l) = /* the properties of this function are defined in formulae 83 to 86 */

   type: LOC -> PROP-PTR-VAL
   pre: is-conn(l)
```

```
82 constr-loc(ptr,edd) =
      ptr = NULL-PTR -> error
      T -> /* the properties of  this function are defined in formula 83 */

   type: PTR-VAL edd -> LOC
```

```
83 (∀l∈LOC)(is-conn(l) ⊃ (constr-loc(addr(l), l-edd(l)) = l))
```

```
84 (∀l,l'∈LOC)(is-indep(l,l') ∧ width(l-edd(l))≠0 ∧ width(l-edd(l'))≠0 ⊃
                  addr(l) ≠ addr(l'))
```

```
85 (∀l,l'∈LOC)(is-conn(l')∧is-l-to-r-loc(l,l') ⊃ addr(l)=addr(l'))
```

```
86 (∀l∈LOC,tp)(is-conn(l) ∧ is-all-str(tp,l-edd(l)) ∧ width(l-edd(l))≥1  ⊃
                 (let l' = mk-STR-LOC(tp)((ordered-atm-locs(l))[1]);
                  addr(l) = addr(l')))
```

2.2.4 Storage

```
87 S = LOC -> VAL
       constr: let (f0:L0->VAL) = stg;
               L0 ⊆ 1-LOC ∧ (∀l∈L0)(f0(l) ∈ values(l-edd(l)))
```

```
88 parts(l) =
      is-STR-LOC(l) ∧ s-vary(l)=NONVARYING ->
            {comp-loc(l,<i,i> | 1≤i≤l(s-loc-l(l))}
      is-SC-LOC(l) -> {l}
      T -> union{parts(comp-loc(l,ind)) | ind ∈ l-indices(l)}

   type: LOC -> LOC-set
```

```
89 cur-parts(l,v) =
      is-STR-LOC(l) ->
         (s-vary(l)=NONVARYING -> parts(l)
          s-vary(l)=VARYING ->
            (cases v:
             ? -> {l}
             mk-STR-VAL()(vl) -> {comp-loc(l,<i,i>) | 1≤i≤lvl} ∪ {l})
      is-SC-LOC(l) -> {l}
      T -> union{cur-parts(comp-loc(l,ind),comp-val(v,ind))
                 | ind ∈ l-indices(l)}

   type: LOC VAL -> LOC-set
   pre:  v ∈ values(l-edd(l))


90 locs(stg) =
      let (f0:L0->VAL) = stg;
      {l | parts(l) ⊆ union{cur-parts(l0,f0(l0)) | l0∈L0}}

   type: S -> LOC-set


91 extend(stg) =
      let (f0:L0->VAL) = stg;
      let L = locs(stg);
      (λf:L->VAL)
         ((∀l∈L)(f(l) ∈ values(l-edd(l))) ∧
          (∀l∈L0)(f(l) = f0(l)) ∧
          (∀l∈LnCMP-LOC)(∀ind∈l-indices(l))
            (comp-loc(l,ind)∈L ⊃ f(comp-loc(l,ind)) = comp-val(f(l),ind)))

   type: S -> (LOC -> VAL)
```

2.2.5 Allocate, Free, Contents & Assignment

```
92 alloc(edd,loc-tp,env-cond) =
      let l: find-new-loc(edd,loc-tp,env-cond);
      S := cS + [l → udf-val(edd)];
      return(l)

   type: edd (AUTO|BASED) (OE CBIF) => 1-LOC
```

```
93 find-new-loc(edd,loc-tp,env-cond) =
      let (f0: L0->VAL): cS;
      let l: let locs = (loc-tp = AUTO -> AUTO-LOC
                         loc-tp = BASED -> BASED-LOC)
             (∃l∈locs)(¬(l∈L0) ∧ l-edd(l) = edd) ->
                    /* choose such an l */
              T  -> (raise-cond(STG,true,env-cond));
                     find-new-loc(edd,loc-tp,env-cond));
      return(l)

   type: edd (AUTO|BASED) (OE CBIF) => 1-LOC
```

```
94 free(l) =
      let (f0:L0->VAL): cS;
      if l∈L0
         then S := f0\{l}
         else error

   type: LOC =>
```

```
95 cont(l) =
      let f0: ⊆S;
      if l∈locs(f0)
         then
            let (f:L->VAL): extend(f0);
            let v = f(l);
            (is-defined-val(v) -> return(v), T -> error)
         else error

   type: LOC => VAL
```

```
96 assign(l,v) =
      let (f0:L0->VAL): ⊆S;
      let (f:L->VAL) = extend(f0);
      if l∈L
         then
            let f0':L0->VAL be s.t.
               (let (f':L'->VAL) = extend(f0');
                f'(l) = v ∧
                (∀l'∈L')(is-indep(l',l) -> f'(l') = f(l'),
                         is-STR-LOC(l') ∧ s-vary(l')=VARYING ∧ ¬(l'∈sc-locs(l)) ->
                           cur-parts(l',f'(l')) = cur-parts(l',f(l'))));
            S := f0'
         else error

   type: LOC VAL =>
   pre:  v ∈ values(l-edd(l))
```

```
97 cur-length(l) =
      let mk-STR-VAL()(vl): cont(l);
      return (lvl)

   type: STR-LOC =>  int
```

2.3 External Storage, File State

2.3.1 External Storage

```
98  ES          =  DS-LOC -> DS-VAL
99  DS-LOC      =  REC-LOC  |  STREAM-LOC
100 REC-LOC     =  K-R-LOC  |  R-LOC
101 DS-VAL      =  REC-VAL*  |  STREAM-VAL*
102 REC-VAL     =  [K-VAL]  VAL
103 STREAM-VAL  =  CHAR-VAL | LNMRK | FGMRK | CARRET
104 K-VAL       =  CHAR-VAL+
```

Constraints: (1) rules 98,99,101: REC goes with REC, STREAM with STREAM; (2) rules 100, 102: K-R-LOC ⊃ K-VAL≠nil; (3) Uniqueness of keys: Let es∈ES, dl∈Des s.t. dl∈K-R-LOC, let dv=es(dl), then (∀i,j∈{1:ldv}) (i≠j ⊃ hdv[i]≠hdv[j]).

2.3.2 File State

```
105 FS          =  uid -> ...
106 uid         =  /*** unique identifiers; generated by c-uid(id) ***/
```

2.3.3 Unique File Constant Identifiers, uid's

```
107 c-uid(id)   =  /*unique id corresponding to file constant id, see Comm. */

    type: id -> uid


108 file-id(uid) =
    let id be s.t. c-uid(id) = uid;
    /* id represented as CHAR-VAL-list */

    type: uid -> CHAR-VAL+
```

2.4 Environment

```
109 ENV =  ID -> DEN
110 DEN =  LOC | ENTRY-VAL | LAB-VAL | DEF-DEN | BASED-DEN | FILE-VAL
111 DEF-DEN = (OE CBIF => LOC)
112 BASED-DEN = ([LCC] OE CBIF =>) ([PTR-VAL] id* OE CBIF => LOC)
113 BL-ENV = (ID -> (LOC | ENTRY-VAL | FILE-VAL)) [ref(ACTIVE|INACTIVE)]

114 bl-env(pb) = /* context function yielding relevant BL-ENV */

    type:   (proc|bl) -> BL-ENV
```

2.5 On Establishment

```
115 OE              =  evd-cond-nm -> ou-ENTRY-VAL
116 ou-ENTRY-VAL    =  OE CBIF =>
117 evd-cond-nm     =  non-ic-cond-nm  |  evd-io-cond-nm
118 evd-io-cond-nm  :: io-cond uid
```

2.6 Cbif Part

```
119 CBIF =  cond-bif-nm -> (LOC|NUM-VAL|CHAR-STR-VAL)
              constr:  cond-bif-nm=ONSOURCE ⇒ LOC
                       cond-bif-nm∈{ONCHAR,ONCOUNT,ONCODE} ⇒ NUM-VAL
                       cond-bif-nm∈{ONKEY,ONFILE} ⇒ CHAR-STR-VAL
```

THIS PAGE HAS BEEN INTENTIONALLY LEFT BLANK.

F1. Block Structure

Contents

1.1 Programs

```
1  int-prog(prog,main-id) =
     let pb-sels = comp-proc-sels(prog) u comp-bl-sels(prog),
         nonrec-sels = {spb ∈ comp-proc-sels(prog) | s-recity(spb(prog)) = nil},
         st-ext-sels = {sdcl ∈ comp-dcl-sels(prog) |
                        sdcl(prog) = mk-dcl(id,mk-prop-var(,mk-static-cl(EXT)))};
     let st-ext-ids = {id | (∃sdcl ∈ st-ext-sels)(s-id(sdcl(prog)) = id)},
         st-int-ids(pb) =
           {id | mk-dcl(id,mk-prop-var(,mk-static-cl(INT))) ∈ s-dcls(pb)};
     initialise-state();
     let aid-0: extend-AA();
     let st-ext-locs: [id → (let sdcl' ∈ st-ext-sels be s.t. s-id(sdcl'(prog))=id;
                             eval-static-dcl-tp(s-dd(sdcl'(prog)),<[ ],[ ],[ ]>)) |
                       id ∈ st-ext-ids],
     let st-int-locs:
           [spb → [id → (let dcl ∈ s-dcls(spb(prog)) be s.t. s-id(dcl) = id;
                         eval-static-dcl-tp(s-dd(dcl),<[ ],[ ],[ ]>)) |
                   id ∈ st-int-ids(spb(prog))] | spb ∈ pb-sels];
     dcl actys := [spb → INACTIVE | spb ∈ nonrec-sels];
     let env-1 = [s-id(p)→<eval-proc-dcl(p,env-1,false),s-id(p),aid-0> | p∈prog'],
         prog' =
           (let bl-env'(spb) =
                 <(let dcls = s-dcls(spb(prog));
                   [id → st-ext-locs(id) |
                    mk-dcl(id,mk-prop-var(,mk-static-cl(EXT)))∈dcls]
                   u st-int-locs(spb)
                   u [id → c-uid(id) | mk-dcl(id,mk-file-const(,)) ∈ dcls]
                   u [id → env-1(id) | mk-dcl(id,mk-ext-entry()) ∈ dcls]),
                  (spb ∈ nonrec-sels -> spb°actys, Γ → nil)>;
            /* prog modified by associating, for each spb ∈ pb-sels, a BL-ENV
               with spb(prog) so that bl-env(spb(prog)) = bl-env'(spb) */,
         main-en-f = eval-proc-dcl((Up ∈ prog')(s-id(p) = main-id),env-1,true);
     let oe-1 = [cn → system-ou-entry-val(cn) |
                 cn ∈ non-io-cond-nm u rel-evd-io-cond-nms(prog)];
     main-en-f(entry0(),oe-1,[ ]);
     for all id ∈ st-ext-ids do free(st-ext-locs(id)),
     for all spb ∈ pb-sels do
         for all id ∈ st-int-ids(spb(prog)) do free(st-int-locs(spb)(id));
     restrict-AA(aid-0)

  type: prog id =>
  pre: s-parms(main-proc) = <>, s-ret-descr(main-proc) = nil.
```

1.2 Blocks

```
2  int-bl(bl,<env,oe,cbif>) =
      let <dcls,procs,eu-l> = bl,
          <st-env,nil> = bl-env(bl);
      int-bl-1(dcls,procs,eu-l,<env+st-env,oe,cbif>)

   type: bl (ENV OE CBIF) =>
```

```
3  int-bl-1(dcls,procs,eu-l,<env,oe,cbif>) =
      let aid: extend-AA();
      let nenv:
         (trap exit (abn) with error:
          let lenv:
               [id → eval-dcl(dtp,nenv,<env,oe,cbif>) |
                  mk-dcl(id,dtp) ∈ dcls ∧
                  (dtp ∈ (based∪def) ∨ is-prop-var(dtp)∧s-stg-cl(dtp)=AUTO) ] ∪
               [id → <id,aid> | mk-dcl(id,LAB) ∈ dcls] ∪
               [s-id(prcc) → <eval-proc-dcl(proc,nenv,[false]),s-id(proc),aid> |
                  proc ∈ procs]  /* false passed iff proc EXTERNAL */;
          return(env+lenv));
      dcl loer := oe;
      (trap exit(abn) with (bl-epilogue(dcls,nenv,aid); exit(abn));
       int-ex-unit-list(eu-l,<nenv,loer,oe,cbif,aid>));
      bl-epilogue(dcls,nenv,aid)

   type: dcl-set proc-set ex-unit* (ENV OE CBIF) =>
```

```
4  bl-epilogue(dcls,nenv,aid) =
      for all mk-dcl(id,mk-prop-var(dd,AUTO)) ∈ dcls do free(nenv(id));
      restrict-AA(aid)

   type: dcl-set ENV AID =>
```

1.3 Procedures

1.3.1 Definition

```
5  eval-proc-dcl(proc,nenv,major) =
       let <id,parm-l,ret-dd,dcls,procs,cprefs,recy,eu-l> = proc;
       let en-f(dtp,loc-l,oe,cbif) =
            (let <st-env,acty> = bl-env(proc);
             if recy = nil        /* assert acty ≠ nil */
                then if cacty = ACTIVE then error else acty := ACTIVE;
             let n = lparm-l;
             let parm-dcls = <(∪dcl∈dcls)(s-id(dcl) = parm-l[i]) | i∈{1:n}>;
             if ¬entry-match(<s-dd(s-dcl-tp(parm-dcls[i])) | 1≤i≤n>,ret-dd,dtp)
                then error;
             let env' = nenv + ([parm-l[i] → loc-l[i] | 1≤i≤n] ∪ st-env);
             trap exit(ctl,arg) with
                (if recy = nil then acty := INACTIVE;
                 if ctl = ret
                   then if ret-dd≠nil then return (arg)
                   else exit(ctl,arg));
             int-bl-1(dcls \ {parm-dcls[i] | 1≤i≤n},procs,eu-l,<env',oe,cbif>));
       en-f

   type: proc ENV [B] -> (entry LOC* OE CBIF => [VAL])
   assert: call of int-bl-1 terminates abnormally(with go or ret).


