THE FORMAL DEFINITION OF A PARAMETER PASSING LANGUAGE *

by

RICHARD E. FAIRLEY
Department of Computer Science
University of Colorado
Boulder, CO 80302

REPORT #CU-CS-010-72 December, 1972

* This work supported by NSF Grant GJ-660.

THE FORMAL DEFINITION OF A PARAMETER PASSING LANGUAGE

by

Richard E. Fairley *

ABSTRACT:

The formal definition of the syntax and semantics of A Parameter Passing Language (APPL) is presented. APPL is a simple, nested structure language which permits the communication of parameters between a main program and a subroutine using call by reference, call by value, or call by name. The methodology and techniques of the Vienna Definition Language are used in the formalization of APPL. The syntactic definition of APPL consists of phrase structure grammars for concrete and abstract representations of valid APPL programs. The semantic definition of APPL is in terms of an abstract machine which interprets abstract programs. The machine specification consists of the machine state components and a transition function that maps states into successor states. The semantics of various parameter passing mechanisms is thus formalized as the sequence of machine states assumed by the abstract machine during the interpretation of abstract programs.

* This work supported by NSF Grant GJ660.

I. *INTRODUCTION*

The formal definition of the syntax and semantics of A Parameter Passing Language (APPL) is presented in this paper. APPL is a simple, nested structure language which permits the communication of parameters between a main program and a subroutine using call by value, call by reference, or call by name. The definition of APPL thus formalizes the characteristics of various parameter passing mechanisms. The definition will also provide a basis for verifying the correctness of APPL implementations in the future.

The methodology and techniques of the Vienna Definition Language (VDL) are used in the formalization of APPL. In the VDL methodology, a distinction is made between the "concrete program", which is the source string representation of the program as written by the programmer; and the "abstract program", which is a normalized representation of the program used in the formal definition of semantics.

The syntactic definition of APPL in VDL consists of specifications for both concrete and abstract representations of valid APPL programs. The specifications are in terms of phrase structure grammars; the grammar for concrete programs is called the "concrete syntax" of APPL, and the grammar for abstract programs is called the "abstract syntax" of APPL. The correspondence between concrete and abstract programs is established by a translator which accepts concrete programs and produces abstract programs.

The semantic definition of APPL is in terms of an abstract machine which interprets abstract programs. The abstract machine is characterized

by internal states, and a transition function that maps states into successor states. The semantics of APPL is thus formalized as the sequence of machine states assumed by the abstract machine during the interpretation of abstract programs.

The formal definition of APPL consists of five components:

1. The concrete syntax
2. The abstract syntax
3. The abstract machine states
4. The transition function
5. The translator

In this paper, the concrete syntax of APPL is specified in the Extended Backus Normal Form (see LLS 68). The abstract syntax, abstract machine states, and the transition function are all specified in the VDL meta-language. The necessary background in VDL can be obtained by reading WEG 72, and LLS 68. The translation of APPL programs from concrete form to abstract form is not specified in this paper.

Before proceeding with the formal definition of APPL, an informal description of the language will be presented.

II. *INFORMAL DESCRIPTION OF APPL*

A Parameter Passing Language (APPL) is a simple, nested structure programming language which permits the communication of parameters between a main program and a subroutine using call by reference, call by value, or call by name.

Typical APPL programs are illustrated by the following example:

```
int  X,I,A(1), A(3), B;
             ⎧ref ⎫
proc  P(A,B);⎨val ⎬A,B;
             ⎩name⎭
      X ← A;
      A ← B;
      B ← X;
      end;
I ← 1;
A(I) ← 3;
P(I,A(I));
end;
```

The declaration part of this program consists of integer declarations and a procedure declaration. The procedure body consists of three assignment statements which have the effect of interchanging the values of the formal parameters, A and B. X is a global variable in the procedure, and is local to the main program. The memory cells to which formal parameters A and B refer will depend on which of <u>ref</u> (call by reference), <u>val</u> (call by value), or <u>name</u> (call by name) is used in the procedure declaration.

The body of the main program initializes I to "1", and A(1) to "3", followed by a call to the procedure with arguments I and A(I).

The interpretation of a procedure call is specified by the copy rule; i.e., the body of the procedure is to be substituted in place of the procedure call, and parameters are renamed to avoid naming conflicts. If the actual arguments are called by <u>ref</u>, the formal parameters are replaced by the names of the actual arguments, evaluated at the time of the procedure call; provided the evaluation of the argument results in a name. If the evaluation yields a value, rather than a name (as

in the case of arithmetic expressions passed by ref), the value is assigned to the corresponding formal parameter and the parameter is used as an identifier in the body of the procedure.

The procedure call in the given program expands as follows, when A and B are ref parameters:

```
P(I,A(I)); → P(I,A(1)); →    X ← I;
                             I ← A(1);
                          A(1) ← X;
```

Interpretation of the three assignment statements results in interchanging the values of I and A(1).

If the arguments are called by val, the corresponding formal parameters are initialized to the values of the arguments at the time of call and treated as identifiers. Parameters are renamed as necessary to avoid naming conflicts. The procedure call in the example program expands as follows, when A and B are val parameters:

```
P(I,A(I)); → P(I,A(1)); →    A ← I;
                            BB ← A(1);
                             X ← A;
                             A ← BB;
                            BB ← X;
```

Note that formal parameter B is renamed BB to avoid a naming conflict with the integer variable B known in the main program. Interpretation of the five assignment statements interchanges the values of A and BB, but does not alter the values of I and A(1).

If the arguments are called by name, the formal parameters are replaced by the text of the corresponding argument. This, of course,

precludes the passing of arithmetic expressions to formal parameters which appear on the left hand side of assignment statements. The procedure call in the example program expands as follows, when A and B are <u>name</u> parameters:

```
P(I,A(I)); →    X ← I;
                I ← A(I);
             A(I) ← X;
```

Interpretation of the procedure body results in assignment of "3" to I, and "1" to A(3), while A(1) retains the value "3".

III. THE CONCRETE DEFINITION OF APPL

The production rules for the concrete syntax of APPL are presented in Table I. The meta-language used to specify the grammar of Concrete APPL is the Extended Backus Normal Form. The head of the language is the symbol "prog". The terminal vocabulary is the set of symbols which are not further defined in Table I. All other symbols belong to the non-terminal vocabulary of APPL.

As defined in Table I, an APPL program is the compound symbol <u>int</u> followed by a declaration part and a statement list. The declaration part is a non-empty list of variables declared to be type integer, followed optionally by a set of procedure declarations. The specification part of the procedure declaration specifies how parameters are to be passed.

There are no local variables in a procedure declaration; thus, global variables and formal parameters are the only permissible variables in a procedure body. A procedure body is composed of assignment statements; in particular, nested procedure declarations, and calls to procedures from within procedures are forbidden.

The statement list of an APPL program consists of assignment statements and procedure calls. The values of constants, variables, and recursively defined binary expressions may be assigned to variables. For simplicity, only the integer addition operator (+) is permitted in binary expressions.

The semantics of a procedure call is only defined when the call is to the procedure declared in the declaration part of the program. External

procedure calls are not meaningful in APPL, although not expressing forbidden by the concrete syntax of APPL. Other syntactically valid constructs which result in undefined semantics are: Using uninitialized variables, using the same identifier more than once in a parameter list, failing to include the specification part for a formal parameter, and passing expressions by name with assignments to the corresponding formal parameter in the procedure body. Thus, the concrete syntax of APPL generates a class of syntactically valid programs which is larger than the class of semantically meaningful programs. An APPL program which is both syntactically valid and semantically meaningful is referred to as a well-defined program.

TABLE I

CONCRETE SYNTAX OF APPL IN EBNF

CS = {V_N, V_T, C, prog}

V_N = {prog, declr-pt, stmt-list, vari, id, par-list, spec-pt, proc-body, ss-vari, const, assgn, var, expr, ss-var, bin, stmt, proc-call, arg-list}

V_T = {int, end, proc, val, ref, name,;,←,+,(,),,,A,B,...Y,Z, 1,2,3....}

C = {C1, C2,...C17,C18}

```
C1   prog:: =  int declr_pt; stmt_list end;
C2   declr_pt:: = {,.vari...}{[proc id(par_list); spec_pt proc_body end;]}
C3   vari:: = id|ss_vari
C4   id:: = A|B|C|........X|Y|Z
C5   ss_vari:: = id(const)
C6   const:: = 1|2|3|........
C7   par_list:: = {,.id...}
C8   spec_pt:: = {[val{,.id...};][ref{,.id...};] [name{,.id...};]}
C9   proc_body:: = {;.assgn...}
C10  assgn:: = var ← expr
C11  var:: = id|ss_var
C12  ss_var:: = id(expr)
C13  expr:: = const|var|bin
C14  bin:: = expr + expr
C15  stmt_list:: = {;.stmt...}
C16  stmt:: = assgn|proc_call
C17  proc_call:: = id(arg_list)
C18  arg_list:: = {,.expr...}
```

IV. *THE ABSTRACT SYNTAX OF APPL*

The abstract syntax of APPL is specified by an abstract grammar, which is defined in terms of elementary objects, selectors, predicates, and a distinguished predicate, "is-prog". Elementary objects are terminal symbols of concrete programs; they are combined into compound objects that preserve the linguistic structure of APPL programs. Selectors are used to identify the various components of abstract programs, and program components are represented as <selector:object> pairs.

