1973

A formal description of SYMBOL

Cheng-Wen Cheng
Iowa State University

A formal description of SYMBOL

by

Cheng-Wen Cheng

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Major: Electrical Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1973

TABLE OF CONTENTS

TABLE OF CONTENTS

I. INTRODUCTION

The function of a programming language is to serve as a set of conventions for communicating algorithms - the communication being either between people and people or between people and machines. The efficiency of the communication process is clearly improved as the conventions are better understood. This is just another way of saying that it is advantageous that the programming languages we use be accurately defined. Conventionally, programming languages have been defined by English language descriptions, as written in manuals. The modern idea in this area is to formalize the definition, using some suitable notation (1). The discovery of what sort of notation is "suitable" is really a major unsolved research problem in this area although there have been many excellent approaches to the problem (2,3,4,5,6,7,8,9,10,11).

Also,

> "since the chief aim of programming languages is their use as communication media with computers, it seems only natural to use a basic set of semantic definitions closely reflecting the computer's elementary operations. The invaluable advantage of such an approach is that the language definition is itself a processing system and that implementations of the language on actual machines are merely adaptations to particular environmental conditions of the language definition itself. The question of correctness of an implementation will no longer be undecidable or controversial, but can be directly based on the correctness of the individual substitutions of the elementary semantic units by the elementary machine operations." (1)

Considering the complexity of known compiler systems, this proposal to let the processor itself be the definition of the language seems to be an unreasonable suggestion. When considering SYMBOL, because of its

unique relation between the SYMBOL programming language and its execution language string (translated object string) (12), this suggestion appears in a different light.

The purpose of defining a formal definition of a processor may be arranged into two categories. The description may be organized to reflect the processor either from a user's point of view, showing what can be done, or from the designer's standpoint, indicating how the operations are performed. Since the author has two goals in mind, the description of SYMBOL not only supplies a semantic definition of the SYMBOL programming language but also reflects the special implementation and execution of the SYMBOL machine; the present description of SYMBOL emphasizes mostly the designer's standpoint. Because the execution of instructions within a process causes side effects, it is beneficial for a programmer to know exactly how the operation is performed. In order to reflect the unique structure and concept of the actual SYMBOL system, the abstract machine in this description is specially defined but not in the standard mechanism used by Lucas, Lauer and Stigleitner (9). Both the SYMBOL system and language are originally designed (21):

(1) That it be able to handle variable field length data and instructions.

(2) That it be able to maintain the structure of the data internally.

(3) That it be able to use and operate on this structure.

(4) That it be able to handle symbolic addressing of the data.

(5) That it be able to interpret links in the data correctly.

(6) That it be able to insert links in the data when necessary.

This implies that in order to convey these ideas to the reader accurately, the description must reflect these special properties and their corresponding machine architecture. To accomplish this requirement, an abstract machine is implemented based on the architectural component of the SYMBOL system.

The SYMBOL IIR computer system supports, among other things, a complete, hardware implementation of the SYMBOL programming language. This language is a high-level, procedual, general purpose programming language. One characteristic of the SYMBOL programming language that sets it off from many other high-level languages such as PL/1 and Algol 68 is that stuctured data values may be created and manipulated dynamically. The format of these data structures will generally not be known prior to execution time. The incorporation of dynamic data structures in the SYMBOL programming language places special demands upon the computer system supporting the implementation of that language. Great generality is required to support this feature and this generality must be achieved with a reasonable degree of efficiency.

This thesis focuses on the aspect of the SYMBOL IIR computer system that implements the dynamic data structures used in the SYMBOL programming language. This implementation is a particularly important aspect of the SYMBOL system, is particularly difficult to describe, and places special constraints on the describing mechanism. Thus, it is felt emphasis in this area would provide a good test for the approach taken here.

Recently a formal description of a mini-computer (PDP-8) has been reported (10). It is a very different challenge to describe a much more complicated system to its very detail than has been done on PDP-8. It also is very difficult for the reader to follow such a detailed description.

The Vienna Method has been chosen as the defining notation (9,10,11). It was developed by the PL/1 definition group of the IBM Laboratory in Vienna and its use to formally define PL/1 represents one of the major achievements in the area of language definition (13). It has also been applied on many other languages (10).