6  entry-match(parm-dd-l,ret-dd,mk-entry(pdd-l,rdd, )) =
       (if rdd=nil then ret-dd=nil else ret-dd≠nil ∧ match(rdd,ret-dd)) ∧
       pdd-l≠nil ∧ lpdd-l=lparm-dd-l ∧ (∀i∈{1:lpdd-l})(match(parm-dd-l[i],pdd-l[i])

   type: dd* [pdd] entry -> B
```

1.3.2 Call

```
7  int-call-st(mk-call-st(en-ref,arg-l),<env,loe,cbif>) =
     let <dtp,en-f,loc-l>: eval-proc-ref(en-ref,arg-l,<env,loe,cbif>);
     (trap exit (abn) with
       (free-dummy(arg-l,loc-l);
        exit(abn));
      en-f(dtp,loc-l,loe,cbif));
     free-dummy(arg-l,loc-l)

   type: call-st (ENV OE CBIF) =>
```

```
8  eval-proc-ref(en-ref,arg-l,env-ex) =
     let dtp = c-entry(en-ref);
     let <,loe,cbif> = env-ex;
     let <en-f,nm,aid>: eval-expr(en-ref,env-ex);
     if ¬(aid ∈ cAA) then error;
     let mk-entry(pdd-l, , ) = dtp;
     trap exit (abn) with error;
     let loc-l:
         <(if is-by-ref-var(arg-l[i])
            then eval-l-var-ref(arg-l[i],env-ex)
            else
             (let arg-v: eval-expr(arg-l[i],env-ex);
              let dummy-loc: alloc(complete-pdd(pdd-l[i],arg-v),AUTO,<loe,cbif>);
              prom-ass(dummy-loc,arg-v,<loe,cbif,cur-cond-prefs(arg-l) >);
              return(dummy-loc)))
          | 1 ≤ i ≤ larg-l>;
     return(<dtp,en-f,loc-l>)

   type: val-ref arg* (ENV OE CBIF) => entry (entry LOC* OE CBIF => [VAL]) LOC*
   BASIS-11: Assume any order allowed cf 6.3.6.1.1.
             Ban on goto fits with prologue, not yet resolved with ANS cf 6.3.7
```

```
9  complete-pdd(pdd,v) =
     /* returns an edd like the pdd but with *'s filled in from v,[1:1] if scalar*/

   type: pdd VAL -> edd
```

```
10 free-dummy(arg-l,loc-l) =
      for all i∈{1:larg-l} do if ¬is-by-ref-var(arg-l[i]) then free(loc-l[i])

   type: arg* LOC* =>
```

1.3.3 Return

```
11 int-ret-st(mk-ret-st(t),<env,loe,cbif>) =
      let proc = /* the (statically known) proc terminated by this mk-ret-st(t) */;
      if is-expr(t)
         then (let rdd = s-ret-descr(proc);
               let v: eval-expr(t,<env,loe,cbif>);
               let cv:prom-conv(complete-pdd(rdd,v),v,<loe,cbif,cur-cond-prefs(t)>);
               exit(ret,cv))
         else (if /* proc is EXTERNAL, static property */
                  then (let major = /*switch passed down from eval-proc-dcl */;
                        if major then raise-cond(FINISH,true,<loe,cbif>));
               exit(ret,nil))

   type: ret-st (ENV OE CBIF) =>
```

F2. Declarations, Reference, Allocation

Contents


```
1  eval-dcl(dcl-tp,nenv,env-ex) =
      is-prop-var(dcl-tp)-> eval-auto-dcl-tp(dcl-tp,env-ex)
      is-based(dcl-tp)    -> eval-based-dcl-tp(dcl-tp,nenv)
      is-def(dcl-tp)      -> eval-def-dcl-tp(dcl-tp,nenv,env-ex)

   type: dcl-tp ENV (ENV OE CBIF) => DEN
   pre:  dcl-tp∈(based ∪ def) ∨ is-prop-var(dcl-tp)∧s-stg-cl(dcl-tp)=AUTO
```

```
2  eval-l-ref(vr,env-ex) =
      if is-var-ref(vr)
         then eval-l-var-ref(vr,env-ex)
         else eval-l-stg-pv-ref(vr,env-ex)

   type: (var-ref|stg-pv-ref) (ENV OE CBIF) => LOC
```

```
3  eval-l-var-ref(mk-var-ref(lq,id,idl,sl),env-ex) =
      let <env,loe,cbif> = env-ex;
      let dt = c-dcl-tp(id);
      let  main-loc:
        (is-prop-var(dt) v is-parm(dt) -> eval-l-prop-ref(id,env-ex)
         is-based(dt) -> eval-l-based-ref(lq,id,idl,env-ex)
         is-def(dt) -> eval-l-def-ref(id,env-ex))
      let esl: eval-subscr-list(sl,env-ex);
      let indl = compose-indl(s-dd(dt),idl,esl);
      if indl ∈ augm-index-lists(l-edd(main-loc))
         then return(sub-loc(main-loc,indl))
         else raise-cond(SUBRG,is-enab(SUBRG,cur-cond-prefs(id)),<loe,cbif>);

   type: var-ref (ENV OE CBIF) => LOC
   pre:  is-valid-index(s-dd(dt),<idl,sl>)
```

2.1 Proper Variables

```
4  eval-static-dcl-tp(dd,env-ex) =
      let < ,loe,cbif> = env-ex;
      let edd: eval-dd(dd,env-ex);
      let l: alloc(edd,AUTO,<loe,cbif>);
      int-init(dd,l,env-ex);
      return(l)

   type: dd (ENV OE CBIF) => LOC
   pre: only used with dd's associated with static-cl
   assert: thus no side effects possible because no general expressions.
```

```
5  eval-auto-dcl-tp(mk-prop-var(dd,stg-cl),env-ex) =
      let < ,loe,cbif> = env-ex;
      let edd: eval-dd(dd,env-ex);
      let l: alloc(edd,AUTO,<loe,cbif>);
      int-init(dd,l,env-ex);
      return(l)

   type: prop-var (ENV OE CBIF) => LOC
```

```
6  eval-l-prop-ref(id,<env,,>) =
      return(env(id))
```

type: id (ENV OE CBIF) => LOC
pre: the declaration of id is proper var or parameter.
assert: env(id) is always defined (because: context cond all referenced ids are declared and because env is always properly passed)
env(id) for automatic, parameters and static is a location

2.2 Based Variables

```
7  eval-based-dcl-tp(mk-based(dd,dft-qual),nenv) =
      let alloc-based(set-opt-r,oe-r,cbif-r) =
              (let set-opt: if set-opt-r ≠ nil
                               then set-opt-r
                               else eval-l-var-ref(dft-qual,<nenv,oe-r,cbif-r>);
               let edd: eval-dd(dd,<nenv,oe-r,cbif-r>);
               let l: alloc(edd,BASED,<oe-r,cbif-r>);
                  (assign(set-opt,addr(l)),
                   init-refer-obs(dd,edd,l));
               int-init(dd,l,<nenv,oe-r,cbif-r>)) ,
          ref-based(qual-r,idl,ce-r,cbif-r) =
              (let qual: if qual-r ≠ nil
                            then qual-r
                            else eval-expr(dft-qual,<nenv,oe-r,cbif-r>);
               let dd-sub = left-struct-part(dd,idl);
               let l: based-lcc(qual,dd-sub);
               return(l)) ;
      <alloc-based,ref-based>
```

type: based (->ENV) -> ((LOC|nil) OE CBIF =>) ((PTR-VAL|nil) id* OE CBIF => LOC)
assert: set-opt defined because of context cond
qual defined because of context cond

```
8  init-refer-obs(dd,edd,l) =
       for all <indl,s> ∈ refers(dd) do
          (let i = s(edd);
           let v = mk-NUM-VAL(intg-tp(),i);
           assign(sub-loc(l,indl),v))
```

type: dd edd LOC =>
pre: all refer objects are intg-tp().

```
9  refers(dd) =
       {<indl,s> | is-index-list(indl) ∧ s∈comp-extent-sels(dd) ∧
                   (let mk-extent(e,idl) = s(dd);
                    idl≠nil ∧ indl=compose-indl(dd,idl,<>))}
```

type: dd -> (index* COMP-SEL)-set

```
10 based-loc(qual,dd) =
       let l = (ιl)
               (∃edd)(l = constr-loc(qual,edd) ∧
                      is-instance(edd,dd) ∧
                      (∀<ind-l,s> ∈ refers(dd))(let v: cont(sub-loc(l,ind-l));
                                                s(edd) = s-num(v)));
       return(l)
```

type: PTR-VAL dd => LOC
BASIS-11: defines l-to-r (check-based-ref) only one level down.

```
11 is-instance(edd,dd) =
       /* shapes match, non-refer extents in dd(constants) agree with values in edd */
```

type: edd dd -> B

```
12 left-struct-part(dd,idl) =
      if idl=<> v ¬is-struct-dd(dd)
         then dd
         else (let id = hidl;
               let mk-struct-dd(dd-l) = dd;
               let i = (Ui)(s-f-nm(dd[i]) = id);
               mk-struct-dd(<dd-l[j] | j∈{1:i-1}> ^
                     <<id,left-struct-part(s-f-dd(dd-l[i]),tidl)>>))

   type: dd id* -> dd
```

```
13 eval-l-based-ref(lq,m-id,idl,env-ex) =
      let <env,oe,cbif> = env-ex;
      let < ,ref-based> = env(m-id);
      let ptr-val: if lq ≠ nil then eval-expr(lq,env-ex) else nil;
      let m-loc: ref-based(ptr-val,idl,oe,cbif);
      return(m-loc)

   type: [val-ref] id id* (ENV OE CBIF) => LOC
```

```
14 int-alloc-st(mk-alloc-st(allcc-l),env-ex) =
       let <env,oe,cbif> = env-ex;
       trap exit (arg) with error;
       for i = 1 to lallcc-l do
        (let <id,set-opt> = alloc-l[i];
         let set-loc: if set-opt=nil
                        then nil
                        else eval-l-var-ref(set-opt,env-ex);
         let <alloc-based, > = env(id);
         alloc-based(set-loc,ce,cbif))

   type: alloc-st (ENV OE CBIF) =>
```

```
15 int-free-st(mk-free-st(free-l),env-ex) =
      let <env,oe,cbif> = env-ex;
      for i = 1 to lfree-l  dc
        (let <lq,id> = free-l[i];
         let ptr-val: if lq ≠ nil then eval-expr(lq,env-ex) else nil;
         let < ,ref-based> = env(id) ;
         let l: ref-based(ptr-val,<>,oe,cbif);
         if is-BASED-LOC(l) then free(l) else error )

   type: free-st (ENV OE CBIF) =>
```

2.3 Defined Variables

```
16 eval-def-dcl-tp(mk-def(dd,base,pos),nenv,env-ex) =
      let edd: eval-dd(dd,env-ex);
      let w = width(edd);
      let eval-def-loc(oe-r,cbif-r) =
            let base-loc: eval-l-var-ref(base,<nenv,oe-r,cbif-r>);
            if ¬is-conn(base-loc) then error;
            let i: s-num(eval-comp-expr(pos,intg-tp(),<nenv,oe-r,cbif-r>));
            let loc-l = ordered-atm-locs(base-loc);
            if 1 ≤ i  ∧  i + w - 1 ≤ lloc-l
               then (let l be s.t.  l-edd(l) = edd ∧
                                    ordered-atm-locs(l) = <loc-l[j] | i≤j≤i+w-1>;
                     return(l))
               else error;
      return(eval-def-loc)

   type: def ENV (ENV OE CBIF) => (OE CBIF => LOC)