An abstract program is a compound object, t, that satisfies the distinguished predicate is-prog (t). The predicate is-prog is defined in terms of other predicates that are satisfied by the various components of the abstract program. All predicates are ultimately defined in terms of "elementary predicates", which are satisfied by elementary objects.

Abstract programs can be represented as labeled parse trees of concrete programs. The branches of the parse tree are labeled by selectors, which serve to name the nodes of the tree. Each node and its associated subtree must satisfy a predicate which is true for that particular class of objects. The root of the parse tree satisfies the predicate is-prog and the terminal nodes of the parse tree satisfy elementary predicates.

Formally, the abstract syntax of APPL is specified by a quadruple:

AS = (EO, S, P, is-prog)

where:
EO is a set of elementary objects
S is a set of selectors
P is a set of predicates
is-prog is the distinguished predicate in P

The set of elementary objects for APPL is the union of the sets of symbols that satisfy various elementary predicates. A set of elementary objects satisfying an elementary predicate, "is-pred", is denoted as "$\widehat{\text{is-pred}}$". APPL has six sets of elementary objects:

$\widehat{\text{is-int}}$ = {1,2,3,...}

$\widehat{\text{is-\underline{int}}}$ = {$\underline{\text{int}}$}

$\widehat{\text{is-id}}$ - {A,B,...Y,Z}

$\widehat{\text{is-idd}}$ = $\widehat{\text{is-id}}$ U {Ai,Bi,...Yi,Zi|i=1,2,3,...}

$\widehat{\text{is-spec}}$ = {$\underline{\text{ref}}$, $\underline{\text{val}}$, $\underline{\text{name}}$}

$\widehat{\text{is-op}}$ = {+}

Thus: EO = $\widehat{\text{is-int}}$ U $\widehat{\text{is-\underline{int}}}$ U $\widehat{\text{is-id}}$ U $\widehat{\text{is-idd}}$ U $\widehat{\text{is-spec}}$ U $\widehat{\text{is-op}}$

The selectors are chosen to reflect the linguistic structure of APPL. The set of selectors is not arbitrary; however, the names of selectors are chosen by mnemonic considerations.

S = {s-dp, s-sl, s-pl, s-sp, s-bo, s-lp, s-rp, s-id, s-ss,
s-op,s-rd, s-al } U $\widehat{\text{is-idd}}$

The selector abbreviations are:

dp	declaration part
sl	statement list
pl	parameter list
sp	specification part
bo	procedure body
lp	left part
rp	right part
id	identifier
ss	subscript
op	operator
rd	operand
al	argument list

Identifiers from the set is-îdd are also used as selectors.

The set of predicates satisfied by the various linguistic components of abstract APPL programs is summarized in Table II. A discussion of the predicates follows:

(P1) is-prog = (<s-dp:is-decl-pt>,<s-sl:is-stmt-list>)

An abstract APPL program consists of a declaration part and a statement list. The declaration part is an unordered set of entities, whereas the statement list is an ordered list of statements. The convention of defining a list of entities by specifying the characteristics of a typical element in the list will be used. Thus, a statement list will be defined by defining a statement.

(P2) is-decl-pt = ({<name:is-<u>int</u>||is-idd(name)>},
is-Ω V{<name:is-proc-decl||is-id(name)>})

A declaration part is a finite, unordered set of names, each paired with the type <u>int</u>, where each name satisfies the elementary predicate is-idd; and an optional part consisting of identifiers associated with procedure declarations.

(P3) is-proc-decl = (<s-pl:is-id-list>,<s-sp:is-spec-pt>,
<s-bo:is-assgn-list>)

A procedure declaration consists of a parameter list, which is a list of identifiers, a specification part, and a procedure body, which is a list of assignment statements.

(P4) is-spec-pt = {<name:is-spec||is-id(name)>}

The specification part of a procedure is a finite set of identifiers, each paired with one of the symbols: *ref*, *val*, or *name*. These symbols comprise the set is-spec.

(P5) is-stmt = is-assgn V is-proc-call

The statement list of an abstract APPL program is an ordered set of statements. Statements are either assignment statements or procedure calls.

(P6) is-assgn = (<s-lp:is-var>,<s-rp:is-expr>)

An assignment statement consists of a left part, which is a variable, and a right part which is an expression.

(P7) is-var = is-id V is-ss-var

(P8) is-ss-var = (<s-id:is-id>,<s-ss:is-expr>)

A variable is an identifier or a subscripted variable. The latter is an identifer, subscript pair; and the subscript is an expression.

(P9) is-expr = is-int V is-var V is-bin

(P10) is-bin = (<s-rd1:is-expr>,<s-rd2:is-expr>,<s-op:is-op>)

An expression is an integer, or a variable, or a binary expression. Binary expressions are recursively defined as two operands, which are expressions, and an operator.

(P11)   is-proc-call = (<s-id:is-id , s-al:is-arg-list>)

(P12)   is-arg = is-expr

A procedure call consists of the procedure name, which is an identifier, and an argument list. Each element of the argument list is an expression.

Abstract APPL programs can now be defined as members of the set is-prog. The abstract program corresponding to the typical concrete program in Section II is illustrated in Figure 1.

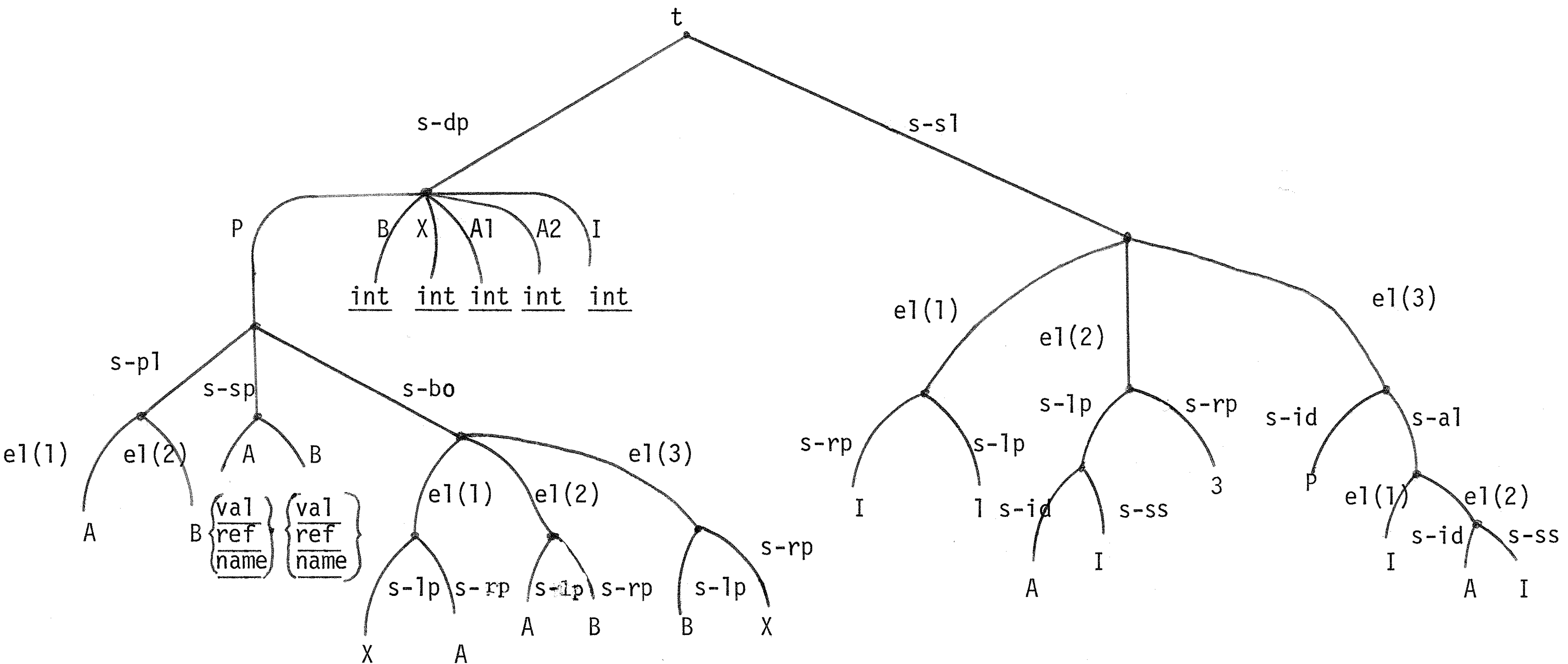


FIGURE 1

TYPICAL ABSTRACT APPL PROGRAM

TABLE II

ABSTRACT SYNTAX OF APPL

AS = (E0, S, P, is-prog)

E0 = is$\hat{-}$int U is$\hat{-}$*int* U is$\hat{-}$id U is$\hat{-}$idd U is$\hat{-}$spec U is$\hat{-}$op

S = {s-dp, s-sl, s-pl, s-op, s-bo, s-lp, s-rp, s-id, s-dd, s-op, s-rd,
s-al} U is$\hat{-}$idd

P = {P1, P2, ... P11, P12}

(P1) is-prog = (<s-dp:is-decl-pt>,<s-sl:is-stmt-list>)

(P2) is-decl-pt = ({<name:is-*int*||is-idd(name)>},
is-Ω V{<name:is-proc-decl||is-id(name)>})

(P3) is-proc-decl = (<s-pl:is-id-list>,<s-sp:is-spec-pt>,
<s-bo:is-assgn-list>)

(P4) is-spec-pt = {<name:is-spec||is-id(name)>}

(P5) is-stmt = is-assgn V is-proc-call

(P6) is-assgn = (<s-lp:is-var>,<s-rp:is-expr>)

(P7) is-var = is-id V is-ss-var

(P8) is-ss-var = (<s-id:is-id>,<s-ss:is-expr>)

(P9) is-expr = is-int V is-var V bin

(P10) is-bin = (<s-rd1:is-expr>,<s-rd2:is-expr>,<s-op:is-op>)

(P11) is-proc-call = (<s-id:is-id>,<s-al:is-arg-list>)

(P12) is-arg = is-expr

is$\hat{-}$int = {1, 2, 3, ... }

is$\hat{-}$*int* = {*int*}

is$\hat{-}$id = {A, B, ...Y, Z}

is$\hat{-}$idd = is$\hat{-}$id U{ Ai, Bi, ...Yi, Zi |i = 1, 2, ...}

is$\hat{-}$spec = {*ref*, *val*, *name*}

is$\hat{-}$op = {+}

Selector abbreviations as previously stated.