The principle of the Vienna method is based on the definition of an abstract machine which is characterized by a set of states and a state transition function. The abstract machine is specified in an artificial language based on abstract concepts of computing. (The abstract machine for describing SYMBOL is based on the logical structure of SYMBOL). It is defined by the set of states Σ which the machine can assume and by a state transition function Λ which, for any given state, s, specifies a successor state $s' \epsilon \Sigma$. The machine stops when it transfers into a subset of state Σ_E, called the end states of the machine. A computation is a sequence of states $s_0, s_1, \ldots, s_i, s_{i+1}, \ldots$. It can be either terminating $s_n \epsilon \Sigma_E$ or nonterminating. The objects to be interpreted by the abstract machine are objects which possess a tree structure. A given program (in abstract form) defined an initial state of the machine. The subsequent behavior of the machine is said to define the interpretation of the program.

The objects which are to be operated by the abstract machine are defined by the abstract syntax. The abstract syntax specifies the essential parts of the structure of the objects over which the interpreter is to work. The abstract syntax will be defined as a set of predicates which not only defines the form of the input objects and intermediate results but also permit the testing of objects within instructions of the interpreter for decision making. The abstract syntax of SYMBOL is based on the SYMBOL programming language but with a minor modification in order to accommodate the property of its translated object string.

Finally the interpretation of the SYMBOL programming language is defined using the abstract machine. The interpretation supplies the rules of executing the abstract program by the abstract machine. The interpretation tries to follow the real execution of the SYMBOL programming language by the SYMBOL system, but the interpretation is conceptual rather than actual.

This thesis has three fundamental goals:

1) To demonstrate that a programming language can be defined by using an abstract machine which closely reflects the logical structure of the actual computer system, and that the interpretation of the definition supplies a set of semantic rules closely reflecting the computer's fundamental operations.

2) To test if the Vienna method can be used to describe a programming language as well as its corresponding computer system efficiently.

3) To supply an unambiguous and concise conceptual description of the SYMBOL system.

II. INTRODUCTION OF THE SYMBOL SYSTEM

The SYMBOL computer executes a program in two modes (14). The first mode is a translation mode and the second mode is an execution mode.

The input program is simply a sequence of characters that expresses a SYMBOL program (15). Whenever a job is chosen, the input program is read in and stored in the memory.

In translation mode, the Translator (16) (TR) scans over the stored program string and generates a corresponding post-fix character string as well as name tables. This post-fix string is called the object string. The name tables are variable tables for blocks and procedures. All of the object string and name tables are stored in the memory. If translation is successful, the processor will enter the execution mode and execute the object string.

The generated object string has two primary features: all variables are separated to a hierarchy of name tables and all language components are in post-fix Polish form. The TR has isolated operands and operators in the source string. These operands and operators now become the basic elements of the object string for the Central Processor (CP) (17,18,19,20) to execute in execution mode.

A. Object String

The translated object string (OS) is stored in the memory consecutively. There is an object string address counter (OSA) pointing to the current executing object string element. Each element may occupy one-half,

one or more than one word of space. If it occupies one-half word, a toggle will indicate which half is the current element. The object string elements are shown in Appendix I (14). Some of them are worth mentioning here.

1. Blocks and procedures

The block structure of the program in the object string is represented by a block number and a block-end pair. The statements of the block are located between the block-end pair. The block number is assigned to the block uniquely during the translation mode. Each block number must match a block end code. The block number is also used to identify the name table of this block. It can be viewed as the name of the block.

The procedure block is the same as an ordinary block, but a transfer command is inserted immediately in front of the procedure block which skips this procedure block in executing. When this procedure is called, it will be reentered at the calling point. The transfer command consists of a code of transfer and a destination address of the transfer. The destination in this case is the next object immediately following the procedure block.

Before a program is executed, it is defaulted into a block with block number zero.

2. Labels

A label in the object string is simply a block number followed by its labelled statement. This block number is the block number of the block in which the label statement is located.

3. Conditional statement

After the translation, the conditional statement is transformed into the following form:

	THEN IF FALSE	
expression	JUMP-address 1	body 1[TRANSFER-address body2]

The expression is the boolean expression. If expression is false, it follows the THEN IF FALSE JUMP-address 1 code and jumps to address 1. Address 1 points to the object immediately following the statement or to body 2. If expression is true, it skips the THEN IF FALSE JUMP-address 1 command and executes body 1. After body 1 is executed, it either goes to the next statement or the TRANSFER-address 2 will skip body 2 and goes to the next statement. Both address 1 following THEN IF FALSE JUMP and address 2 following TRANSFER are inserted by the translator.