17 eval-l-def-ref(m-id,<env,oe,cbif>) =
      let eval-def-loc = env(m-id);
      let m-loc: eval-def-loc(oe,cbif);
      return(m-loc)

   type: id (ENV OE CBIF) => LOC
```

2.4 Storage Pseudo Variable References

```
18 eval-l-stg-pv-ref(vr,env-ex) =
      cases vr:
      mk-stg-pv-ref(STR,arg-l) ->
        (let str-tp = /* statically known str-tp of components of arg-l[1] */;
         let loc: eval-l-ref(arg-l[1],env-ex);
         let atms = ordered-atm-locs(loc);
         return(mk-STR-LOC(str-tp)(NONVARYING,atms)))
      mk-stg-pv-ref(SUBSTR,arg-l) ->
        (let cprefs = cur-cond-prefs(vr);
         let st-tp  = /* edd, shape of el-sdd(arg-l[2]), sc-tp all intg-tp() */;
         let len-tp = /* edd, shape of el-sdd(arg-l[3]), sc-tp all intg-tp() */;
         let b-loc: eval-l-var-ref(arg-l[1],env-ex),
             st: eval-comp-expr(arg-l[2],st-tp,env-ex),
             len: (arg-l = 3 -> eval-comp-expr(arg-l[3],len-tp,env-ex)
                   T         -> nil);
         let res-loc: distrib-substr-pv(cprefs,b-loc,st,len,env-ex);
         return(res-loc))

   type: stg-pv-ref (ENV OE CBIF) => LOC
```

```
19 distrib-substr-pv(cprefs,b-loc,st,len,env-ex) =
      let <env,loe,cbif> = env-ex;
      cases b-loc:
      mk-STR-LOC( ) ->
        (let k = cur-length(b-loc);
         let i = s-num(st);
         let j = (is-VAL(len) -> s-num(len)
                  T           -> k - i + 1);
         if 1≤i≤j+i≤k+1
            then return(comp-loc(b-loc,<i , i + j - 1>))
            else raise-cond(STRG,is-enab(STRG,cprefs),<loe,cbif>))
      mk-STRUCT-LOC(loc-l) ->
         mk-STRUCT-LOC(<distrib-substr-pv(cprefs,loc-l[i],
                              (is-STRUCT-VAL(st)-> st[i]
                               T -> st),
                              (is-STRUCT-VAL(len)-> len[i]
                               T->len),env-ex) | iε{1:lloc-l}>)
      mk-ARRAY-LOC(m) -> /* similar, but incl bound checks, fail gives error */
```

type: cond-pref-set LOC VAL [VAL] (ENV OE CBIF) => LOC

2.5 Initial

```
20 evd-init-elem = init-elem | evd-simple-init | evd-iterated-init

21 evd-simple-init :: VAL

22 evd-iterated-init :: s-iter-fact:intg  evd-init-elem+
```

```
23 int-init(dd,l,env-ex) =
      let edd: l-edd(l);
      for all sεinit-comp(dd) do
        (let indll = sc-indices(s,dd,edd),
             mk-sc-dd(dtp,iel) = s(dd);
         init-sc-parts(indll,iel,l,env-ex,cur-cond-prefs(dd)))
```

type: dd LOC (ENV OE CBIF) =>

```
24 init-sc-parts(indll,eiel,l,env-ex,cprefs) =
     let <env,oe,cbif> = env-ex;
     let env-op = <oe,cbif,cprefs>;
     if indll=<> v eiel=<>
        then
           I
        else
          (cases heiel:
           mk-iterated-init(if,eiel1)|mk-evd-iterated-init(if,eiel1)->
             (let n: if is-intg(if)
                        then if
                        else s-num(eval-comp-expr(if,intg-tp(),env-ex));
              let eiel2: eval-init-elem-list(eiel1,env-ex);
              let eiel3 = (conc <eiel2|1≤i≤n>)^teiel;
              init-sc-parts(indll,eiel3,l,env-ex,cprefs))
           mk-evd-simple-init(v)->
             (prom-ass(sub-loc(l,hindll),v,env-op);
              init-sc-parts(tindll,teiel,l,env-ex,cprefs))
           mk-simple-init(ex)->
             (let v: eval-expr(ex,env-ex);
              prom-ass(sub-loc(l,hindll),v,env-op);
              init-sc-parts(tindll,teiel,l,env-ex,cprefs))
           * ->
              init-sc-parts(tindll,teiel,l,env-ex,cprefs))

   type: (index*)* evd-init-elem* LOC (ENV OE CBIF) cond-pref-set =>
```

```
25 sc-indices(s,dd,edd) = /* yields a list of index lists to scalar parts
                             corresponding to s(dd), dimensions are inherited */

   type: COMP-SEL dd edd -> (index*)*
   pre:  is-instance(edd,dd)
```

```
26 eval-init-elem-list(iel,env-ex) =
                  /* yields a list of the same shape as iel, but with possibly
                     some iterations factors being replaced by their evaluated
                     intgs, and possibly some other expressions
                     being replaced by their values */

   type: evd-init-elem* (ENV OE CBIF) => evd-init-elem*
```

```
27 init-comp(dd) =
      {s∈comp-sc-dd-sels(dd) | s-init(s(dd)) ≠ nil}

   type: dd -> COMP-SEL-set
```

2.6 Auxiliary Definitions

```
28 eval-dd(dd,env-ex) =
     cases dd:
     mk-array-dd(u-dd,bpl)->
       (let ev-u-dd: eval-dd(u-dd,env-ex),
            ev-bpl: <mk-ebp(eval-extent(s-lb(bpl[i]),env-ex),
                            eval-extent(s-ub(bpl[i]),env-ex))| i∈{1:lbpl}>;
        if (∀i∈{1:lev-bpl}) (s-lb(ev-bpl[i]) ≤s-ub(ev-bpl[i]))
           then return(mk-array-edd(ev-u-dd,ev-bpl))
           else error)
     mk-struct-dd(sdd-l) ->
        return(mk-struct-edd(<eval-dd(s-f-dd(sdd-l[i]),env-ex)|i∈{1:lsdd-l}>))
     mk-sc-dd(dtp, ) ->
       (is-str(dtp)->
          (let mk-str(tp,maxl,vy) = dtp;
           let e-maxl: eval-extent(maxl,env-ex);
           if e-maxl ≥ 0
              then return(mk-str-edd(tp,e-maxl,vy))
              else error)
        is-entry(dtp) -> return(ENTRY)
        T             -> return(dtp))

   type: dd (ENV OE CBIF) => edd
   pre:  no contained bp or maxl may be *.
         all contained extents must be comp.
```

```
29 eval-extent(mk-extent(expr, ),env-ex) =
     let cv: eval-comp-expr(expr,intg-tp(),env-ex);
     return(s-num(cv))

   type: extent (ENV OE CBIF) => intg
   pre: extents must be convertable to intg-tp().
```

```
30 compose-indl(dd,idl,esl) =
      cases dd:
      mk-array-dd(un-dd,bpl) ->
         if esl = <>
            then                              /* assert: id-l = <> */
               <>
            else
              (let esl1,esl2 be s.t. esl1^esl2 = esl  ∧  lesl1 = lbpl;
               esl1^compose-indl(un-dd,idl,esl2))
      mk-struct-dd(ssd-l)->
         if idl = <>
            then                              /* assert esl = <> */
                 <>
            else(let i = (ιi)(s-f-nm(ssd-l[i]) = hidl);
                 <i>^compose-indl(s-f-dd(ssd-l[i]),tidl,esl))
      mk-sc-dd( , )->
                                              /* assert idl = esl = <> */
         <>

   type: dd id* (*|intg)* -> index*
   pre:  is-valid-index(dd,<idl,esl>)


31 eval-subscr-list(sl,env-ex) =
      <(sl[i] = * -> *
        T         -> s-num(eval-comp-expr(sl[i],intg-tp(),env-ex))) | i∈{1:lsl}>

   type: (expr|*)* (ENV OE CBIF) => (intg|*)*
```

THIS PAGE HAS BEEN INTENTIONALLY LEFT BLANK.

F3. Statements

Contents


```
1  int-ex-unit(eu,env-eu) =
      let mk-ex-unit(cprefs,snms,t) = eu;
      trap exit (gc,<lab-t,aid-t>) with
         if aid-t=s-aid(env-eu) ∧ lab-t∈snms
            then                                  /* assert pre-cue-int-ex-unit */
               cue-int-ex-unit(eu,lab-t,env-eu)
                                                  /* assert post */
            else exit(gc,<lab-t,aid-t>);
                                                  /* assert post, from test */
      int-prop-st(t,env-eu)
                                                  /* assert (if no exit) post vacuous */

   type: ex-unit (ENV refOE OE CEIF AID) =>
   post: result = <σ',<gc,<lab-t,aid-t>>> ⊃ (aid-t ≠ s-aid(env-eu) ∨
                              ¬is-contained-lab(lab-t,eu))
```

```
2  cue-int-ex-unit(eu,lab,env-eu) =
      let mk-ex-unit(cprefs,snms,t) = eu;
      trap exit (go,<lab-t,aid-t>) with
         if aid-t=s-aid(env-eu) ∧ lab-t∈snms
            then                                  /* assert pre-cue-int-ex-unit */
               cue-int-ex-unit(eu,lab-t,env-eu)
                                                  /* assert post */
            else exit(go,<lab-t,aid-t>);
                                                  /* assert post, from test */
      if lab ∈ snms
         then int-prop-st(t,env-eu)
                                                  /* assert (if no exit) post vacuous */
         else
            (cases t:
             mk-if-st( , , ) ->
                                                  /* assert pre-cue-int-if-st (pre) */
                    cue-int-if-st(t,lab,env-eu)
                                                  /* assert (if no exit) post vacuous */
             mk-non-iter-grp(eu-l) ->
                                                  /* assert pre-cue-int-ex-unit-list */
                    cue-int-ex-unit-list(eu-l,lab,env-eu)
                                                  /* assert (if no exit) post vacuous */
             T ->                                 /* is-iter-grp(t) */
                    error)
```

```
type: ex-unit id (ENV refOE OE CBIF AID) =>
pre:  is-contained-lab(lab,eu)
post: result = <σ',<go,<lab-t,aid-t>>> ⊃ (aid-t ≠ s-aid(env-eu) ∨
                        ¬is-contained-lab(lab-t,eu))
```

```
3  is-contained-lab(lab,t) =
     cases t:
     mk-ex-unit( ,snms,st) ->
         lab ∈ snms ∨
           (cases st:
            mk-ctld-grp(,,eu-l)       -> is-contained-lab(lab,eu-l)
            mk-wh-only-grp(,eu-l)     -> is-contained-lab(lab,eu-l)
            mk-non-iter-grp(eu-l)     -> is-contained-lab(lab,eu-l)
            mk-if-st(,then-u,else-u) -> is-contained-lab(lab,then-u) ∨
                                        is-contained-lab(lab,else-u)
            T -> false)
     T -> (∃i∈{1:lt})(is-contained-lab(lab,t[i]))

   type: id (ex-unit*|ex-unit) -> B


4  int-prop-st(st,env-eu) =
     let <env,loer,boe,cbif,aid> = env-eu;
     let env-ex = <env,cloer,cbif>;
    (is-on-st(st)          -> int-on-st(st,<env,loer,cbif>)
     is-rev-st(st)         -> int-rev-st(st,<env,<loer,boe>,cbif>)
     is-non-iter-grp(st)   -> (let mk-non-iter-grp(eu-l) = st;
                               int-ex-unit-list(eu-l,env-eu))
     is-iter-grp(st)       -> int-iter-grp(st,env-eu)
     is-if-st(st)          -> int-if-st(st,env-eu)
     is-bl(st)             -> int-bl(st,env-ex)
     is-call-st(st)        -> int-call-st(st,env-ex)
     is-goto-st(st)        -> int-goto-st(st,env-ex)
     is-sig-st(st)         -> int-sig-st(st,env-ex)
     is-ret-st(st)         -> int-ret-st(st,env-ex)
     is-io-st(st)          -> int-io-st(st,env-ex)
     is-ass-st(st)         -> int-ass-st(st,env-ex)
     is-alloc-st(st)       -> int-alloc-st(st,env-ex)
     is-free-st(st)        -> int-free-st(st,env-ex)
     is-NULL(st)           -> I)

   type: prop-st (ENV refOE OE CBIF AID ) =>
   post: result = <σ',<go,<lab-t,aid-t>>> ⊃ (aid-t ≠ s-aid(env-eu) ∨
                          ¬is-contained-lab(lab-t,mk-ex-unit(cprefs,snms,st)))
   assert:  all labels within blocks, procs get new aids cf. F1
            all labels within on-units get new aids cf. F4
            stmts without nested ex-units cannot have local labs
            thus, with the remaining post conds, this post true
```

3.1 Non-iterative Grcups

```
5  int-ex-unit-list(eu-l,env-eu) =
      trap exit (qo,<lab-t,aid-t>) with
         if aid-t = s-aid(env-eu) ∧ is-contained-lab(lab-t,eu-l)
            then                                  /* assert pre-cue-int-ex-unit-list */
               cue-int-ex-unit-list(eu-l,lab-t,env-eu)
                                                  /* assert post */
            else exit(qc,<lab-t,aid-t>);
                                                  /* assert post, from test */
      iter-ex-unit-list(eu-l,1,env-eu)
                                                  /* assert (if no exit) post vacuous */

   type: ex-unit* (ENV refOE OE CBIF AID) =>
   post: result = <σ',<qc,<lab-t,aid-t>>> ⊃ aid-t ≠ s-aid(env-eu) ∨
                                                  ¬is-contained-lab(lab-t,eu-l)