V. THE ABSTRACT APPL MACHINE

The abstract machine is a sequential machine which interprets abstract APPL programs. The machine is characterized by internal states and a transition function that maps states into successor states.

The initial state of the machine, ξ_0, incorporates the abstract program and the input data, so that successor states are functions only of present states. The final state of the machine is reached when the transition from some state, ξ_n, yields a null successor instruction.

The meaning (semantics) of an APPL program is defined as the sequence of machine states, $(\xi_0, \xi_1, \ldots \xi_n)$ assumed by the machine as it interprets the abstract program.

V.I *The Abstract Machine State*

The structure of the machine states is defined in the same metalanguage (VDL) used to define the abstract syntax. Thus, the machine state, ξ, can be represented as a tree structured object which satisfies the predicate is-state(ξ).

The formal definition of the APPL machine state is summarized in Table III, and discussed below.

```
(S1)  is-state = (<s-c:is-c>,<s-e:is-e>,<s-dn:is-dn>,<s-un:is-un>,
                  <s-dp:is-dp>)
```

The machine state is a 5-tuple of <selector:object> pairs consisting of a control component, c, an environment component, e, a denotation component, dn, a unique name component, un, and a dump component, dp.

(S2) is-c = "see Section V.2"

The control component contains the abstract machine instructions which interpret the abstract program. A detailed specification of these instructions is deferred to Section V.2.

(S3) is-e = ({<name:is-un||is-idd(name)>})

The environment component is a finite set of <selector:object>pairs, where each selector is a member of the set of elementary objects is-idd, and each object is a unique name. The environment component associates each identifier with a unique name. The use of unique names solves the renaming problem in procedure calls, and permits the sharing of data items between identifers, as in call by reference.

(S4) is-dn = ({<n:is-den||is-un(n)>})

(S5) is-den = Ω V is-int V is-proc-dn V is-name-dn

The denotation component is a finite set of <selector:object>pairs. The selectors are unique names, and the objects satisfy the predicate is-den. The denotation component contains information about identifers. To locate the information associated with an identifier, the environment component is consulted to find the unique name corresponding to the identifier (the identifier is the selector, and the unique name is the object). The unique name is then used as the selector in the denotation component to locate the information. If the identifier is an integer variable, the corresponding object in the denotation component is either

an integer from the set $\hat{\text{is}}$-int, or else satisfies the predicate is-Ω in the case of uninitialized variables. The denotation of procedure identifiers satisfies the predicate is-proc-dn, and the denotation of formal parameters called by name satisfies the predicate is-name-dn.

(S6) is-proc-dn = (<s-tp:proc>,<s-pl:is-id-list>,
<s-sp:is-spec-pt>,<s-bo:is-assgn-list>)

The denotation of a procedure identifier is a quadruple of <selector: object>pairs. The pairs are <type selector:proc>,<parameter list selector: identifier list>,<specification part selector:specification part>, and <procedure body selector:assignment statement list>.

(S7) is-name-dn = (<s-tp:name>,<s-tx:is-arg

The denotation of a formal by-name parameter has two components: a <type selector:name> pair that labels the identifier as a formal by name parameter, and a <text:argument> pair which has the text of the actual by-name parameter from the procedure call.

(S8) is-dp = (<s-dp:is-e>) V (<s-dp:is-ξ>)

The dump component of the machine state is normally used to save the environment component of the state when a procedure is called. Thus, the calling program <identifier:unique name> mappings are saved and restored upon procedure exit. The saved calling environment is also used to evaluate actual arguments which correspond to formal parameters in procedure bodies. The dump is also used to save the entire machine state prior to abnormal termination of the computation due to run-time error conditions (see Section V.2).

(S9) $\hat{\text{is-un}} = \{N_i | i = 1, 2, 3 \ldots\}$

The unique name component contains only one elementary object, which is the next available unique name.

The following predicates have been used in the specification of APPL machine states. These predicates are defined in Table II:

is-int

is-id

is-idd

is-spec

is-assgn

is-arg

The initial state of the APPL machine, ξ_0, is defined in terms of the construction operator, μ_o (see Appendix I):

$$\xi_0 = \mu_o(\langle\text{s-c}:\underline{\text{int-prog}}\ (t)\rangle, \langle\text{s-un}:N_1\rangle)$$

where t satisfies the predicate is-prog(t).

The initial control component has a single instruction, int-prog, and the unique name component is initialized to N_1. All other state components are initially null.

The initial machine state is illustrated in Figure 2.

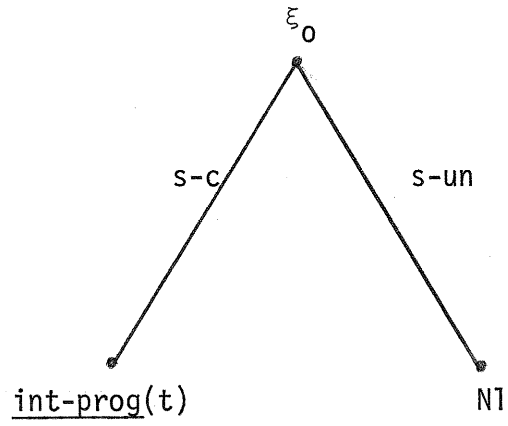


FIGURE 2

Initial Machine State

TABLE III

APPL ABSTRACT MACHINE STATE

Elementary Objects:

EO = is$\hat{-}$int U is$\hat{-}$idd U is$\hat{-}$spec U is$\hat{-}$un U proc U name U Ω

Selectors:

S = {s-c, s-e, s-dn, s-dp, s-un, s-tp, s-pl, s-sp, s-bo, s-tx} U
is$\hat{-}$idd U is$\hat{-}$un

Predicates:

(S1) is-state = (<s-c:is-c>,<s-e:is-e>,<s-dn:is-dn>,<s-un:is-un>,
<s-dp:is-dp>)

(S2) is-c = "see Section V.2"

(S3) is-e = ({<name:is-un||is-idd(name)>})

(S4) is-dn = ({<n:is-den||is-un(n)>})

(S5) is-den = Ω V is-int V is-proc-dn V is-name-dn

(S6) is-proc-dn = (<s-tp:proc>,<s-pl:is-id-list>,
<s-sp:is-spec-pt>,<s-bo:is-assgn-list>)

(S7) is-name-dn = (<s-tp:name>,<s-tx:is-arg-list>)

(S8) is-dp = (<s-dp:is-e>) V (<s-dp:is-ξ>)

(S9) is$\hat{-}$un = {N_i|i = 1, 2, 3 ...}

Initial State: ξ_0 = μ_0(<s-c:int-proc(t)>,<s-un:N_1>)

Final State: ξ_n; where s-c (ξ_{n+1}) = { }

Notes: 1. See section V.2 for the definition of the transition function Λ, and the control predicate is-c.

2. See Table II for the following definitions:
is$\hat{-}$int, is$\hat{-}$idd, is$\hat{-}$spec, is$\hat{-}$un, is$\hat{-}$id, is$\hat{-}$arg.

V.2 The Transition Function

State transitions result from the interpretation of instructions contained in the control component of the state. Macro instructions alter the state by replacing the current instruction with a subtree of instructions. Value returning instructions are deleted as they are executed, and can modify various components and subcomponents of the machine state.

Instructions are defined in terms of primitive operators which manipulate and transform state components. The primitive operators used in this report are described in an appendix. In addition, the following abbreviations are used in this section:

$$\begin{aligned} E &= \text{s-e}\ (\xi) \\ C &= \text{s-c}\ (\xi) \\ DN &= \text{s-dn}\ (\xi) \\ UN &= \text{s-un}\ (\xi) \\ DP &= \text{s-dp}\ (\xi) \end{aligned}$$

Components of the current state are thus denoted by capital letters. Other abbreviations used here are defined in previous sections of the report.

(I1) <u>int-prog</u> (t) =

<u>int-st-list</u> (s-sl(t));

<u>int-decl-pt</u> (s-dp(t));

<u>upd-env</u> (s-dp(t))

for:is-prog(t)

The initial state contains the macro instr <u>int-prog</u> (t), where t satisfies the abstract syntax predicate is-prog. This macro instruction is replaced by a sequence of instructions to update the environment component, interpret the declaration part of the abstract program, and interpret the statement list of the abstract program. An empty statement list will result in an empty control tree, thus producing the final state of the machine.