4. Scope statement

This statement supplies the inter-block variable linking. This global link is completed in translation mode. This statement is simply ignored in the object string and in execution. In translation mode, the Translator simply inserts a pointer in the name table entry of the linked variable. This pointer points to this variable's entry in the name table of the nesting block.

B. Name Table

Corresponding to each block or procedure, the TR generates a name table and stores it in memory. This name table is also named by its block number.

In the name table, each identifier of this block has two entries. The first is the identifier's name and immediately following it is this identifier's control word (cw). The identifier's name is the actual code of the identifier. The control word consists of two flag fields and two address fields. Flag fields indicate the type of the variable. Address fields store either indirect linking addresses or data. (Please refer to Appendix II for more details.)

The first entry of each name table is reserved for a block control word. It also consists of two flag fields and two address fields. These two address fields are for block linking. The first address field links the nesting block by storing the address of the first entry of the nesting block's name table. The second address field links nested block by storing the address of the first entry of the nested block's name table.

C. Stack

When a block is entered, a push-down stack is created and assigned to this block. This stack can be dynamically expanded. When a new block is entered before the old one exited, a new stack is created and the old one is left in the memory without touching it. When the old block is resumed, its stack will also be resumed. When a block is exited, its corresponding stack is deleted. The system has a (STD) stack data register and a stack address register (STA) for manipulating the stack. STA points the top of the stack and STD stores the top entry of the stack.

D. Memory

The memory is organized in order to meet the dynamic length storage requirements. The main memory is divided into 32 pages. Each page has 32 groups. Each group contains 8 words. One word is 64-bits. A page is divided into three distinct regions. (Refer to Figure 8, p. 16 (14).)

The first region has four words called page headers. Page headers are used to form the page list and to manage the group lists within the page.

The second region is a set of 28 words called group link words. The third region is group data space which consists of the 28 groups of this page. The group data space is for data storage. Corresponding to each group in the group data space, there is a group link word in the second region. This group link word maintains the information for group linking (both forward link and backward link).

A group is the smallest unit of space allocation. Large space is formed by linking groups or even by linking pages.

There are 16 special memory operation commands to give a full service of this special memory organization. (Interested reader please refer to (7,15).)

E. Page List

Page lists are formed by linking pages. Pages available for assignment are maintained on an available page list in the system.

When a user needs space, a user page list is formed by assigning a page from the available page list to this user. A control word is established for this user to manage this page list. When more pages are needed, they are added on to the user's page list. When a job is completed, its user's page list is no longer needed, this page list is given back to the system by returning it to the system's available page list.

F. System Headers

In the memory, for each terminal, a definite space is assigned for the use of system headers which contains all the information for the system management. Only those headers which are related to the execution of programs or to the central processor will be mentioned.

AH0: It contains current block number, block flag, inner block bit and current break point.

AH1: It contains error code, status word, etc.

AH2: It contains I/O op code, object string current address (OSA), CP stack current address (STA).

AH5: It contains CP stack data word (STD).

Inside of AH1, there is a limit counter (LC) which contains an integer L in the range of $0 \leq L \leq 99$. The default value of L is 9. It can also be set to any value within its range by an assignment statement. This value is the upper bound on the relative precision of all dynamic numeric operations.

III. ABSTRACT SYNTAX OF SYMBOL PROGRAMMING LANGUAGE

The abstract syntax of a programming language is defined to identify its abstract programs. Basically source programs require cumbersome scanning and sometimes complicated procedures for gathering the components of statements relevant for the interpretation. Because of this, when designing an interpreter for a programming language, source programs are usually not interpreted directly; instead a translated form of the program is used as the input to the interpreter. This translation process produces structures that allow ready access to the components of statements needed for interpretation. It also eliminates source program elements not required for the semantic interpretation. This translated form of a programming language is defined by its abstract syntax.

The abstract syntax of SYMBOL is based on the syntax of SYMBOL Programming Language (refer to Appendix III). Because the SYMBOL source programs are translated by the Translator into a reverse polish string, the structure of the compound statement is no longer held in this object string but is transformed into a linear form. As a consequence in the abstract syntax, some minor modification of the compound statement has been made. That means in the abstract syntax, compound statements are defined in a one-level list form instead of their original tree form.

Let the set of objects representing the set of object strings of legal programs in SYMBOL be denoted by $\Lambda$is-obj-string.