6  cue-int-ex-unit-list(eu-l,lab,env-eu) =
      trap exit (qo,<lab-t,aid-t>) with
         if aid-t = s-aid(env-eu) ∧ is-contained-lab(lab-t,eu-l)
            then                                  /* assert pre-cue-int-ex-unit-list */
               cue-int-ex-unit-list(eu-l,lab-t,env-eu)
                                                  /* assert post */
            else exit(qc,<lab-t,aid-t>);
                                                  /* assert post, from test */

      (let i = (ιi)(is-contained-lab(lab,eu-l[i])) ;
                                                  /* assert (∃i) from pre */
                                                  /* assert pre-cue-int-ex-unit (pre, i)*/
       cue-int-ex-unit(eu-l[i],lab,env-eu);
                                                  /* assert (if no exit) post vacuous */
       iter-ex-unit-list(eu-l,i + 1,env-eu))
                                                  /* assert (if no exit) post vacuous */

   type: ex-unit* id (ENV refOE OE CBIF AID) =>
   pre:  is-contained-lab(lab,eu-l)
   post: result = <σ',<qc,<lab-t,aid-t>>> ⊃ (aid-t ≠ s-aid(env-eu) ∨
                                                  ¬is-contained-lab(lab-t,eu-l))
```

```
7  iter-ex-unit-list(eu-l,i,env-eu) =
      for j = i to leu-l do
         int-ex-unit(eu-l[j],env-eu)

   type: ex-unit* intg (ENV refOE OE CBIF AID) =>
```

3.2 Iterative Groups

```
8  int-iter-grp(t,env-eu) =
      let <env,loer,,cbif,> = env-eu;
      cases t:
      mk-wh-only-grp(test,eu-l) ->
          while (eval-truth(test,<env,cloer,cbif>)) do
                  int-ex-unit-list(eu-l,env-eu)
      mk-ctld-grp(cv,sp-l,eu-l) ->
         (let cv-loc: eval-targ-ref(cv,<env,cloer,cbif>);
          for i = 1 to lsp-l do
           (let <init,byto,while> = sp-l[i];
            cases byto:
            nil      -> int-init-wh-do(cv-loc,init,while,eu-l,env-eu)
            <by,to> -> int-step-do(cv-loc,el-sdd(cv),init,by,to,while,eu-l,env-eu)))

   type: iter-grp (ENV refOE OE CBIF AID) =>
   post: result = <σ',<qo,<lab-t,aid-t>>> ⊃ (aid-t ≠ s-aid(env-eu) ∨
                                             ¬is-contained-lab(lab-t,t))
   assert:  c.c. gives pre of eval-truth
            no labels local to exprs, post-int-ex-unit-list gives post first clause
            is-sc-LOC(cv-loc)
            post of int-init-wh-do or iter-step-do gives post of this
```

```
9  int-step-do(cv-loc,cv-dd,init,by,to,while,eu-l,env-eu) =
     let cprefs = cur-cond-prefs(init),
         <env,loer,,cbif,> = env-eu;
     let env-ex = <env,cloer,cbif>;
     let init-val: eval-expr(init,env-ex),
         by-val: eval-comp-expr(by,der-arith(el-sdd(by),cv-dd),env-ex),
         to-val: if is-expr(to)
                   then eval-comp-expr(to,der-arith(el-sdd(to),cv-dd),env-ex)
                   else return(nil);
     prom-ass(cv-loc,init-val,<cloer,cbif,cprefs>);
     while (let b: if is-expr(to)
                     then (let cv-val: cont(cv-loc);
                           let c-cv-val: conv(der-arith(cv-dd,el-sdd(to)),
                                              cv-val,<cloer,cbif,cprefs>,SIZE);
                           if s-num(by-val)≥0 ∧ s-num(c-cv-val)>s-num(to-val) ∨
                              s-num(by-val)<0 ∧ s-num(c-cv-val)<s-num(to-val)
                              then return(false)
                              else if is-expr(while)
                                     then eval-truth(while,env-ex)
                                     else return(true))
                     else if is-expr(while)
                            then eval-truth(while,env-ex)
                            else return(true);
            return(b)) do
           (int-ex-unit-list(eu-l,env-eu);
            let cv-val: cont(cv-loc);
            let c-cv-val: conv(der-arith(cv-dd,el-sdd(by)),cv-val,
                               <cloer,cbif,cprefs>,nil);
            let new-cv-val: arith-rep(s-num(c-cv-val) + s-num(by-val),
                              der-arith(cv-dd,el-sdd(by)),<cloer,cbif,cprefs>,nil);
            prom-ass(cv-loc,new-cv-val,<cloer,cbif,cprefs>))
```

type: SC-LOC sc-edd expr expr [expr] [expr] ex-unit* (ENV refOE OE CBIF AID) =>

post: result = <σ',<go,<lab-t,aid-t>>> ⇒ (aid-t ≠ s-aid(env-eu) ∨
¬is-contained-lab(lab-t,eu-l))

BASIS-11: Converts by-val once per iteration, odd types as well!
Omits conversion to cv-tp, this must be an error!

```
10 int-init-wh-do(cv-loc,init,while,eu-l,env-eu) =
      let <env,loer,,cbif,> = env-eu,
          cprefs = cur-cond-prefs(init);
      let init-val: eval-expr(init,<env,cloer,cbif>);
      prom-ass(cv-loc,init-val,<cloer,cbif,cprefs>);
      let b: if is-expr(while)
               then eval-truth(while,<env,cloer,cbif>)
               else return(true);
      if b
         then int-ex-unit-list(eu-l,env-eu)
         else I
```

type: SC-LOC expr [expr] ex-unit* (ENV refOE OE CBIF AID) =>

3.3 If

```
11 int-if-st(if-st,env-eu) =
      let mk-if-st(test,then-u,else-u) = if-st;
      trap exit (go,<lab-t,aid-t>) with
         if aid-t = s-aid(env-eu) ∧ (is-contained-lab(lab-t,then-u) ∨
                                     is-contained-lab(lab-t,else-u))
            then                                  /* assert pre-cue-int-if-st */
               cue-int-if-st(if-st,lab-t,env-eu)
                                                  /* assert post */
            else exit(go,<lab-t,aid-t>);
                                                  /* assert post,from test */
      let <env,loer,,cbif, > = env-eu;
      let b: eval-truth(test,<env,cloer,cbif>);
      if b
         then int-ex-unit(then-u,env-eu)
         else if is-ex-unit(else-u)
                 then int-ex-unit(else-u,env-eu)
                 else I
```

type: if-st (ENV refOE OE CBIF AID) =>

post: result = <σ',<go,<lab-t,aid-t>>> ⊃ (aid-t ≠ s-aid(env-eu) ∨
¬is-contained-lab(lab-t,if-st)))

```
12 cue-int-if-st(if-st,lab,env-eu) =
      let mk-if-st(test,then-u,else-u) = if-st;
      trap exit (go,<lab-t,aid-t>) with
         if aid-t = s-aid(env-eu) ∧ (is-contained-lab(lab-t,then-u) ∨
                                     is-contained-lab(lab-t,else-u))
            then                                  /* assert pre-cue-int-if-st */
               cue-int-if-st(if-st,lab-t,env-eu)
                                                  /* assert post */
            else exit(go,<lab-t,aid-t>);
                                                  /* assert post, from test */
      if is-contained-lab(lab,then-u)
         then cue-int-ex-unit(then-u,lab,env-eu)
         else cue-int-ex-unit(else-u,lab,env-eu)

   type: if-st id (ENV refOE OE CBIF AID) =>
   pre:  is-contained-lab(lab,if-st)
   post: result = <σ',<go,<lab-t,aid-t>>> ⊃ (aid-t ≠ s-aid(env-eu) ∨
                           ¬is-contained-lab(lab-t,if-st))



13 eval-truth(expr,env-st) =
      let <env,loe,cbif> = env-st;
      let mk-sc-sdd(sdtp) = el-sdd(expr);
      let val: eval-expr(expr,env-st);
      let sc-edd = mk-str-edd(BIT,
                              if is-arith(sdtp) then der-bit-len(sdtp)
                                                else ls-val-l(val),
                              NONVARYING);
      let mk-BIT-STR-VAL(bit-str): conv(sc-edd,val,<loe,cbif,cur-cond-prefs(expr)>);
      return((∃i∈{1:lbit-str})(bit-str[i] = 1-BIT))

   type: expr (ENV OE CBIF) => B
   pre:  el-sdd(expr) must be convertible to BIT.
```

3.4 Goto

```
14 int-goto-st(mk-goto-st(val-ref),env-st) =
      let targ: eval-expr(val-ref,env-st);
      if ¬(s-aid(targ) ∈ cAA) then error;
      exit(go,targ)

   type: goto-st (ENV OE CBIF) =>
   assert:  a.s. gives is-val-ref(val-ref)
            thus is-expr(val-ref)
            c.c. gives is-LAB-VAL(targ)
```

3.5 Assignment

```
15 int-ass-st(ass-st,<env,loe,cbif>) =
      let mk-ass-st(tr-l,rhs) = ass-st;
      let rhs-v: eval-expr(rhs,<env,loe,cbif>),
          targ-lcc: eval-targ-ref(tr-l[1],<env,loe,cbif>);
      prom-ass(targ-loc,rhs-v,<loe,cbif,cur-cond-prefs(ass-st)>);
      for i = 2 to ltr-l do
         (let targ-lcc: eval-targ-ref(tr-l[i],<env,loe,cbif>);
          prom-ass(targ-loc,rhs-v,<loe,cbif,cur-cond-prefs(ass-st)>))

   type: ass-st (ENV OE CBIF) =>
   assert:  context conditions give pre of prom-ass
```

```
16 eval-targ-ref(tr,env-st) =
      if is-cond-pv(tr)
         then eval-l-cond-pv(tr,env-st)
         else eval-l-ref(tr,env-st)

   type: targ-ref (ENV OE CBIF) => LOC
```

```
17 prom-ass(loc,val,env-op) =
      let cval: prom-conv(l-edd(loc),val,env-op);
      assign(loc,cval)
```

type: LOC VAL (OE CBIF cond-pref-set) =>
pre: shape/types must be promotable.
assert: prom-conv only called with promotable shapes, only bounds may differ:
see context conditions
post of prom-conv gives cval ∈ l-values(loc)
thus assign used within pre

F4. Conditions

Contents:

4.1 Condition Handling Functions

4.1.1 The ON Statement

```
1  int-on-st(mk-on-st(snap,cn-l,ps),<env,loer,cbif>)=
       (for i = 1 to lcn-l do
         (is-non-io-cond-nm(cn-l[i])  ->
             loer := cloer + [cn-l[i] → ou-entry-val(<snap,cn-l[i],ps>,env)],
          is-nmd-io-cond(cn-l[i]) ->
             (let mk-nmd-io-cond(ioc,vr) = cn-l[i];
             let fuid : eval-expr(vr,<env,cloer,cbif>);
             let evd-cn = mk-evd-io-cond-nm(ioc,fuid);
             loer := cloer + [evd-cn → ou-entry-val(<snap,evd-cn,ps>,env)])))
```

type: on-st (ENV refOE CBIF) =>

```
2  ou-entry-val(<snap,evd-cn,ps>,env)=
     (let fct1() = if snap=nil
                     then I
                     else cond-impl-def-act(SNAP);
      let fct2(oe,cbif)=
            if ps=SYSTEM
               then
                  (fct1();
                   system-ou-entry-val(evd-cn)(oe,cbif))
               else
                  (fct1();
                   (eval-proc-dcl(ps,env,false))(entry0(),<>,oe,cbif));
      result is(fct2))

   type:  ([SNAP] evd-cond-nm (proc|SYSTEM)) ENV -> ((OE CBIF) => )