(I2) $\underline{\text{upd-env}}\ (t) = \underline{\text{null}};$
$\{\underline{\text{upd-id}}\ (id,n);\ n{:}\underline{\text{un-name}} \mid id(t) \neq \Omega\}$

for is-dp(t)

(I3) $\underline{\text{upd-id}}\ (id,n) = \mu(E;<id:n>)$

for: is-idd(id)
is-un(n)

(I4) $\underline{\text{un-name}} = \text{PASS:UN}$
$\mu(\xi;<UN:N_{i+1}>)$

The update environment instruction (I2) is a macro instruction which is replaced by a set of instructions to create <selector:object> pairs of the form <identifier:unique name> in the current environment component. An entry is created for each identifier in the declaration part of the abstract program, t.

Environment updating for a particular identifier is accomplished by interpreting (I3) and (I4). (I4) passes the current unique name, obtained from the unique name state component, to the second argument position in the <u>upd-id</u> instruction, in (I2) and increments the unique name component by 1. The <identifier:unique name> pair is entered into the

current environment component by the generalized assignment operator, μ, in (I3). See the appendix for the definition of the primitive operator, u. The machine state following interpretation of the upd-env instruction for the abstract program of Figure 1 is illustrated in Figure 3.

```
(I5)  int-decl-pt (t) = null;
                          {int-decl(id ∘ E, id ∘ t)|id(t) ≠ Ω}

      for:  is-dp(t)

(I6)  int-decl(n,m) =
              is-int (m) → μ(DN;<n:Ω>)
              is-proc-decl(m) →
                            μ(DN;<n:μ_0(<s-tp:proc>,
                                       <s-pl:is-id-list>
                                       <s-sp:is-sp>
                                       <s-bo:is-assgn-list>)>)

      for:  is-un(n)
            is-int(m) V is-proc-decl(m)
```

Interpret-declaration-part, (I5), is a macro instruction which is replaced by a set of interpret-declaration instructions, (I6); one for each identifier in the declaration part of the abstract program. The arguments of int-decl instructions are the unique name of the identifier, which is retrieved from the environment component, and the declaration of the identifier, which is retrieved from the declaration part of the abstract program.

Interpret-declaration instructions create entries in the denotation component of the state. For integer variables, the selector is the unique name, n, and the object is a null object, which symbolizes an uninitialized variable.

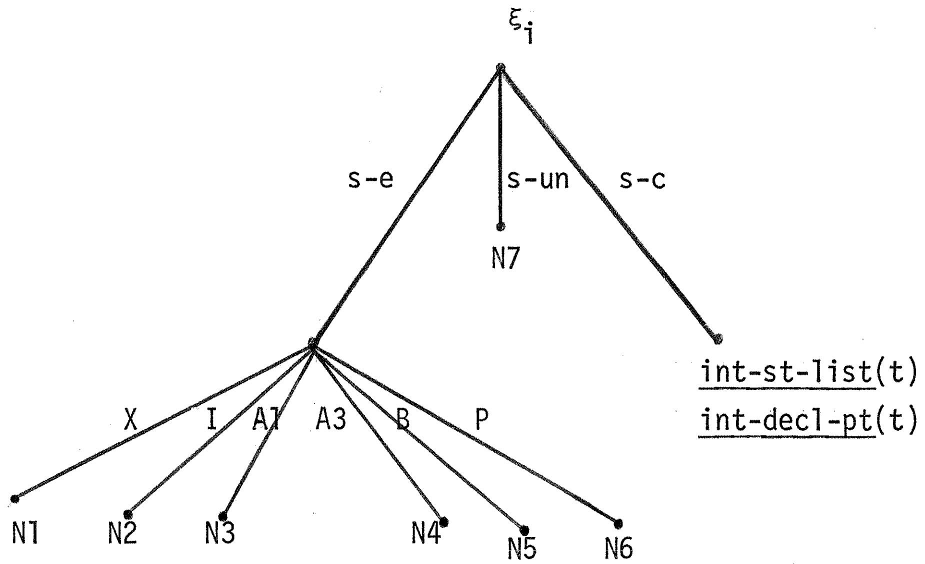


FIGURE 3. MACHINE STATE FOLLOWING INTERPRETATION OF THE UPDATE-ENVIRONMENT INSTRUCTION

For procedure variables, the selector is the unique name, n, and the object is a compound object having four components: <type:proc>, <parameter list:identifiers>, <specification part:specifications>, <procedure body:assignment statements>. The specification part specifies the parameter passing mechanisms (val, ref, or name) for the parameters. The machine state following interpretation of the int-decl-pt instruction for the program in Figure 1 is illustrated in Figure 4.

In more general block structured languages (Algol 60, PL/1), the current environment component is also saved in the procedure denotation, to permit the evaluation of global variables in the proper environment in accordance with the copy rule, as modified by the renaming rule. In APPL, there are only two environments, main program and procedure, and the main program environment is accessible in the dump component during procedure activation. Thus, it is not necessary to save the procedure declaration environment in the procedure denotation.

```
(I7)  int-st-list (t) =
            is-< >(t) → null;
               T → int-st-list (tail (t));
                     int-st (head (t))

for:  is-sl(t)
```

The interpret-statement-list instruction, (I7), is a macro instruction whose argument, t, is a statement list. The instruction is defined in terms of primitive list processing functions. If the statement list is empty, (is-< >(t) = T), the int-st-list instruction is replaced by the null instruction. Otherwise, the instruction is replaced by two instructions: one to interpret the first statement in the statement list, and another to execute the int-st-list instruction for the remaining statements in the statement list.

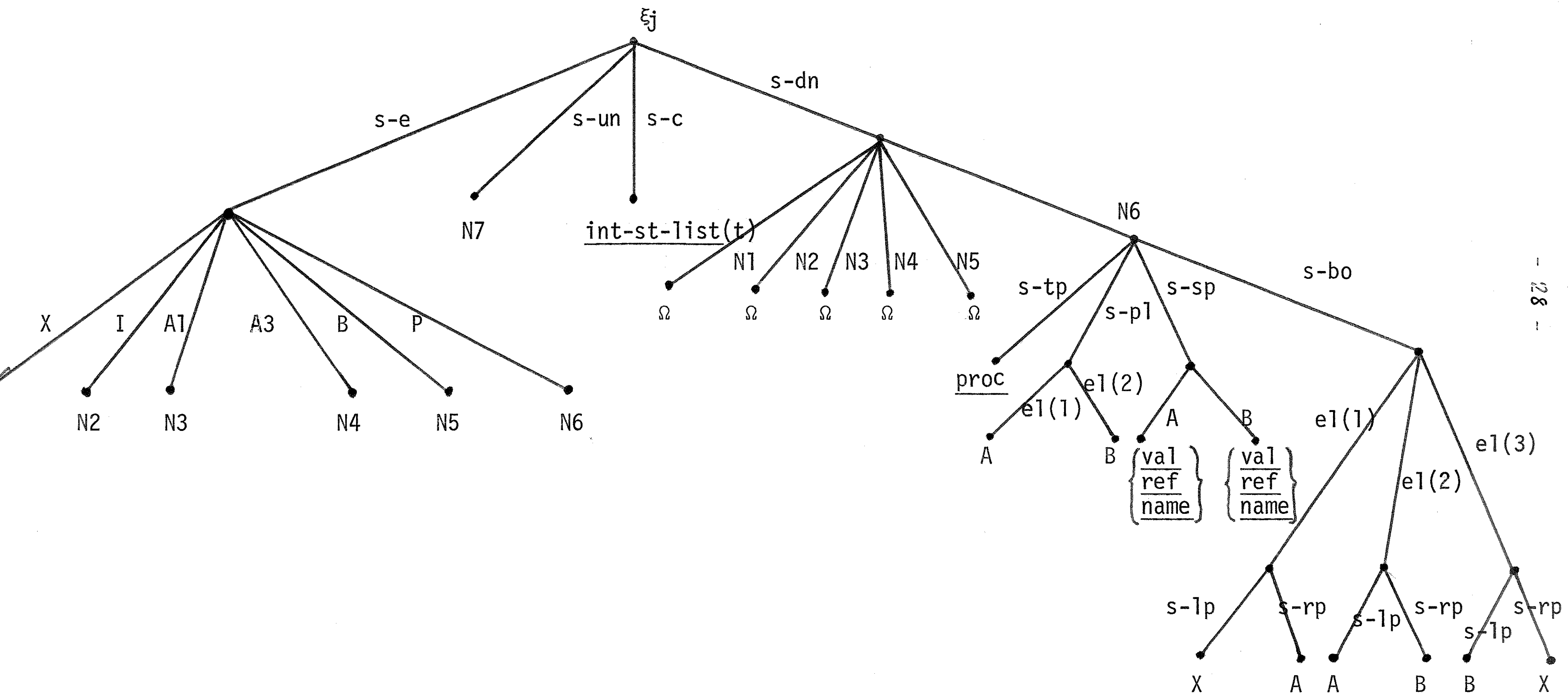


FIGURE 4. MACHINE STATE FOLLOWING INTERPRETATION OF THE INTERPRET DECLARATION PART INSTRUCTION

```
(I8)  int-st (t) =
                  is-assgn (t) → int-assgn-st (t)
                  is-proc-call (t) ∧ (type = proc) →
                                         int-proc-call (t)
                         T → error

where:  type = s-tp ∘ id(E) ∘ s-dn(ξ)
for:    is-st (t)
```

A statement is either an assignment statement or a procedure call. Procedure calls are valid if the identifier is of type proc. The type is verified by checking the type component of the unique name of the identifier in the denotation component of the state. The unique name of the identifier is located in the current environment component. Statements which are neither assignment statements, nor calls to valid procedure identifiers, result in an error condition.

```
(I9)  error = μ(ξ;<s-dp:(<s-e:E>,<s-e:C>,
                        <s-dn:DN>,<s-un:UN>,
                                  <s-dp:DP>)>)
              μ(ξ;<s-c:       >)
```

The error instruction saves the current state of the machine in the dump component of the state, and replaces the control component with the empty instruction, placing the abstract machine in the final state. This is the only instruction which saves all five state components in the dump. Thus, abnormal termination can be detected by examining the dump component of the final machine state.

```
(I10)  int-proc-call (t) =
          μ(ξ;<DP:E>)
           exit;
            int-st-list(s-bo ∘ n_t ∘ DN);
              inst-arg-list(al_t, sp_t, pl_t)
```

where: $n_t = \text{s-id}(t) \circ E$

$al_t = \text{s-al}(t)$

$sp_t = \text{s-sp} \circ n_t \circ DN$

$pl_t = \text{s-pl} \circ n_t \circ DN$

for: is-proc-call (t)

The interpret-procedure-call instruction, (I10), is a macro instruction which is replaced by the instruction sequence: install argument list, interpret statement list, and exit. In addition, a copy of the current environment component is placed in the dump, where it is saved until the exit instruction is interpreted. At that time, the main program environment is reinstalled as the current environment.