All predicates have a prefix "is-", and all selectors have a prefix "s-" except possibly identifiers. Identifiers are also used as

selectors. The predicate definitions are labelled (A1), (A2), ... for reference purposes.

Λ is-id	an infinite set of identifiers.
Λ is-blk-no	a set of integers denoting block numbers.
Λ is-blk-end	{END} denotes the end of block.
Λ is-string	an infinite set of sequence of length zero or more of any characters.
Λ is-comt	an infinite set of comments.
Λ is-number	an infinite set of numbers.
Λ is-stm-add	a set of statement address.
Λ is-data-record	an infinite set of input data of terminals.
Λ is-terml-no	a set of terminal numbers.
Λ is-arith-rel	{GREATER, GREATER THAN, GTE, EQUAL, EQUALS, NEQ, LTE, LESS, LESS THAN, $<$, $\leq$, $=$, $\neq$, $\geq$, $>$}.
Λ is-string-rel	{BEFORE, SAME, AFTER}.
Λ is-arith-op	{+, -, *, / }.
Λ is-string-op	{JOIN, FORMAT, MASK}.
Λ is-boolean-op	{AND, OR}.

The set of selectors necessary for the specification of the abstract syntax of SYMBOL is infinite, because the set of identifiers is infinite. However, the set of selectors that are not identifiers S is finite and enumerable. We can define the set of S as follows:

S = {{elem(i)| $1 \leq i \leq n$, } s-recip-list, s-assign-elem,
s-recip-hd, s-recip-tl, s-as, s-as-list, s-field-hd,
s-field-tl, s-as-hd, s-as-tl, s-as-struct,
s-comp-id, s-comp-list, s-exp-hd, s-exp-tl
s-proc-id, s-act-para-list, s-mon-op, s-mon-exp,
s-dya-opd1, s-dya-opd2, s-dya-op, s-exp,
s-L-par, s-R-par, s-goto-id, s-goto-destn, s-ret-id,
s-ret-exp, s-call-id, s-proc-call, s-comt-id, s-comt,
s-init-val-id, s-init-val, s-ds, s-ds-list, s-sf-hd,
s-sf-tl, s-ddp-hd, s-ddp-tl, s-data-struct,
s-id-hd, s-id-tl, s-link-list, s-link-exp,
s-sw, s-sw-id, s-sw-struct, s-ls, s-label-list,
s-llp-hd, s-llp-tl, s-label-struct, s-if-id,
s-jump-ptr, s-transf-id, s-transf-ptr, s-input-id,
s-input-term1, s-input-recip-list,
s-output-id, s-output-term1, s-output-exp-list,
s-part-ref-id, s-part-ref-idx, s-prat-ref-bound,
s-part-ref-length, s-in-id, s-in-comp }

(A1) is-obj-string = is-obj-list

(A2) is-obj-list = {$<$elem(i): is-obj$>$| $1 \leq i \leq n$} v is-Ω
where n is the length of the object string

(A3) is-obj = is-blk-no v is-proc-blk-no v is-proc-hd v
is-stm v is-blk-end v is-proc-blk-end v
is-if-exp v is-if-jump-ptr v is-transf-ptr

(A4) is-proc-blk-no = is-blk-no

(A5) is-proc-blk-end = is-blk-end

(A6) is-proc-hd = is-form-para-list

(A7) is-stm = is-assign-stm v is-goto-stm v is-call-stm v
is-dummy-stm v is-comment-stm v is-init-val-stm v
is-retn-stm v is-link-stm v is-input-stm v
is-output-stm v is-sw-stm

(A8) is-assign-stm = (< s-recip-list:is-recip-list >,
< s-assign-elem:is-exp v is-assign-struct >)

(A9) is-recip-list = (< s-recip-hd:is-recip v is-LIMIT >,
< s-recip-tl:is-recip-list v is- Ω>)

(A10) is-exp = is-prim-exp v is-mon-com v is-dya-com v
is-paren-exp

(A11) is-assign-struct = (< s-as:is-as>, < s-as-list:is-assign-as-list
v is-Ω>)

(A12) is-as = is-field-list v is-Ω

(A13) is-field-list = (< s-field-hd:is-exp>, < s-field-tl:is-field-list
v is-Ω >)

(A14) is-assign-as-list = (< s-as-hd:is-assign-as-pair>,
< s-as-tl : is-assign-as-list v is-Ω >)

(A15) is-assign-as-pair = (< s-as-struct:is-assign-struct>,
< s-as:is-as>)