3  system-ou-entry-val(evd-cn)=
     (let <cn,fuid> = cases evd-cn:
                        mk-evd-io-cond-nm(ioc,fuid') -> <ioc,fuid'>,
                        T                            -> <evd-cn,nil>;
      let fct(oe,cbif)=
         cases cn:
            FOFL | OFL | SUBRG | ZDIV | CONV | SIZE |
            ENDFILE | KEY | RECORD | TRANSMIT | UNDEFINEDFILE  ->
                  (cond-comment(cn);
                   raise-cond(ERROR,true,<oe,cbif>);
                        /*** assert: next line interpreted iff        ***/
                        /***         no GOTO out of loe(ERROR)!       ***/
                   error),
            UFL | STRG  ->
                  (cond-comment(cn)),
            STRZ | FINISH  ->
                  (I),
            ERROR | STG  ->
                  (cond-impl-def-act(cn)),
            ENDPAGE  ->
                  (put-page(fuid));
      result is(fct))

   type:  evd-cond-nm -> ((OE CBIF) => )
```

4.1.2 The REVERT Statement

```
4  int-rev-st(mk-rev-st(cn-l),<env,<loer,boe>,cbif>)=
      (for i=1 to lcn-l do
        (is-non-io-cond-nm(cn-l[i]) ->
            loer := cloer + [cn-l[i] → boe(cn-l[i])],
         is-nmd-io-cond(cn-l[i]) ->
           (let mk-nmd-io-cond(ioc,vr) = cn-l[i];
            let fuid : eval-expr(vr,<env,cloer,cbif>);
            let evd-cn = mk-evd-io-cond-nm(ioc,fuid);
            loer := cloer + [evd-cn → boe(evd-cn)])))

   type:  rev-st (ENV (refOE OE) CBIF) =>
```

4.1.3 The SIGNAL Statement

```
5  int-sig-st(mk-sig-st(cn),<env,loe,cbif>)=
      (is-comp-cond-nm(cn) ->
            if is-enab(cn,cur-cond-prefs(cn))
               then
                  (cn=CONV) ->
                     (let dummy:
                           raise-conv(true,<mk-CHAR-STR-VAL(<>),0,nil>,<loe,cbif>);
                      I),
                  T ->
                     raise-comp-cond(cn,true,<loe,cbif>))
                        /*** assert: no normal return if                ***/
                        /***         cn∈{FOFL,OFL,SIZE,SUBRG,ZDIV,STRG} ***/
               else I,
      is-nmd-io-cond(cn) ->
            (let mk-nmd-io-cond(ioc,vr) = cn;
             let fuid : eval-expr(vr,<env,loe,cbif>);
             if (ioc=KEY)
                then raise-evd-io-cond(<KEY,fuid,mk-CHAR-STR-VAL(<>)>,<loe,cbif>)
                else raise-evd-io-cond(<ioc,fuid,nil>,<loe,cbif>),
      T ->
            /*** assert: cn ∈ {ERROR,STG,FINISH} ***/
            raise-cond(cn,true,<loe,cbif>))

   type:  sig-st (ENV OE CBIF) =>
```

4.1.4 The Raise-Condition Functions

```
6  raise-cond(evd-cn,enabled,<loe,cbif>)=
      (let cbif-1 = if ¬(ONCODE ∈ Dcbif)
                    then
                       [ONCODE → integer(oncode-val(evd-cn))]
                          /*** assert: {¬(ONCODE ∈ Dcbif) ⇒ cbif=[ ] ***/
                    else
                       cbif;
      is-comp-cond-nm(evd-cn) ->
         raise-comp-cond(evd-cn,enabled,<loe,cbif-1>),
      is-evd-io-cond-nm(evd-cn) ->
         (let mk-evd-io-cond-nm(ioc,fuid) = evd-cn;
          raise-evd-io-cond(<ioc,fuid,nil>,<loe,cbif-1>)),
      T  ->  (loe(evd-cn))(loe,cbif-1))

   type:  evd-cond-nm B (OE CBIF) =>
```

```
7  raise-comp-cond(comp-cn,enabled,<loe,cbif>)=
         (if enabled
            then
              ((loe(comp-cn))(loe,cbif);
                    /*** assert: nxt. lines interpreted iff        ***/
                    /***         no GOTO out of loe(comp-cn)!      ***/
               if comp-cn∈{FOFL,OFL,SIZE,SUBRG,ZDIV,STRG}
                  then error
                  else I)
            else error)

   type:  comp-cond-nm B (OE CBIF) =>
```

```
8  raise-conv(enabled,<mk-CHAR-STR-VAL(symbol-l),char-pos,fuid>,<loe,cbif>)=
      if enabled
         then
            (let edd = mk-str-edd(CHAR,lsymbol-l,NONVARYING);
             let loc-onsource : alloc(edd,AUTO,<loe,cbif>);
             assign(loc-onsource,mk-CHAR-STR-VAL(symbol-l));
             (trap exit(abn) with
                 (free(loc-onsource);
                  exit(abn));
              let cbif-1 = cbif + ([ONSOURCE → loc-onsource]                    u
                                  [ONCHAR   → integer(char-pos)]                u
                                  if fuid≠nil
                                     then
                                        [ONFILE → mk-CHAR-STR-VAL(file-id(fuid))]
                                     else
                                        [ ]);
              (loe(CONV))(loe,cbif-1);
              (let scurce-str-val : cont(loc-onsource);
               free(loc-cnsource);
               return(scurce-str-val))))
         else
            error

   type:  B (CHAR-STR-VAL intg [uid]) (OE CBIF) => CHAR-STR-VAL
```

```
9  raise-evd-io-cond(<ioc,fuid,str-val>,<loe,cbif0>)=
      (let cbif = cbif0 + [ONFILE → mk-CHAR-STR-VAL(file-id(fuid))]
                        + (if ¬(ONCODE ∈ Dcbif0)
                              then [ONCODE → integer(oncode-val(ioc))]
                              else [ ]);
      let evd-cn = mk-evd-io-cond-nm(ioc,fuid);
      if ioc=KEY
         then
           (let cbif-1  = cbif + [ONKEY → str-val];
            (loe(evd-cn))(loe,cbif-1))
         else
            (loe(evd-cn))(loe,cbif))

   type:   (io-cond-nm uid CHAR-STR-VAL) (OE CBIF) =>
   assert: ¬(ONCODE ∈ Dcbif0) ⊃ cbif0=[ ], the empty map!
```

```
10 oncode-val(evd-cn)=
        /*** returns appropriate NUM ***/

   type:  evd-cond-nm -> NUM



11 cond-comment(arg)=
      /*** prints comments on system output file ***/
      /***          returns normally             ***/

   type: (non-io-cond-nm|evd-io-cond) =>



12 cond-impl-def-act(nm)=
      /*** obvious ***/

   type:  ( SNAP | ERROR | STG )  =>
```

4.2 BIF Value- and Pseudo Variable Location Functions

```
13 eval-cond-bif(cbif-nm,cbif)=
     (cbif-nm = ONCHAR ->
       if ONCHAR∈Dcbif
          then(let pos = s-num(cbif(ONCHAR));
               let loc = comp-loc(cbif(ONSOURCE),<pos,pos>);
               let val : cont(loc);
               return(val))
          else(let val = mk-CHAR-STR-VAL(<blank-SYMBOL>);
               return(val)),
      cbif-nm = ONCODE ∨ cbif-nm = ONCCUNT ->
         (let val = if cbif-nm∈Dcbif then cbif(cbif-nm) else integer(0);
          return(val)),
      cbif-nm = ONSOURCE ->
         (let val : if ONSOURCE∈Dcbif
                       then cont(cbif(cbif-nm))
                       else mk-CHAR-STR-VAL(<>);
          return(val)),
      cbif-nm = ONKEY ∨ cbif-nm= ONFILE ->
         (let val = if cbif-nm ∈ Dcbif
                       then cbif(cbif-nm)
                       else mk-CHAR-STR-VAL(<>);
          return(val)))

   type:  cond-bif-nm CBIF => VAL
```

```
14 eval-l-cond-pv(cpv,< , ,cbif>)=
     (if ¬(ONSOURCE ∈ Dcbif)
         then
            error
         else
            (let l = cbif(ONSOURCE);
             if cpv=ONSOURCE
                then
                   return(l)
                else
                   (let pos = s-num(cbif(ONCHAR));
                    return(ccmp-loc(l,<pos,pos>)))))

   type:  cond-pv (ENV OE CBIF) => LOC
```

4.3 Enablement Status (Context) Functions

```
15 cur-cond-prefs(t) =
      (let <c1,t1> = if t∈T
                     then
                        <{},t>
                     else
                        <im-cprefs(t),im-embr-eubl-proc(t)>;
       let c2 = extract-cprefs(t1);
       merge-cprefs(c2,c1))

   type:  text  ->  cond-pref-set
```

```
16 im-cprefs(t) =
      (let t1 = if is-proc(t) ∨ is-ex-unit(t)
                then
                   t
                else
                   (let t2 = (∪t2)(is-contained(t2,T)                    ∧
                                  (is-proc(t2) ∨ is-ex-unit(t2))         ∧
                                  is-contained(t,t2)                     ∧
                                  ¬(∃ t3) ((is-proc(t3) ∨ is-ex-unit(t3)) ∧
                                           is-contained(t3,t2)           ∧
                                           is-contained(t,t3)            ∧
                                           t2 ≠ t3));
                    if t2=mk-ex-unit( , ,mk-on-st( ,cn1, ))
                       ∧ ¬is-contained(t,cn1)
                       then nil
                       else t2);
       if t1=nil then {} else s-cprefs(t1))

   type:  text  ->  cond-pref-set
```

```
17 extract-cprefs(t) =
      (if (t∈π)
          then
             (let c-default = {mk-cond-pref(cn,ENABLED) |
                                        cn∈{CONV,FOFL,OFL,STRZ,UFL,ZDIV}} ∪
                              {mk-cond-pref(cn,DISABLED) |
                                        cn∈{SUBRG,STRG,SIZE}};
              merge-cprefs(c-default,s-cprefs(t)))
          else
             (let c  = s-cprefs(t),
                  t' = im-embr-eubl-proc(t);
              merge-cprefs(extract-cprefs(t'),c)))

   type:  (ex-unit|proc)  ->  cond-pref-set
   pre:   is-ex-unit(t) ⊃ is-bl(s-prop-st(t))


18 merge-cprefs(c-outer,c-inner) =
      (c-inner ∪ {mk-cond-pref(cn,enab) |
                          mk-cond-pref(cn,enab) ∈ c-outer                ∧
                          ¬(mk-cond-pref(cn,ENABLED) ∈ c-inner)          ∧
                          ¬(mk-cond-pref(cn,DISABLED) ∈ c-inner)})

   type:  cond-pref-set  cond-pref-set  ->  cond-pref-set


19 im-embr-eubl-proc(t) =
       ((∪t')(is-contained(t',π)                          ∧
              (is-proc(t') ∨ is-ex-unit(t'))              ∧
              is-contained(t,t')                          ∧
              is-ex-unit(t') ⊃
                 is-bl(s-prop-st(t'))                     ∧
              ¬(∃ t") (is-contained(t",π)                                ∧
                       (is-proc(t") ∨ is-ex-unit(t"))                    ∧
                       is-ex-unit(t") ⊃
                          is-bl(s-prop-st(t"))                           ∧
                       is-contained(t,t") ∧ is-contained(t",t')          ∧
                       t' ≠ t")))

   type:  text  ->  (ex-unit|proc)
```

```
20 is-enab(comp-cn,cprefs)=
      (mk-cond-pref(comp-cn,ENABLED) ∈ cprefs)

   type:  comp-cond-nm  cond-pref-set  ->  B
```

F5. Expressions

Contents


```
1  eval-comp-expr(t,r-edd,<env,loe,cbif>) =
     let cprefs = cur-cond-prefs(t);
     let val : eval-expr(t,<env,loe,cbif>);
     prom-conv(r-edd,val,<lce,cbif,cprefs>)

   type: expr edd (ENV OE CBIF) => VAL
   post: sdd of val is el-sdd(t)
         edd of result is r-edd
```

```
2  eval-expr(t,env-ex)=
     let <env,loe,cbif> = env-ex,
         cprefs = cur-cond-prefs(t);
     let env-op = <loe,cbif,cprefs>;
     is-const(t) ->
       (let mk-const(dtp,symbl) = t;
        let val1 : symbl-to-val(symbl,is-c-const,env-op);
        let sc-edd = if is-arith(dtp)
                     then dtp
                     else (let mk-str-sdd(tp, ) = dtp;
                           mk-str-edd(tp,ls-val-l(val1),NONVARYING));
        conv(sc-edd,val1,env-op))
     is-nmd-const-ref(t) -> return(env(s-id(t)))
     is-var-ref(t) ->
        (let loc : eval-l-var-ref(t,env-ex);
         cont(loc))
     is-cond-bif-nm(t) -> eval-cond-bif(t,cbif)
     is-non-distr-bif-ref(t) -> eval-non-distr-bif(t,env-ex)
     is-proc-fct-ref(t) ->
       (let <en-ref,arg-l>  = t;
        let <dtp,en-f,loc-l> : eval-proc-ref(en-ref,arg-l,env-ex);
        trap exit(targ) with
          (free-dummy(arg-l,loc-l);
           exit(targ));
        let val : en-f(dtp,loc-l,loe,cbif);
        free-dummy(arg-l,loc-l);
        return(val))
     is-inf-expr(t) v is-pref-expr(t) v is-distr-bif-ref(t) ->
       (let <op,expr-l>  = (cases t :
                            mk-inf-expr(e1,op,e2)    -> <op,<e1,e2>>
                            mk-pref-expr(op,e)       -> <op,<e>>
                            mk-distr-bif-ref(op,argl) -> <op,argl>);
        let sdd-l = <el-sdd(t),<el-sdd(expr-l[i]) | 1≤i≤lexpr-l>>;
        let val-l : <eval-expr(expr-l[i],env-ex) | 1≤i≤lexpr-l>;
        distrib-op(cp,sdd-l,val-l,env-op))

  type: expr (ENV OE CBIF)  => VAL
  post: sdd of val is el-sdd(t)
```

5.1 Distribution.