The arguments of the install-argument-list instruction are the argument list from the text of the procedure call, the specification part of the corresponding procedure declaration, and the parameter list of the procedure declaration. The latter two arguments are retrieved from the denotation component, using the unique name of the procedure identifier which is in turn retrieved from the current environment component. The argument of the interpret-statement-list instruction is the procedure body; obtained from the denotation component.

(I10a) $\underline{\text{exit}} = (\xi; \langle E{:}DP \rangle)$

The exit instruction restores the main program environment on return from the procedure call.

(I11) $\underline{\text{inst-arg-list}}\ (al,\ sp,\ pl) =$

$$
\begin{array}{c}
len(al) = len(pl) \wedge \\
(\forall i, j)\ [i \neq j \wedge 1 \leq i \leq len(pl) \supset elem(i,pl) \neq elem(j,pl)] \wedge \\
(\forall i)\ [1 \leq i \leq len(pl) \supset el(i)\ (s\text{-}pl) \circ s\text{-}sp \neq \Omega] \rightarrow
\end{array}
$$

$$
\underline{null};\ \{\underline{\text{inst-arg}}(par_i,\ sp_i,\ arg_i) \mid 1 \leq i \leq len(pl)\}
$$

$$
T \rightarrow \underline{error}
$$

where: $par_i = el(i) \circ s\text{-}pl$

$sp_i = par_i \circ s\text{-}sp$

$arg_i = el(i) \circ s\text{-}al$

For: is-arg-list(al)
is-spec-pt(sp)
is-par-list(pl)

Install-argument-list, (I11), is interpreted only if three conditions are satisfied: the length of the argument list equals the length of the parameter list, the same identifier is not repeated twice in the parameter list, and every parameter has a corresponding specification part. If these three conditions are satisfied, the instruction is replaced by a set of instructions to install the individual arguments.

Formal parameters and actual arguments are paired by corresponding element number selectors in the parameter list and the argument list. Selectors for the specification part are the formal parameter names, which are the elementary objects in the parameter list.

```
(I12)  inst-arg (par, spec, arg) =

       is-ref(spec) ∧ is-var(arg) →
                          upd-id(par_i ,n);
                            n:eval-lp(arg)

       is-val(spec) V (is-ref(spec) ∧    is-var(arg)) →
                            assign(n,v);
                               upd-dn(n,Ω); v:int-expr(arg,DP)
                                 n:un-name

       is-name(spec) →
                  upd-dn(n,μ_0(<s-tp:name>,<s-tx:arg>))
                       upd-id (par,n);
                            n:un-name


       for:  is-id(par)
             is-spec(spec)
             is-arg(arg)
```

The interpretation given to the install-argument instruction, (I12), depends on the mechanism utilized to call the argument. If the argument is call by ref, and if the argument is a variable, the unique name assigned to the formal parameter identifier is the same as the unique name of the argument at the time of procedure call. Thus, the formal parameter and the actual argument share the same "memory cell" in the denotation component.

If the argument is call by val, or if the argument is an expression other than a variable and called by ref, the following actions occur: a unique name is generated; the environment is updated by the <formal parameter:unique name> pair; a new entry, with the unique name as the selector, is created in the denotation component; the argument expression is evaluated in the calling environment (which was saved in the dump); and the value of the argument expression is assigned to the unique name in the denotation. Thus, the argument is evaluated in the calling environment, and the formal parameter is initialized to that value. Arguments other than variables (binary expressions and constants) called by reference are handled in the same manner as call by value arguments.

If the argument is called by name, a <formal parameter:unique name> pair is entered in the environment component. The denotation component

is updated by the unique name selector and a compound object consisting of a <type:name> pair, and a <text:argument text> pair. The calling environment has been saved in the dump. The calling environment and the argument text will permit the evaluation of by name arguments in the calling environment when the corresponding formal parameter is referenced in the procedure body.

```
(I13) upd-dn (n, x) = μ(DN;<n:x>)

for:  is-un(n)
      is-VO(x)
```

Update-denotation updates the denotation component using the unique name, n, as the selector and the argument, x, as the object. The argument, x, is any object which satisfies the predicate is-Vienna Object; where a Vienna Object is any object described in the VDL notation.

Figure 5 illustrates the machine state following interpretation of the procedure call in the program of Figure 1.

```
(I14)     int-assgn-st(t) =
                              assign (n, v);
                                  n: int-lp (lp(t)),
                                  v: int-expr (rp(t))

for:  is-assgn (t)

(I15)     assign (n, v) = μ(DN;<n:v>)

for:  is-un (n)
      is-int (v)
```

Interpret-assignment-statement, (I14), is a macro instruction which expands to interpret-left-part; which returns a unique name, and interpret-expression; which returns a value. The value is then assigned as the