```
3  distrib-op(op,<r-sdd,el-sdd-l>,val-l,env-op)=
     cases r-sdd :
     mk-array-sdd(r-unit-sdd, ) ->
        (let indices-set = {v-indices(val-l[i]) | 1≤i≤lel-sdd-l ∧
                                                  is-array-sdd(el-sdd-l[i])};
         if (∃ indices1,indices2 ∈ indices-set)(indices1 ≠ indices2) then error;
         let indices = (∪indices)(indices∈indices-set);
         let u-sdd-l = <r-unit-sdd,
                       <( cases el-sdd-l[i]:
                             mk-array-sdd(unit-sdd, ) -> unit-sdd
                             T                        -> el-sdd-l[i])
                                                       | 1≤i≤lel-sdd-l>>;
         let m : [i-l → distrib-op(op,u-sdd-l,
                                  <(if is-array-sdd(el-sdd-l[i])
                                       then comp-val(val-l[i],i-l)
                                       else val-l[i]) | 1≤i≤lel-sdd-l>,
                                  env-op) | i-l ∈ indices];
         return(mk-ARRAY-VAL(m)))
     mk-struct-sdd(r-sdd-l) ->
        (let c-val-l :
           <(let c-sdd-l = <r-sdd-l[j],<( cases el-sdd-l[i]:
                                            mk-struct-sdd(sdd-l) -> sdd-l[j]
                                            T                    -> el-sdd-l[i])
                                            | 1 ≤ i ≤ lel-sdd-l>>;
               let c-val-l = <(if is-struct-sdd(el-sdd-l[i])
                                  then comp-val(val-l[i],j)
                                  else val-l[i]) | 1≤i≤lel-sdd-l>;
               distrib-op(op,c-sdd-l,c-val-l,env-op)) | 1≤j≤lr-sdd-l>;
         return(mk-STRUCT-VAL(c-val-l)))
     mk-sc-sdd(r-sdtp) ->
        (let edd-l = gen-comp-edd(op,el-sdd-l,<if is-STR-VAL(val-l[i])
                                                 then ls-val-l(val-l[i])
                                                 else nil | 1≤i≤lel-sdd-l>);
         let conv-val-l : <conv(edd-l[i],val-l[i],env-op,nil) | 1≤i≤lel-sdd-l>;
         apply-and-conv(op,r-sdtp,conv-val-l,env-op))

  type: (inf-op|pref-op|distr-bif-nm)(sdd sdd*) VAL* (OE CBIF cond-pref-set)=>VAL
  pre: is-array-sdd(r-sdd)  ⇒ (∃i)(is-array-sdd(el-sdd-l[i]))
       is-struct-sdd(r-sdd) ⇒ (∃i)(is-struct-sdd(el-sdd-l[i]))
       is-sc-sdd(r-sdd)     ⇒ (∀i)(is-sc-sdd(el-sdd-l[i]))
       sdd[i] of val-l[i] is el-sdd-l[i]
  post: sdd of val is r-sdd
```

```
4  gen-comp-edd(op,sdd-l,len-l) =
     let sdtp-l = <(let mk-sc-sdd(sdtp) = sdd-l[i];
                    sdtp) | 1≤i≤lsdd-l>;
     is-arith-op(op)   ->
         <let <b,s> = der-bs(Rsdtp-l);
          mk-arith(b,s,conv-prec(sdtp-l[i],b,s)) | 1≤i≤lsdtp-l>
     is-compar-op (op)   ->
        is-non-comp-tp(sdtp-l[1]) ∧ is-non-comp-tp(sdtp-l[2])   ->  sdtp-l
        is-arith(sdtp-l[1]) ∨ is-arith(sdtp-l[2])    ->
           <let <b,s> = der-bs(Rsdtp-l);
            mk-arith(b,s,conv-prec(sdtp-l[i],b,s))) | 1≤i≤lsdtp-l>
        is-str-sdd(sdtp-l[1]) ∧ is-str-sdd(sdtp-l[2])    ->
           <mk-str-edd(der-tp(sdtp-l),max(len-l[1],len-l[2]),NONVARYING)
                                                    | 1≤i≤lsdtp-l>
     is-SUBSTR(op)  ->
           <mk-str-edd(der-tp(<sdtp-l[1]>),
                       if is-arith(sdtp-l[1]) then der-char-len(sdtp-l[1])
                                              else len-l[1],
                       NONVARYING),
            intg-tp(),
            if sdtp-l[3]=nil then nil else intg-tp()>
     is-CHAR(op) ->
           <mk-str-edd(CHAR,
                       if is-arith(sdtp-l[1]) then der-char-len(sdtp-l[1])
                                              else len-l[1],
                       NONVARYING),
            if sdtp-l[2]=nil then nil else intg-tp()>
     is-BIT(op)  ->
           <mk-str-edd(BIT,
                       if is-arith(sdtp-l[1]) then der-bit-len(sdtp-l[1])
                                              else len-l[1],
                       NONVARYING),
            if sdtp-l[2]=nil then nil else intg-tp()>
     is-CAT(op) ∨ is-LENGTH(op)   ->
           <mk-str-edd(der-tp(sdtp-l),
                       if is-arith(sdtp-l[i]) then der-char-len(sdtp-l[i])
                                              else len-l[i],
                       NONVARYING) | 1≤i≤lsdtp-l>
     is-VERIFY(op) ∨ is-INDEX(op) ∨ is-TRANSLATE(op)   ->
           <mk-str-edd(CHAR,
                       if is-arith(sdtp-l[i]) then der-char-len(sdtp-l[i])
                                              else len-l[i],
                       NONVARYING) | 1≤i≤lsdtp-l>
     is-AND(op) ∨ is-OR(op)   ->
```

```
        (let j = max(if is-arith(sdtp-l[1]) then der-bit-len(sdtp-l[1])
                                            else len-l[1],
                     if is-arith(sdtp-l[2]) then der-bit-len(sdtp-l[2])
                                            else len-l[2]);
         <mk-str-edd(BIT,j,NONVARYING) | 1≤i≤2>
    is-BOOL(op) ->
        (let j = max(if is-arith(sdtp-l[1]) then der-bit-len(sdtp-l[1])
                                            else len-l[1],
                     if is-arith(sdtp-l[2]) then der-bit-len(sdtp-l[2])
                                            else len-l[2]);
         <mk-str-edd(BIT,j,NONVARYING),
          mk-str-edd(BIT,j,NONVARYING),
          mk-str-edd(BIT,4,NONVARYING)>)
    is-NOT(op) ->
         mk-str-edd(BIT,if is-arith(sdtp-l[1]) then der-bit-len(sdtp-l[1])
                                               else len-l[1],
                    NONVARYING)
```

type: (inf-op | pref-op | distr-bif-nm) sc-sdd* [intg]* -> sc-edd*

5.2 Operations

```
5  apply-and-conv(op,r-sdtp,val-l,env-op) =
      is-compar-op (op) ->
         (let b = compar(op,val-l);
          let bit = if b then 1-BIT else 0-BIT;
          return(mk-BIT-STR-VAL(<bit>)))
      T  ->
        (let opd-l = <if is-NUM-VAL(val-l[i])
                        then s-num(val-l[i])
                        else (let mk-STR-VAL( )(str) = val-l[i];
                              str)    | 1 ≤ i ≤ lval-l>;
         is-arith-op(op)  ->
           (let num : num-res(op,opd-l,env-op);
            arith-rep(num,r-sdtp,env-op,nil))
         is-string-op(op) ->
            (let str : if is-substr-class(op)
                          then substr-res(op,opd-l[1],topd-l,env-op)
                          else return(string-res(op,opd-l));
             let mk-str-sdtp(tp,maxl) = r-sdtp;
             return(mk-STR-VAL(tp)(str)))
         is-mix-op(op)    ->
            (let num = mix-res(op,opd-l);
             return(integer(num))))

   type:(inf-op | pref-op | distr-bif-nm) sc-comp-tp  SC-VAL*
                                          (OE CBIF cond-pref-set) => SC-VAL
```

```
6  arith-op     =  ADD | SUBT | MULT | DIV | PLUS | MINUS | BIN | DEC | PREC | ABS |
                   FIX | FLOAT | MAX | MIN | MOD | SIGN | FLOOR | CEIL
7  compar-op    =  GT | GE | EQ | NE | LE | LT
8  string-op    =  CAT | substr-class | TRANSLATE | BOOL | NOT | AND | OR
9  mix-op       =  LENGTH | VERIFY | INDEX
10 substr-class =  SUBSTR | CHAR | BIT
```

```
11 num-res(op,num-l,env-op)=
      op∈{BIN,DEC,FLT,FIX,PREC} -> return(num-l[1])
      op∈{MIN,MAX} ->
         (let num = (εv)(∃i∈{1:lnum-l})(v = num-l[i] ∧
                        (∀j∈{1:lnum-l})(op = MAX ⊃ v ≥ num-l[j] ∧
                                       op = MIN ⊃ v ≤ num-l[j]);
          return(num))
      op∈{ADD,SUBT,MULT,DIV,PLUS,MINUS,ABS,MOD,SIGN,FLOOR,CEIL}  ->
         lnum-l = 1 ->
           (let num = num-l[1];
            cases op :
               PLUS  -> return( num )
               MINUS -> return( -num )
               ABS   -> return( abs(num))
               CEIL  -> return( ceil(num))
               FLOOR -> return( floor(num))
               SIGN  -> return( sign(num)))
         lnum-l = 2 ->
            (let <num1,num2> = num-l;
             cases op :
               ADD  -> return( num1 + num2 )
               SUBT -> return( num1 - num2 )
               MULT -> return( num1 * num2 )
               DIV  -> if num2≠0
                         then return(num1/num2)
                         else (let <loe,cbif,cprefs> = env-op;
                                   raise-cond(ZDIV,is-enab(ZDIV,cprefs),<loe,cbif>))
               MOD  -> if num2≠0
                         then return(num1 - num2*(floor(num1/num2)))
                         else return(num1))
```

type: arith-op NUM*(OE CBIF cond-pref-set) => NUM

BASIS-11: optionally raises FOFL | OFL | UFL before the evaluation of the mathematical value (ref: 9.4.4.45, step 2; case 2.1)

```
12 arith-rep(num,rdd,env-op,size)=
      let <loe,cbif,cprefs> = env-op;
      let mk-arith(base,scale,mk-prec(nod,scale-f)) = rdd;
      let b = (if base = DEC then 10 else 2),
      let <cond,p> = if size = SIZE then <SIZE,nod> else <FOFL,nn(base,scale)>;
      scale = FIX ->
        (let num1 = (b↑ - scale-f) * floor(abs(num) * b↑scale-f) * sign(num);
         if abs(num1) < b↑(p - scale-f)
            then return(mk-NUM-VAL(rdd,num1))
            else raise-cond(cond,is-enab(cond,cprefs),<loe,cbif>))
      scale = FLT ->
         (let num1 = if num ∈ intg ∧ abs(num) < b ↑ nn(b,FLT)
                        then num
                        else approx(num,rdd);
          return(mk-NUM-VAL(rdd,num1)))
```

type: NUM arith (OE CBIF cond-pref-set) [SIZE] => NUM-VAL
BASIS-11: conversion to FIX is implementation defined(9.5.1.2,case 3.2)

```
13 approx(x,dd)=
      /* impl. def. */  -> raise-cond(OFL,is-enab(OFL,cprefs),<loe,cbif>)
      /* impl. def. */  -> (raise-cond(UFL,is-enab(UFL,cprefs),<loe,cbif>);
                            return(0))
      /* impl. def. */  -> return( /* impl. def. */ )
```

type: NUM arith => NUM
note: test and result deperds only on x and dd.

```
14 compar(op,opd-1)=
     is-ENTRY-VAL-list(opd-1) ∨ is-LAB-VAL-list(opd-1) ∨ is-FILE-VAL-list(opd-1)->
        (let <opd1,opd2> = opd-1;
         cases op :
           EQ  ->  opd1 = opd2
           NE  ->  opd1 ≠ opd2)
     is-PTR-VAL-list(opd-1)  ->
       (let <opd1,opd2> = opd-1;
        if opd1 ≠ NULL-PTR ∧ (∀ l∈locs(S))(addr(l) ≠ opd1)  ∨
           opd2 ≠ NULL-PTR ∧ (∀ l∈locs(S))(addr(l) ≠ opd2) then error;
        cases op :
           EQ  ->  opd1 = opd2
           NE  ->  opd1 ≠ opd2)
     is-NUM-VAL-list(opd-1)  ->
        (let mk-NUM-VAL( ,num1) = opd-1[1],
             mk-NUM-VAL( ,num2) = opd-1[2];
         cases op:
           EQ  ->  num1 = num2
           NE  ->  num1 ≠ num2
           LT  ->  num1 < num2
           LE  ->  num1 ≤ num2
           GE  ->  num1 ≥ num2
           GT  ->  num1 > num2)
    is-STR-VAL-list(opd-1)  ->
      (let mk-STR-VAL( )(str1) = opd-1[1],
           mk-STR-VAL( )(str2) = opd-1[2];
       let i = first({k | 1≤k≤lstr1 ∧ str1[k]≠str2[k]});
       is-BIT-VAL-list(<str1,str2>)  ->
         (i = 0             ->  op = EQ ∨ op = GE ∨ op = LE
          str1[i] = 0-BIT   ->  op = NE ∨ op = LE ∨ op = LT
          str1[i] = 1-BIT   ->  op = NE ∨ op = GE ∨ op = GT)
       is-CHAR-VAL-list(<str1,str2>) ->
         (i = 0 -> op = EQ ∨ op = GE ∨ op = LE
          i ≠ 0 -> (let k1=first({k | 1≤k≤lCOLL-SEQU() ∧ str1[i]=COLL-SEQU()[k]}),
                        k2=first({k | 1≤k≤lCOLL-SEQU() ∧ str2[i]=COLL-SEQU()[k]});
                    k1 < k2  ->  op = LE ∨ op = LT ∨ op = NE
                    k1 > k2  ->  op = GE ∨ op = GT ∨ op = NE)))

type: compar-op  SC-VAL² -> B
```

```
15 substr-res(op,str,num-l,env-op) =
      op = CHAR ∨ op = BIT ->
         (let num = if num-l = <> then lstr else num-l[1];
          if num < 0 then error;
          return(<(if i ≤ lstr
                     then str[i]
                     else if op = CHAR then blank-SYMBOL else 0-BIT) | 1≤i≤num>))
      op = SUBSTR ->
         (let ind = num-l[1];
          let length = if lnum-l = 1 then lstr-ind+1 else num-l[2];
          if 0 ≤ ind - 1 ≤ ind + length - 1 ≤ lstr
            then return(<str[i] | ind ≤ i ≤ ind + length - 1>)
            else(let <loe,cbif,cprefs> = env-op;
                 raise-cond(STRG,is-enab(STRG,cprefs),<loe,cbif>)))

   type: substr-class CHAR-VAL*  NUM* (OE CBIF cond-pref-set) => CHAR-VAL* |
         substr-class BIT-VAL*   NUM* (OE CBIF cond-pref-set) => BIT-VAL*.
```

```
16 string-res(op,str-1)=
      cases op:
         NOT  ->  <(if str-1[1,i] = 1-BIT then 0-BIT else 1-BIT) | 1≤i≤lstr-1[1]>
         CAT  ->  conc(str-1)
         AND  ->  <(if str-1[1,i] = 1-BIT ∧ str-1[2,i] = 1-BIT
                     then 1-BIT
                     else 0-BIT) | 1≤i≤lstr-1[1]>
         OR   ->  <(if str-1[1,i] = 1-BIT ∨ str-1[2,i] = 1-BIT
                     then 1-BIT
                     else 0-BIT) | 1≤i≤lstr-1[1]>
         BOOL ->
           (let <str1,str2,str3> = str-1;
            <str3[(if str1[i] = 1-BIT then 2 else 0)+(if str2[i] = 1-BIT
                                                      then 1
                                                      else 0)+1] | 1≤i≤lstr1>)
         TRANSLATE ->
            (let str-11 = if lstr-1 = 2 then str-1 ^ <COLL-SEQU()> else str-1;
             let <str1,str2,test-str> = str-11;
             <(let j = first({k | 1≤i≤ltest-str ∧ str1[i] = test-str[k]})
               if j = 0 then str1[i] else str2[k]) | 1≤i≤lstr1>)

   type: string-op (CHAR-VAL*)* -> CHAR-VAL*  |
         string-op (BIT-VAL*)*  -> BIT-VAL*
   pre: for AND, OR lengths will be equal,
        for BOOL lstr1 = lstr2 ∧ lstr3 = 4.



17 mix-res(op,str-1)=
      op = LENGTH -> lstr-1[1]
      op = VERIFY ∨ op = INDEX ->
         (let str1 = str-1[1],
              str2 = str-1[2];
          first({k | 1≤k≤lstr1 ∧ (∀j∈{1:lstr2})
                                  (if op = VERIFY
                                          then str1[k] ≠ str2[j]
                                          else str1[k+j-1] = str2[j])}))

   type: mix-op ((CHAR-VAL*)* | (BIT-VAL*)*) -> intg
```

```
18 eval-non-distr-bif(t,env-ex)=
      let mk-non-distr-bif-ref(nm,arg-l) = t;
      let <,loe,cbif> = env-ex;
      let env-op = <loe,cbif,cur-cond-prefs(t)>;
      nm = DIM v nm = HBOUND v nm = LBOUND ->
         (let <ag,d-expr>  = arg-l;
          let j : s-num(eval-comp-expr(d-expr,intg-tp(),env-ex));
          if 1≤j≤ls-bpl(el-sdd(ag))
            then (let ag-loc : eval-l-var-ref(ag,env-ex);
                  let mk-ebp(lbd,ubd) = s-bpl(l-edd(ag-loc))[j];
                  let num = (nm = DIM     -> ubd - lbd + 1
                             nm = HBOUND  -> ubd
                             nm = LBOUND  -> lbd );
                  return(integer(num)))
            else error)
      nm = NULL -> return(NULL-PTR)
      nm = DATE ->  return(mk-CHAR-STR-VAL(/* impl. def. */))
      nm = TIME ->  return(mk-CHAR-STR-VAL(/* impl. def. */))
      nm = ADDR ->
            (let ag = arg-l[1];
             let ag-loc : eval-l-var-ref(ag,env-ex);
             if is-ccnn(ag-loc)
                then return(addr(ag-loc))
                else error)
      nm = STR ->
         (let ag = arg-l[1];
          let mk-sc-sdd(mk-str-sdd(tp-t,l)) = el-sdd(t);
          let ag-val : eval-expr(ag,env-ex);
          let str-ag : conv-to-str-ag(tp-t,el-sdd(ag),ag-val,env-op);
          let str = concat(str-ag,el-sdd(ag));
          return(mk-STR-VAL(tp-t)(str)))
      nm = COLLATE ->  return(mk-CHAR-STR-VAL(COLL-SEQU()))

   type: non-distr-bif-ref (ENV OE CBIF) => SC-VAL
```

```
19 conv-to-str-ag(tp-t,sdd-s,val,env-op)=
      cases sdd-s:
         mk-struct-sdd(sdd-l)   ->
            return(mk-STRUCT-VAL(<conv-to-str-ag(tp-t,sdd-s[i],comp-val(val,i),
                                                   env-op) | 1 ≤ i ≤ lsdd-l>))
         mk-array-sdd(unit,bp) ->
            return(mk-ARRAY-VAL([indl → conv-to-str-ag(tp-t,unit,comp-val(val,indl),
                                                 env-op) | indl ∈ v-indices(val)]))
         mk-sc-sdd(sdtp)        ->
            let edd = mk-str-edd(tp-t,if is-arith(sdtp)
                                      then der-char-len(sdtp)
                                      else ls-val-l(val) , NONVARYING);
            conv(edd,val,env-cp)

   type: str-tp sdd VAL (OE CBIF cond-pref-set) => VAL


20 concat(str-ag,sdd)=
      cases sdd :
         mk-array-sdd(unit-sdd, ) ->
            (let list = v-ordered-indices(str-ag);
             <concat(comp-val(str-ag,list[i]),unit-sdd) | 1≤i≤llist>)
         mk-struct-sdd(sdd)        ->
            conc(<concat(comp-val(str-ag,i),sdd[i]) | 1≤i≤lsdd>)
         mk-sc-sdd(sdtp)           ->  s-val-l(str-ag)

   type: VAL sdd -> (CHAR-VAL* | BIT-VAL*)
```

5.3 Conversions.

```
21 prom-conv(t-edd,v,env-op)=
      cases t-edd:
      mk-struct-edd(t-edd-l) ->
         (let val-l:(if is-STRUCT-VAL(v)
                     then <prom-conv(t-edd-l[i],comp-val(v,i),env-op) |1≤i≤lt-edd-l>
                     else <prom-conv(t-edd-l[i],v,env-op) | 1≤i≤lt-edd-l>);
          return(mk-STRUCT-VAL(val-l)))
      mk-array-edd(unit-edd,t-ebp) ->
         (let m:(if is-ARRAY-VAL(v)
                   then (if v-indices(v) = indices(t-ebp)
                           then [ind-l → prom-conv(unit-edd,comp-val(v,ind-l),env-op)
                                                          | ind-l∈indices(t-ebp)]
                           else error)
                   else [ind-l → prom-conv(unit-edd,v,env-op)
                                                  | ind-l ∈ indices(t-ebp)]);
          return(mk-ARRAY-VAL(m)))
      T  ->   conv(t-edd,v,env-op)
```

type: edd VAL (OE CBIF cond-pref-set) => VAL
pre: shape of v must be promotable to shape of t-edd
post: val ∈ values(t-edd)

```
22 conv(dd,val,env-op)=
      is-non-comp-tp(dd) -> return(val)
      is-arith(dd) ->
         (let num :
          (cases val:
            mk-NUM-VAL(tp,num)      ->  num
            mk-CHAR-STR-VAL(symbl) ->  s-num(symbl-to-val(symbl,is-c-num-str,env-op))
            mk-BIT-STR-VAL(bit-l)  ->  digitl-to-intg(bit-l,2));
          arith-rep(num,dd,env-op,SIZE))
      is-str-edd(dd) ->
         (s-tp(dd) = BIT  -> conv-to-bit(dd,val,env-op)
          s-tp(dd) = CHAR -> conv-to-char(dd,val,env-op))
```

type: sc-edd SC-VAL (OE CBIF cond-pref-set) => SC-VAL

```
23 conv-to-bit(e-str,val,env-op)=
      let symbl:
         (is-STR-VAL(val) ->
             cases val :
               mk-BIT-STR-VAL(bit-list)   -> bit-list
               mk-CHAR-STR-VAL(symbl)     ->
                  (let symbl1 : test-and-correct-bit-str(symbl,env-op);
                   <(if symbl1[k] = 0-SYMBOL then 0-BIT else 1-BIT) | 1≤k≤lsymbl1>)
          is-NUM-VAL(val) ->
            (let p = s-ncd(conv-prec(s-tp(val),BIN,FIX));
             let tdd = mk-arith(BIN,FIX,mk-prec(s-nod(p),0));
             let conv-val : ccnv(tdd,val,env-op,);
             let num = abs(s-num(conv-val));
               <(if mod(floor(num / 2↑(p-i)),2) = 0
                     then 0-BIT
                     else 1-BIT) | 1≤i≤p>));
      let symbl':
         if lsymbl > s-maxl(e-str)
            then (let <oe,cbif,cprefs> = env-op;
                  raise-cond(STRZ,is-enab(STRZ,cprefs),<oe,cbif>);
                  <symbl[k] | 1 ≤ k ≤ ls-maxl(e-str)>)
            else symbl;
      let adj-symbl : if s-vary(e-str) = NONVARYING
                        then
                          <(if k ≤ lsymbl'
                              then symbl'[k]
                              else 0-BIT) | 1≤k≤s-maxl(e-str)>
                        else symbl';
      return(mk-STR-VAL(BIT)(adj-symbl))

  type: str-edd VAL (OE CBIF cond-pref-set)  => STR-VAL
```

```
24 test-and-correct-bit-str(symbl,env-op) =
      let i = first({ k | ¬(symbl[k] ∈ {0-SYMBOL,1-SYMBOL)});
      if i = 0
         then return(symbl)
         else
            (let <oe,cbif,cprefs> = env-op;
             let mk-CHAR-STR-VAL(symbl') = raise-conv(is-enab(CONV,cprefs),
                                   <mk-CHAR-STR-VAL(symbl),i,nil>,<oe,cbif>);
             test-and-correct-bit-str(symbl',env-op))
```

type: symbol* (OE CBIF cond-pref-set) => (0-SYMBOL | 1-SYMBOL)*

```
25 conv-to-char(e-str,val,env-op)=
      let symbl =
        cases val :
          mk-BIT-STR-VAL(bit-list)  ->
            <(if bit-list[k]=0-BIT then 0-SYMBOL else 1-SYMBOL) | 1 ≤ k ≤ lbit-list>
          mk-CHAR-STR-VAL(symbl)    -> symbl
          mk-NUM-VAL(mk-arith(b,s,p), ) ->
            (let dec-prec = (if b = BIN then conv-prec(s-tp(val),DEC,s) else p);
             let dd = mk-arith(DEC, s,dec-prec);
             let num = s-num(conv(dd,val,env-op));
             let mk-prec(dp,dq) = dec-prec;
             let symbl =
               (if s = FLT
                  then (let l = (Ue)(10↑e ≤ abs(num) < 10↑(e+1));
                        (Usymbl∈term-str(c-prop-num-str))(∃val1∈NUM-VAL)
                          (let mk-NUM-VAL(mk-arith(b1,s1,mk-prec(dp1,)),num1)=val1;
                           abs(num1 - num) ≤ 5*(10↑(e-p)) ∧
                           <b1,s1,dp1> = <b,s,dp> ∧
                           val1 = symbl-to-val(symbl,is-c-prop-num-str,env-op) ∧
                           normalized(symbl,dp,num1)))
                  else (Usymbl∈term-str(c-prop-num-str))
                          (val = symbl-to-val(symbl,is-c-prop-num-str,env-op) ∧
                           correct-prec(symbl,dp,dq))));
      let symbl':
         if lsymbl > s-maxl(e-str)
            then (let <ce,cbif,cprefs> =env-op;
                  raise-cond(STRZ,is-enab(STRZ,cprefs),<oe,cbif>);
                  <symbl[k] | 1 ≤ k ≤ ls-maxl(e-str)>)
            else symbl;
      let adj-symbl : if s-vary(e-str)=NONVARYING
                        then
                          <(if k ≤ lsymbl'
                              then symbl'[k]
                              else BLANK) | 1≤k≤s-maxl(e-str)>)
                        else symbl';
      return(mk-STR-VAL(CHAR)(adj-symbl))

   type: str-edd VAL (OE CBIF cond-pref-set)  => VAL
```

5.4 Translation of Symbol-Lists

5.4.1 Concrete Syntax of Constants.