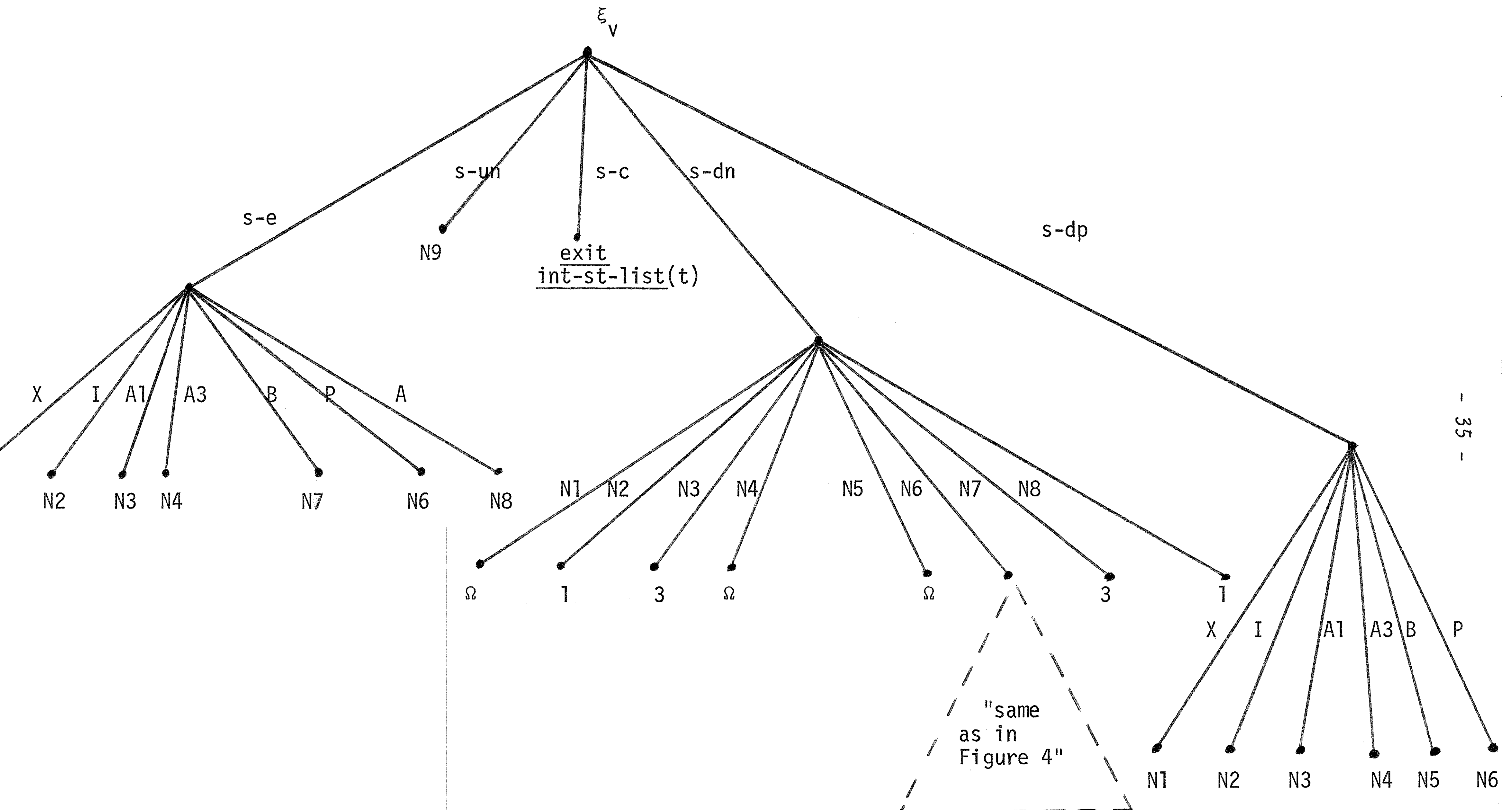


FIGURE 5a. MACHINE STATE FOLLOWING INTERPRETATION OF THE PROCEDURE ENTRY INSTRUCTIONS, FOR CALL BY VALUE

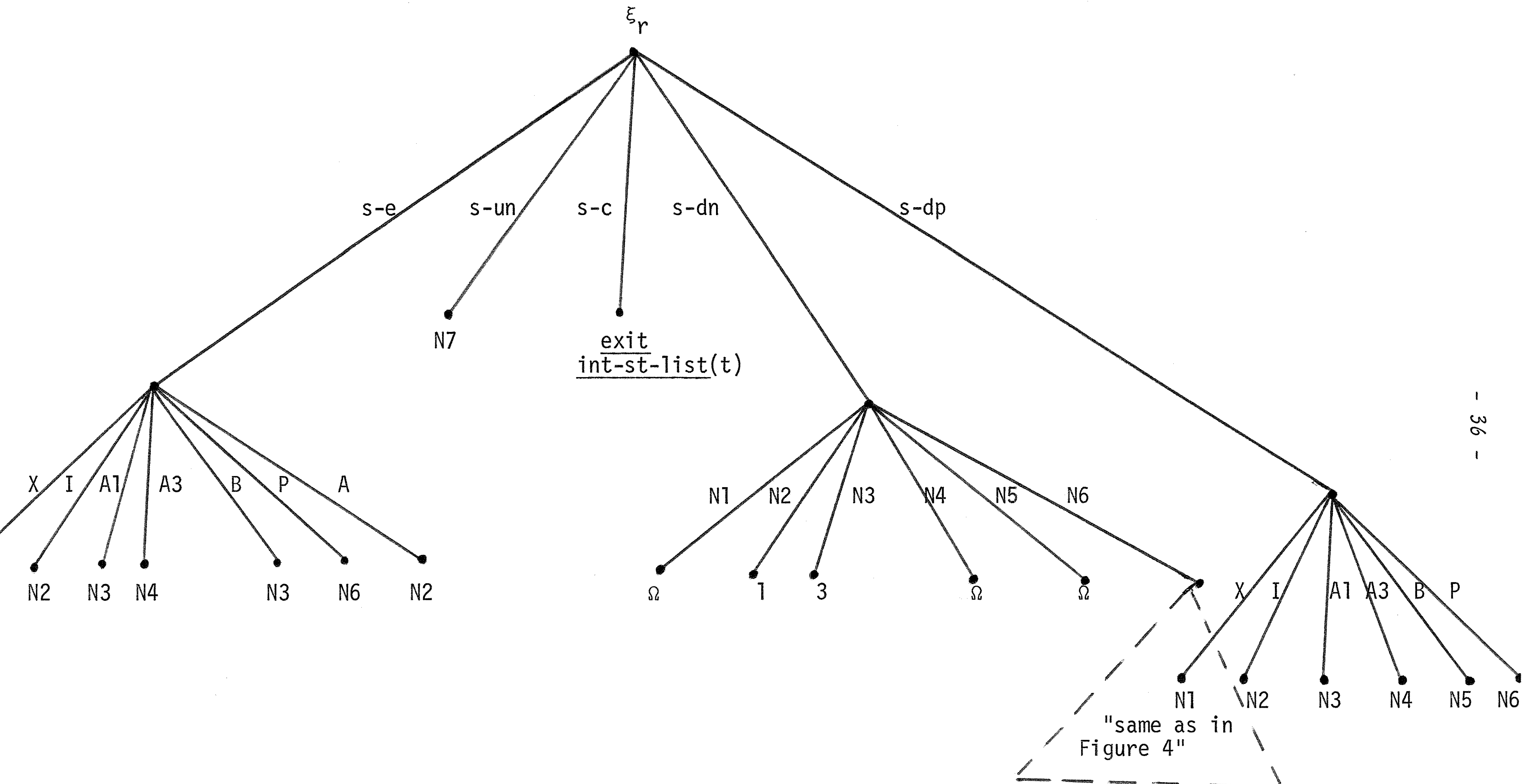


FIGURE 5b. MACHINE STATE FOLLOWING INTERPRETATION OF THE PROCEDURE ENTRY INSTRUCTIONS, FOR CALL BY REFERENCE

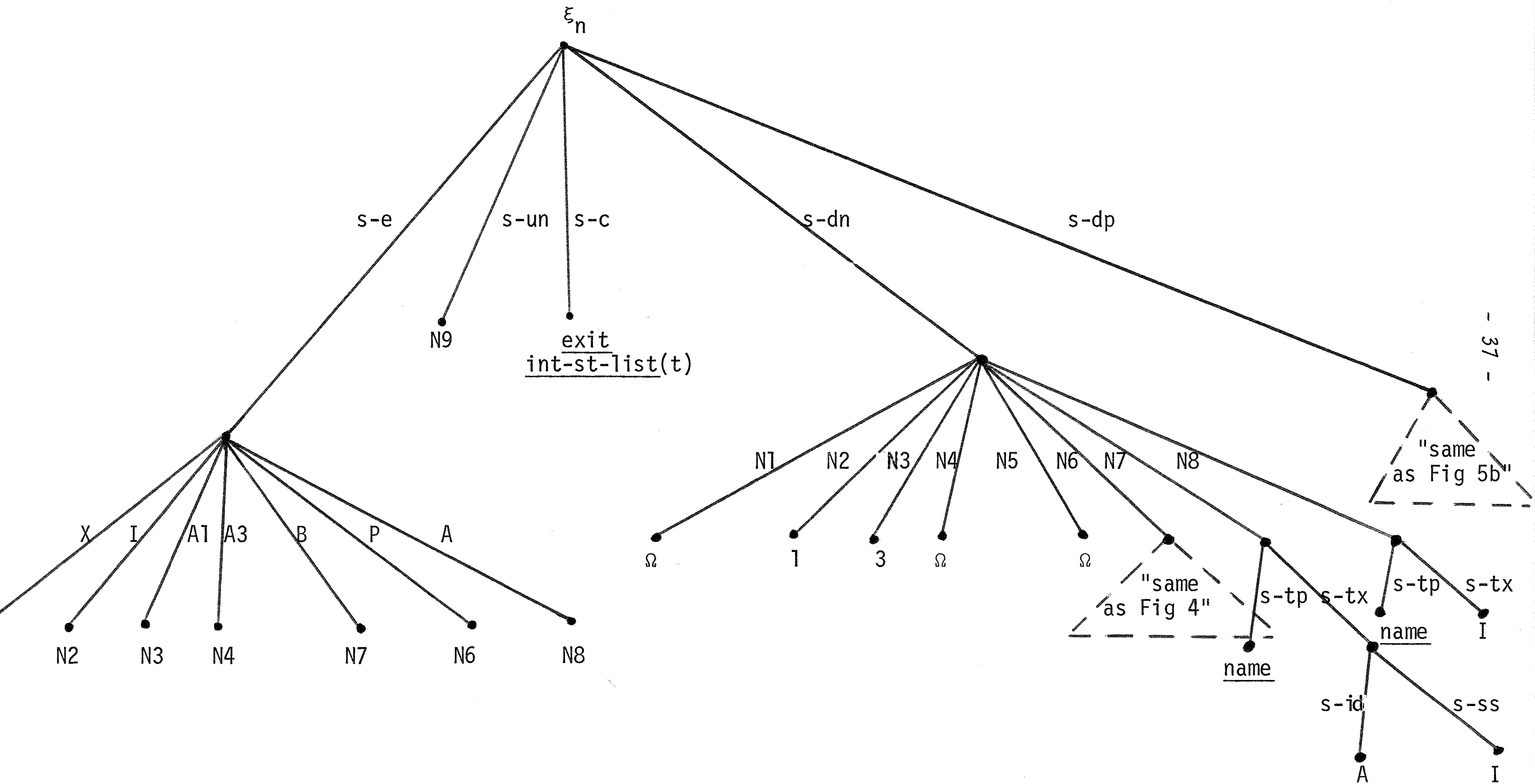


FIGURE 5c. MACHINE STATE FOLLOWING INTERPRETATION OF THE PROCEDURE ENTRY INSTRUCTIONS, FOR CALL BY NAME

object corresponding to the unique name selector in the denotation component by the assign instruction (I15).

(I16) int-lp (lp) =
is-name (s-tp∘ n_{lp}(DN)) → eval-bnlp (lp)
T → eval-lp (lp)

where: n_{lp} = s-id(lp) ∘ E

for: is-var(lp)

If the left part is a by-name parameter, the interpret-left-part instruction is replaced by an evaluate-by-name-left-part instruction. Otherwise, int-lp is replaced by eval-lp. By-name formal parameters are identified by consulting the type component in the denotation of the left part identifier. Formal parameter names are restricted to simple identifiers. Thus, using the left part as a selector in the environment component will always return a unique name if the identifier is a by-name formal parameter.

(I17) eval-bnlp (lp) =

is-id(s-tx ∘ n_{lp}(DN)) → pass:s-id(s-e ∘ DP)

is-ss-var(s-tx ∘ n_{lp}(DN)) → pass:s-idd(s-e ∘ DP);
chk-ss(s-idd, s-e ∘ DP);
s-idd:conc (id,v);
v:int-expr(ss)

T → error

where: n_{lp} = s-id(lp) ∘ E

id = s-id ∘ s-tx ∘ n_{lp}(DN)

ss= s-ss ∘ s-tx ∘ n_{lp}(DN)

for: is-id(lp)

If the text of the argument corresponding to the by-name left part is an identifier, the unique name of that identifier in the calling environment is returned. If the text of the argument is a subscript variable, the subscript is evaluated in the calling environment; the current name of the subscripted variable is formed by concatenating the identifier and the value of the subscript ; a check is made on the subscript range; and the unique name of the subscripted variable in the calling environment is returned.

If the text of the argument corresponding to a left part by-name parameter is not a variable an error instruction is executed, in accordance with the copy rule.

```
(I18)      chk-ss(id, e) =
                      s-id(e) ≠ Ω →  null;
                         T → error;

for:  is-idd(id)
      is-e(e)

(I19)      conc (id, v) = pass:idv

for:  is-id(id)
      is-int(v)
```

The concatenate instruction, (I19), concatenates the identifier and the value of the subscript to form an elementary object in is$\hat{-}$idd, and returns the concatenated value. The check subscript instruction, (I18), verifies the existence of the concatenated identifier in the given environment component by self replacement with the null instruction. Otherwise, it is replaced by the error instruction.

```
(I20)   eval-lp (lp) =
                 is-id(lp) → pass:lp ∘ E
               is-ss-var(lp) → pass:s-idd ∘ E;
                                 chk-ss (s-idd, E);
                                   s-idd:conc(id, v);
                                      v:int-expr(ss)

where:  id = s-id ∘ lp

        ss = s-ss ∘ lp

for:    is-var(lp)
```

Evaluation of a left-part which is not a by-name parameter returns a unique name from the present environment. As before, subscripted variables are evaluated and concatenated into simple variables, and the existence of the resulting identifier is verified.

```
(I21)  int-expr (rp, e) =
                 is-int (rp) → pass:rp
                 is-var (rp) →  pass:eval-var (rp, e)
                 is-bin (rp) →  pass:valu (rd1, rd2, t);
                                  rd1:int-expr(s-rd1 ∘ rp);
                                  rd2:int-expr(s-rd2 ∘ rp)

for:  is-expr(rp)
      is-e(e)
```

```
(I22)  eval-var (rp, e) =
                 is-name (s-tp ∘ n_rp(DN)) → eval-bnrp (rp)
                                  T        → eval-rp (rp, e)

where:  n_rp = s-id(rp) ∘ E

for:    is-var(rp)
        is-e(e)
```

If the right part is a by-name formal parameter, the evaluate-variable instruction is replaced by an evaluate-by-name right-part instruction. Otherwise, evaluate-variable is replaced by evaluate-right-part. As

explained in (I16), by-name formal parameters are identified by the type component in the denotation of the parameter. Also, formal parameters are restricted to being simple identifiers. Thus, using the right-part argument as a selector in the environment component will return a unique name if the right-part is a by-name formal parameter.

Figure 6 illustrates the machine state following exit from the procedure and return to the main program environment for the program of Figure 1.