```
26 c-const           =  c-char-str | c-bit-str | c-num-str
27 c-char-str        :: '-SYMBOL c-string-spec '-SYMBOL
28 c-string-spec     :: ((any symbol except '-SYMBOL) | '-SYMBOL '-SYMBOL)*
29 c-bit-str         :: '-SYMBOL bin-digit* '-SYMBOL B-SYMBOL
30 bin-digit         =  0-SYMBOL | 1-SYMBOL
31 c-num-str         =  c-blanks | c-prop-num-str
32 c-blanks          =  BLANK+
33 c-prop-num-str    :: [c-blanks] [c-sign] c-arith-const [c-blanks]
34 c-sign            =  +-SYMBOL | --SYMBOL
35 c-arith-const     =  c-bin-const | c-dec-const
36 c-bin-const       :: c-bin-num [c-scale] B-SYMBOL
37 c-bin-num         =  c-bin-intg | c-bin-rat-num
38 c-bin-rat-num     :: bin-digit* .-SYMBOL bin-digit+
39 c-bin-intg        :: bin-digit+ [.-SYMBOL]
40 c-scale           :: c-scale-type [c-sign] digit+
41 c-dec-const       :: c-dec-num [c-scale]
42 c-dec-num         =  c-intg | c-dec-rat-num
43 c-dec-rat-num     :: digit* .-SYMBOL digit+
44 c-intg            :: digit+ [.-SYMBOL]
45 c-scale-type      =  E-SYMBOL | F-SYMBOL
```

5.4.2 Parsing and Translation of Symbol-Lists.

```
46 symbl-to-val(symbl,pred,env-op) =
      let <oe,cbif,cprefs> = env-op;
      let const: parse(test-and-correct(symbl,pred,env-op));
      is-c-char-str(const) -> basic-char-val(const)
      is-c-bit-str(const)  -> basic-bit-val(const)
      is-c-num-str(const)  -> basic-num-val(const)

   type: symbol* (c-const -> B) (OE CBIF cond-pref-set) => VAL
   post: pred = is-c-const ⊃ is-SC-VAL(val)
         pred = is-c-num-str ⊃ is-NUM-VAL(val)
```

```
47 test-and-correct(symbl,pred,<oe,cbif,cprefs>) =
      let n = wrong-pos(symbl,pred);
      if n ≠ 0
         then (let mk-CHAR-STR-VAL(new-symbl) : raise-conv(is-enab(CONV,cprefs),
                                    <mk-CHAR-STR-VAL(symbl),n,nil>,<oe,cbif>);
               test-and-correct(new-symbl,pred,<oe,cbif,cprefs>))
         else return(symbl)

   type: symbol* (c-const -> B) (OE CBIF cond-pref-set) => symbol*


48 wrong-pos(symbl,pred) =
      first({i | (1 ≤ i ≤ lsymbl ∧ (∀ c ∈ c-const, l ∈ symbol-list)
                                    (pred(c) ⊃ term-str(c) ≠ <symbl[k] | 1≤k≤i>^l)) ∨
                ( i = lsymbl ∧(∀ c)(pred(c) ∧ term-str(c) ≠ symbl))})

   type: symbol* (c-const -> B) -> intg


49 parse(Symbl) =
      (εc-const)(is-c-const(c-const) ∧ term-str(c-const) = symbl)

   type: symbol* -> c-const
```

```
50 term-str(c)=
      is-c-char-str(c) ->
         (let mk-c-char-str(qu,str-spec,unqu) = c;
          <qu> ^ conc(<(if str-spec[k] = <'-SYMBOL,'-SYMBOL>
                          then <'-SYMBOL,'-SYMBOL>
                          else <str-spec[k]>) | 1 ≤ k ≤ lstr-spec>) ^ <unqu>)
      is-c-bit-str(c) ->
         (let mk-c-bit-str(qu,bit-list,unqu,b) = c;
          <qu>^bit-list^<unqu,b>)
      is-c-num-str(c) ->
         if is-c-blanks(c)
            then c
            else
             (let mk-c-prop-num-str(bl1,sign,arith-const,bl2) = c;
              bl1^
              (if sign = nil then <> else <sign>)^
              (if is-c-bin-const(arith-const)
                  then
                     (let mk-c-bin-const(bin-num,scale,b-symb) = arith-const;
                      if is-c-bin-rat-num(bin-num)
                        then
                          (let mk-c-bin-rat-num(intg-part,point,fract-part)=bin-num;
                           intg-part^<point>^fract-part)
                        else
                          (let mk-bin-intg(intg-part,point);
                           intg-part^(if point ≠ nil then <point> else <>))^
                      if scale = nil
                        then <>
                        else
                          (let mk-c-scale(scale-type,sign,digitl) = scale;
                           <scale-type>^(if sign = nil then <> else <sign>)^digitl)^
                     <b-symb>)
                  else
                     (let mk-c-dec-const(dec-num,scale) = arith-const;
                      if is-c-dec-rat-num(dec-num)
                        then
                          (let mk-c-dec-rat-num(intg-part,point,fract-part)=dec-num;
                           intg-part^<point>^fract-part)
                        else
                          (let mk-c-intg(intg-part,point) = dec-num;
                           intg-part^(if point = nil then <> else <point>))^
                     if scale = nil
                        then <>
                        else
```

```
                        (let mk-c-scale(scale-type,sign,digitl) = scale;
                         <scale-type>^(if sign=nil then <> else <sign>)^digitl)))^
              bl2)
```

type: c-const -> symbol*

```
51 basic-char-val(c)=
      let mk-c-char-str( ,symbl, ) = c;
      let new-symbl = <if symbl[k] = <'-SYMBOL,'-SYMBOL>
                        then '-SYMBOL
                        else symbl[k] | 1≤k≤lsymbl>;
      mk-STR-VAL(CHAR)(new-symbl)
```

type: c-char-str -> STR-VAL

```
52 basic-bit-val(c)=
      let mk-c-bit-str( ,symbl, , ) = c;
      let bit-list = <(if symbl[k]=0-SYMBOL then 0-BIT else 1-BIT) | 1 ≤ k ≤ lsymbl>;
      mk-STR-VAL(BIT)(bit-list)
```

type: c-bit-str -> STR-VAL

```
53 basic-num-val(c)=
      if is-c-blanks(c)
         then mk-NUM-VAL(mk-arith(DEC,FIX,mk-prec(1,0)),0)
         else (let mk-prop-num-str( ,sign,const, ) = c;
               let s = if sign = --SYMBOL then -1 else +1;
               let abs-val = if is-c-bin-const(const) then basic-bin-val(const)
                                                      else basic-dec-val(const);
               let mk-NUM-VAL(mk-arith(b,s,p,num)) = abs-val;
               mk-NUM-VAL(mk-arith(b,s,p),s * num))
```

type: c-num-str -> NUM-VAL

```
54 basic-bin-val(bc)=
      let mk-c-bin-const(num,scale, ) = bc;
      let <i,f> = (if is-c-bin-intg(num)
                     then <s-bin-digit-list(num),<>>
                     else <s-bin-digit-list-1(num),s-bin-digit-list-2(num)>);
      let <sign,exp> = (if scale = nil
                          then <nil,<>>
                          else <s-c-sign(scale),s-digit-list(scale)>);
      let num1 = num-val(i,f,sign,exp,2);
      let <sc,prec> = (if scale = nil v s-c-scale-type(scale) = F-SYMBOL
                       then
                          (let s = if sign = --SYMBOL then -1 else 1,
                           let scale-val = if scale = nil
                                           then 0
                                           else num-val(exp,<>,nil,<>,10);
                           <FIX,mk-prec(li + lf,lf + s * scale-val)>)
                       else <FLT,mk-prec(li + lf,nil)>);
      mk-NUM-VAL(mk-arith(BIN,sc,prec),num1)

   type: c-bin-const -> NUM-VAL



55 basic-dec-val(dc)=
      /* analog to basic-bin-val */

   type: c-dec-const -> NUM-VAL



56 digitl-to-intg(dl,b)=
      dl = <> -> 0
      T  ->   (cases hdl:
               0-SYMBOL -> 0, 0-BIT -> 0,
               1-SYMBOL -> 1, 1-BIT -> 1,
               2-SYMBOL -> 2,
               .
               .
               .
               9-SYMBOL -> 9) * b↑(ldl-1) + digitl-to-intg(tdl,b))

   type: (symbol* | BIT-VAL*)(2 | 10) -> NUM
```

```
57 normalized(symbl,p,num)=
      lsymbl = der-char-len(DEC,FLT,mk-prec(p,nil)) ∧
      (∀i ∈ {2} ∪ {4:p+2} ∪ {p+5:p+SIZE-EXP+4})(is-digit(symbl[i]))
      ∧ is-c-sign(symbl[p+4]) ∧
      (if num = 0 then symbl[2] = 0-SYMBOL else symbl[2] ≠ 0-SYMBOL)

   type: symbol* intg NUM -> B
   note: SIZE-EXP denotes the impl-def length of the exponent-field.
```

```
58 correct-prec(symbl,p,q)=
      lsymbl = der-char-len(mk-arith(DEC,FIX,mk-prec(p,q)) ∧
      (∀i∈ {2:max(1,p-q-2)})(symbl[i] = 0-SYMBOL ⊃
      (∃j∈ {1:i-1})(is-digit(symbl[j]) ∧ symbl[j] ≠ 0-SYMBOL)) ∧
      (0<q≤p      ->  symbl[p+3-q] = .-SYMBOL
       q = 0      ->  is-digit(symbl[lsymbl])
       q<0 ∨ p<q  ->  is-c-sign(symbl[p+3]) ∧
       digitl-to-intg(<symbl[k] | p+4≤k≤lsymbl>,10) = abs(q))

   type: symbol* intg intg -> B
```

```
59 num-val(i,f,s,e,b)=
      let s' = if s = --SYMBOL then -1 else +1 ;
      (digitl-to-intg(i,b) +
       digitl-to-intg(f,b) * b ↑ (-lf * b ↑ s' * digitl-to-intg(e,10))

   type: symbol* symbol* [+-SYMBOL | --SYMBOL] symbol* intg  -> NUM
```

5.5 Auxiliary Functions.

```
60 integer(i)=
      mk-NUM-VAL(intg-tp(),i)

   type: intg -> NUM-VAL
```

```
61 first(intg-set)=
      if intg-set = {} then 0 else (∪i ∈ intg-set)(∀j ∈ intg-set)(i ≤ j)
```

type: intg-set -> intg

```
62 abs(x)=
      if x ≥ 0 then x else -x
```

type: NUM -> NUM

```
63 floor(x)=
      x - mod(x,1)
```

type: NUM -> intg

```
64 ceil(x)=
      x + mod(-x,1)
```

type: NUM -> intg

```
65 intg-tp()=
      mk-arith(BIN,FIX,mk-prec( /* impl. def. */,0))
```

type: <> -> arith
note: the number of digits is not necessarily the same for all activations of intg-tp.

```
66 COLL-SEQU() =   /* impl def */
```

type: -> CHAR-VAL*

```
67 sign(x) =
      x > 0 -> 1
      x = 0 -> 0
      x < 0 -> -1
```

type: NUM -> (-1 | 0 | 1)

```
68 der-tp(sdd-l) =
      (∀ i ∈ {1:lsdd-l})(sdd-l[i] = mk-str-sdd(BIT,*))  -> BIT
      T  ->  CHAR
```

type: sc-sdd* -> str-tp

```
69 der-char-len(arith) =
      let mk-arith(b,s,prec) = arith;
      let mk-prec(p,q) = conv-prec(arith,DEC,s);
      s = FIX  ->
         if  0 ≤ q ≤ p
            then p + 3
            else
               (let n = (∪n ∈ intg)(10↑n ≤ abs(q) < 10↑(n+1));
                p + n + 3 )
      s = FLT  ->  p+SIZE-EXP+4
```

type: arith -> intg

note: SIZE-EXP denotes the impl-def length of the exponent-field.

```
70 der-bit-len(arith) =
      let mk-prec(p,q) = conv-prec(arith,BIN,FIX);
      p
```

type: arith -> intg

THIS PAGE HAS BEEN INTENTIONALLY LEFT BLANK.

F6. Input/Output

```
1  int-io-st(st,env-st) :
     (is-open-st(st)       ->   /***   int-open-st(st,env-st),        ***/
      is-close-st(st)      ->   /***   int-close-st(st,env-st),       ***/
      is-get-st(st)        ->   /***   int-get-st(st,env-st),         ***/
      is-put-st(st)        ->   /***   int-put-st(st,env-st),         ***/
      is-read-st(st)       ->   /***   int-read-st(st,env-st),        ***/
      is-write-st(st)      ->   /***   int-write-st(st,env-st),       ***/
      is-locate-st(st)     ->   /***   int-locate-st(st,env-st),      ***/
      is-rewrite-st(st)    ->   /***   int-rewrite-st(st,env-st),     ***/
      is-delete-st(st)     ->   /***   int-delete-st(st,env-st))      ***/

   type:  io-st (ENV OE CBIF) =>


2  put-page(uid) :
     /***    output-stream-item(PGMRK,uid)    ***/

   type:  uid =>
```

THIS PAGE HAS BEEN INTENTIONALLY LEFT BLANK.