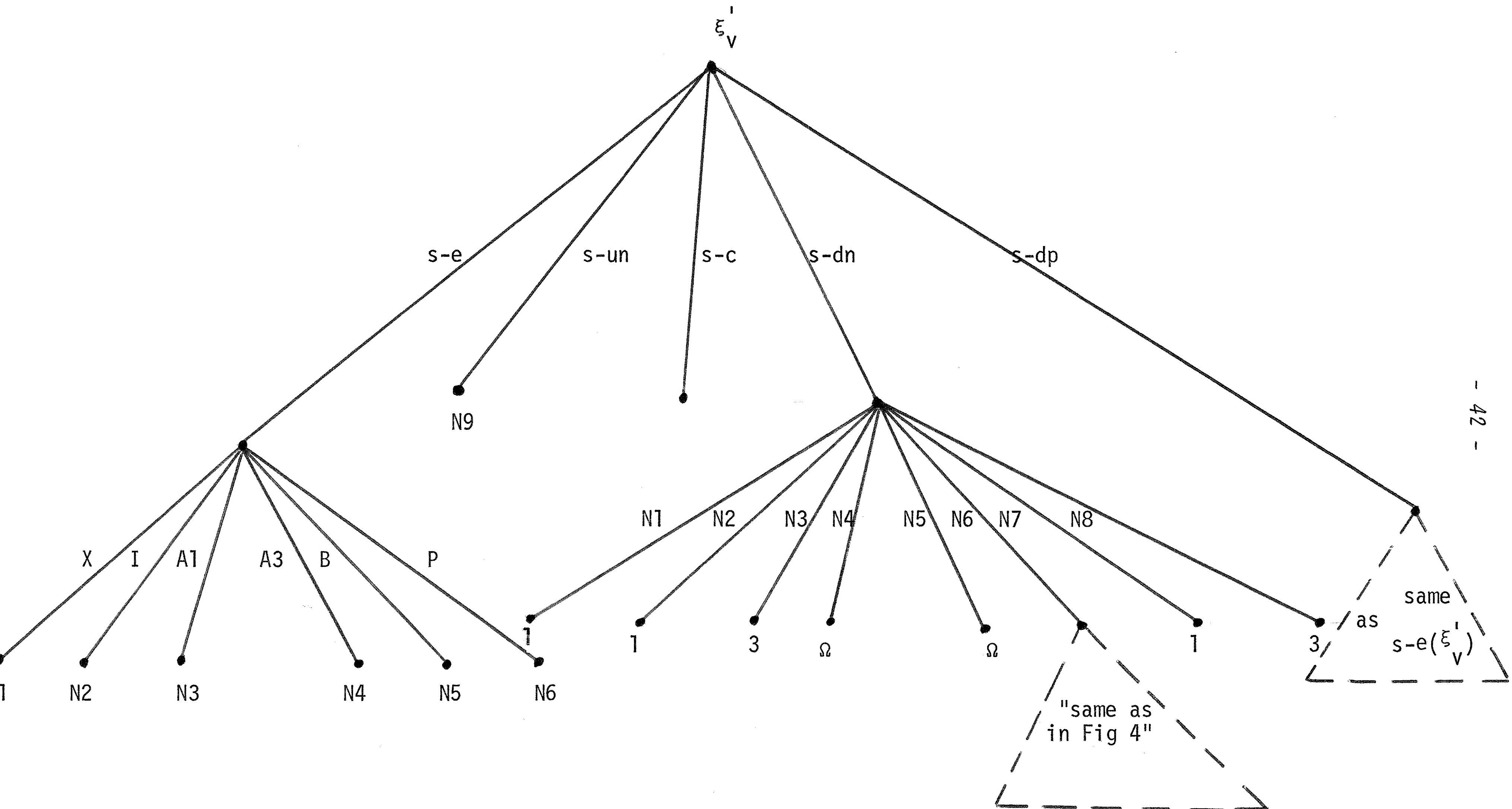


FIGURE 6a. FINAL MACHINE STATE FOR CALL BY VALUE

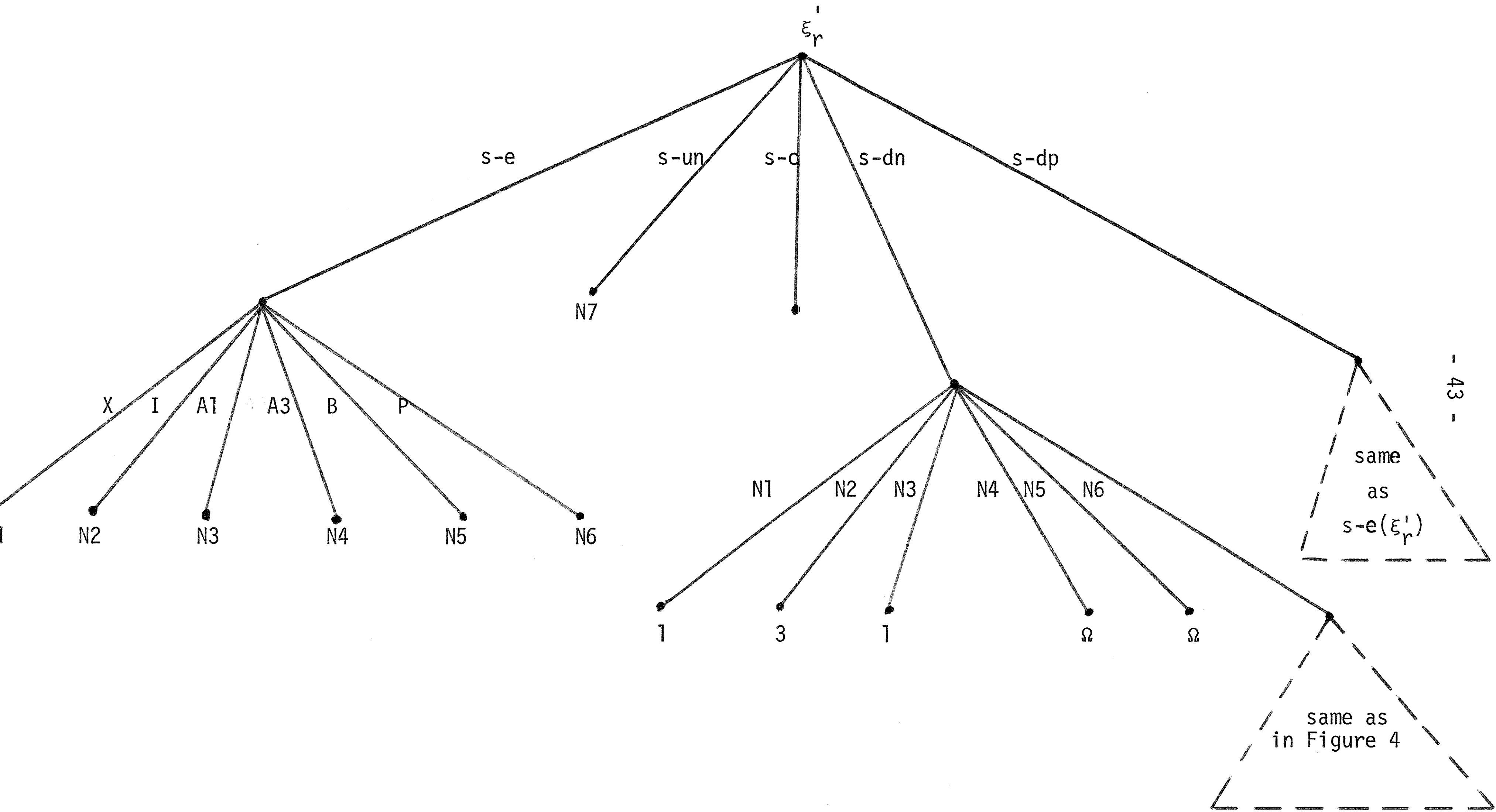


FIGURE 6b. FINAL MACHINE STATE FOR CALL BY REFERENCE

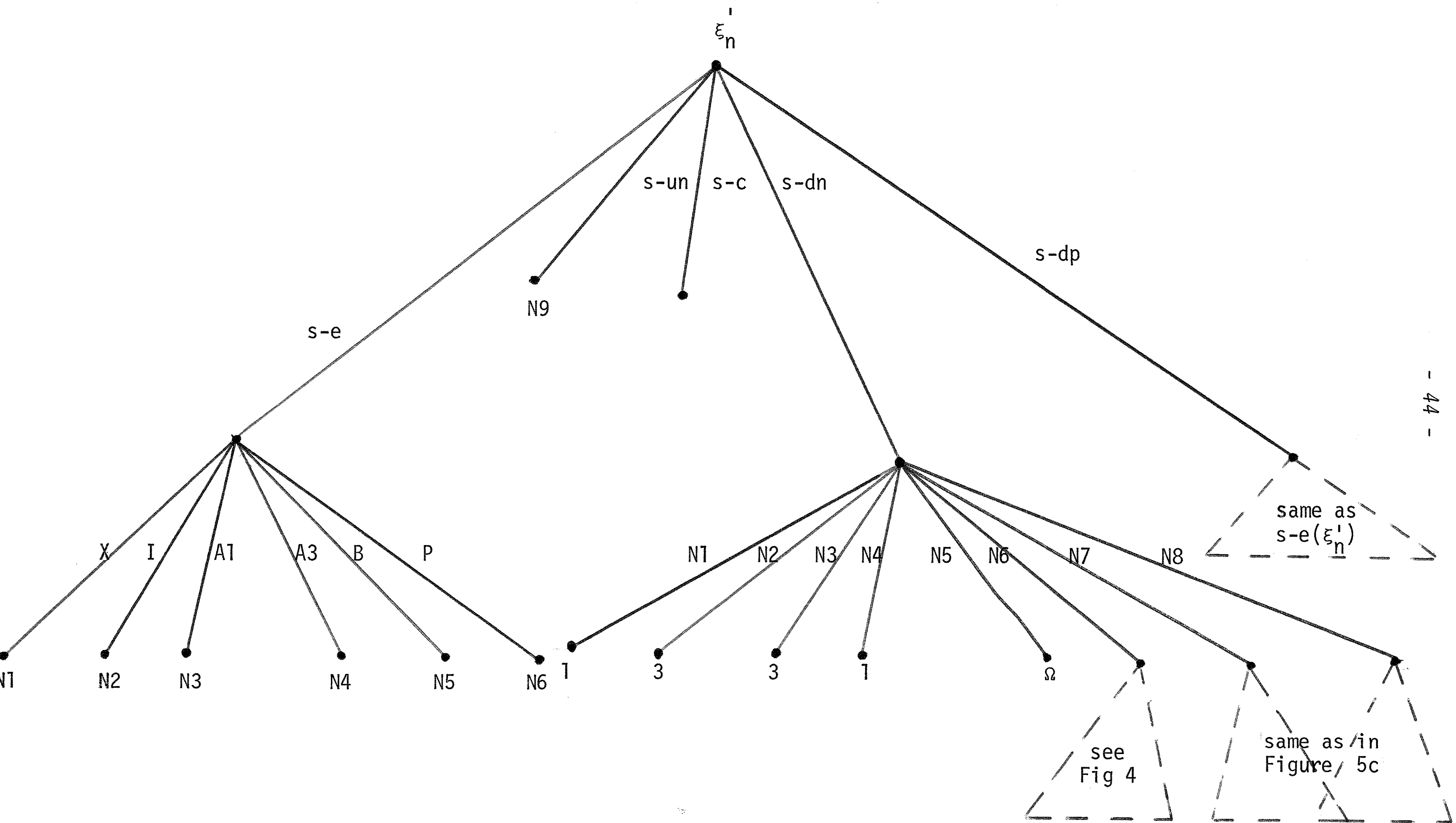


FIGURE 6c. FINAL MACHINE STATE FOR CALL BY NAME

REFERENCES

LLS 68 : Lucas, P.; Lauer, P.; and Stigleitner, H., "Method and Notataion for the Formal Definition of Programming Languages." TR-25.087, IBM Lab. Vienna, June 1968.

WEG 72 : Wegner, P., "The Vienna Definition Language," ACM Computing Surveys, Vol. 4, No. 1, March 1972, p. 5-63.

REF:cah

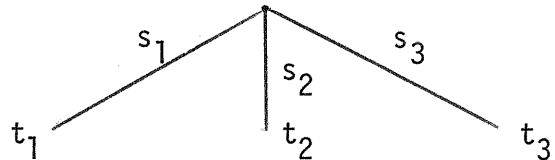
APPENDIX I

DEFINITION OF PRIMITIVE OPERATORS

The two classes of Vienna Objects utilized in this paper are Elementary Objects and Composite Objects. Elementary Objects are atomic elements that have no structural components. Composite Objects have components that are selected by unique selectors. The components of composite objects are either elementary objects or composite objects. The notation for selector-object pairs is of the form <S:OB>.

1. <u>The Construction Operator</u>

   The construction operator, μ_0, accepts a variable number of arguments, each of which is an <S:OB> pair. The effect of applying μ_0 to a set of selector-object pairs is to create a new composite object. For example, $\mu_0(<s_1,t_1>,<s_2,t_2>,<s_3,t_3>)$ =



   where the t_i may be elementary or composite objects.

2. <u>The Selection Operator</u>

   Selectors may be used as operators to select components of composite objects. The operation s(t), where s is a selector and t a composite object, selects the object that is the s-component of object t. If t has no s-component, then $s(t) = \Omega$.

Selectors are either simple selectors or composite selectors. A composite selector is a sequence of simple selectors of the form: $s_i \circ s_{i-1} \circ\circ\circ\ s_2 \circ s_1\ (t)$

Composite selectors are applied in right-to-left order:

$$s_3 \circ s_2 \circ s_1(t) = s_3(s_2(s_1(t)))$$

It is sometimes convenient to use an elementary object, $s_i(t\varepsilon)$, as a selector in a different composite object, t_j. The notation is of the form:

$$s_k \circ s_{k-1} \circ\circ\circ\ s_i(t_i) \circ\circ\circ\ s_1(t_j)$$

3. <u>The Generalized Assignment Operator</u>

The generalized assignment operator, μ, has two arguments: an object, t, and a selector-object pair, $<x,w>$. In general, x is a composite selector, and t,w composite objects. The result of applying $\mu(t_j<x:w>)$ is defined as follows:

a. If object t has a x-component, replace the x component with w; unless $w = \Omega$. If $w = \Omega$, delete the x-component of object t.

b. If object t has no x-component, create a x component and assign w as the object.

4. <u>PASS</u>

Value returning instructions are of the form: PASS:expression

The value of the expression is to be passed to predecessor instructions, and stored as the value returned by the present instruction.

5. List Operators

Ordered sequences of objects are referred to as lists. The elements of a list are ordered by selectors of the form el(i); i = 1, 2, ...n. A list of n elements is a composite object of the form:

$$t = (<el(1);t_1>,<el(2):t_2>, \ldots .<el(n):tn>)$$

An empty list is denoted by the sumbol < >, and satisfies the predicate is-< >.

The following operators apply to lists:

a. len -- The length of a list is the index of the last element in the list; e.g., len(t) = n.

b. head -- The head of a list is the first element; e.g., head (t) = t_1.

c. tail -- The tail of a list is a new list consisting of all elements except the first; e.g., tail (t) = $(<el(1):t_2>,<el(2):t_3>, \ldots <el(n-1):t_n>)$

d. elem -- A given element of an indicated list is returned by this function; e.g., elem (2, tail(t)) = t_3.