(A16) is-recip = is-id v is-component v is-proc-call

(A17) is-prim-exp = is-literal v is-id v is-component v
is-proc-call v is-'LIMIT' v is-part-ref v
is-in-comp

(A18) is-component = (<s-comp-id:is-id>, <s-comp-list:is-exp-list>)

(A19) is-exp-list = (<s-exp-hd:is-hd>, <s-exp-tl:is-exp-list>)

(A20) is-proc-call = (<s-proc-id: is-id>, <s-act-para-list : is-act-para-list>)

(A21) is-act-par-list = {<elem(i): exp> | $1 \leq i \leq$ no. of act parameters}

(A22) is-part-ref = (<s-part-ref-id:is-id>, <s-part-ref-idx: is-exp-list v is-Ω><s-part-ref-bound:is-exp>, <s-part-ref-length:is-exp>)

(A23) is-in-comp = (<s-in-id:is-'IN'>, <s-in-comp:is-component>)

(A24) is-mon-com = (<s-mon-op:is-mon-op>, <s-mon-exp : is-mon-exp>)

(A25) is-mon-exp = is-prim-exp v is-paren-exp

(A26) is-dya-exp = (<s-dya-opd1:is-exp>, <s-dya-opd2:is-mon-exp>, <s-dya-op:is-dya-op>)

(A27) is-dya-op = is-rel-op v is-arith-op v is-string-op v is-Boolean-op

(A28) is-rel-op = is-arith-rel v is-string-rel

(A29) is-paren-exp = (<s-L-par : is-'('>, <s-exp:is-exp>, <s-R-par:is-')'>)

(A30) is-goto-stm = (<s-goto-id:is-'GO' v is- 'GO TO'>, <s-goto-destn:is-id v is-component v is-proc-call>)

(A31) is-ret-stm = (<s-ret-id:is-'RETURN'>, <s-ret-exp:is-exp>)

(A32) is-call-stm = (<s-call-id:is-'CALL'>,
<s-proc-call:is-proc-call>)

(A33) is-dummy-stm = is-Ω v is-'CONTINUE'

(A34) is-comment-stm = (<s-comt-id:is-'NOTE'>,
<s-comt : is-comt>)

(A35) is-init-val-stm = (<s-init-val-id : is-id>,
<s-init-val:is-string v is-data-struct>)

(A36) is-data-struct = (<s-ds : is-ds >,
<s-ds-list: is-data-ds-pair-list v is-Ω>)

(A37) is-ds = is-string-fd-list v is-Ω

(A38) is-string-fd-list = (<s-sf-hd:is-string >,
<s-sf-tl:is-string-fd-list v is-Ω >)

(A39) is-data-ds-pair-list = (<s-ddp-hd:is-data-ds-pair>,
<s-ddp-tl:is-data-ds-pair-list v is-Ω>)

(A40) is-data-ds-pair = (<s-data-struct:ls-data-struct>,
<s-ds:is-ds>)

(A50) is-form-Para-list = is-id-list

(A51) is-id-list = (<s-id-hd:is-id>,<s-id-tl:is-id-list>)

(A52) is-link-stm = (<s-link-list:is-recip-list>,
<s-link-exp:is-exp>)

(A53) is-sw-stm = (<s-sw: is-'SWITCH'>,
<s-sw-id: is-id>,
<s-sw-struct: is-label-struct>)

(A54) is-label-struct = (<s-ls: is-ls>,
<s-label-list:is-label-ls-pair-list v is-Ω>)

(A55) is-ls = is-id-list v is-Ω

(A56) is-label-ls-pair-list = (<s-llp-hd:is-label-ls-pair>,
<s-llp-tl:is-label-ls-pair-list v is-Ω
>)

(A57) is-label-ls-pair = (<s-label-struct : is-label-struct>;
<s-ls=is-ls>)

(A58) is-if-exp = is-exp

(A59) is-if-jump-ptr = (<s-if-id:is-'IF FALSE JUMP'>,
<s-jump-ptr:is-stm-add>)

(A60) is-transfer-ptr = (<s-transf-id : is-'TRANSFER'>,
<s-tranf-ptr:is-stm-add>)

(A61) is-input-stm = (<s-input-id:is-'INPUT; v is-'INPUT STRING'>,
<s-input-terml:is-exp>,
<s-input-recip-list:is-recip-list>)

(A62) is-output-stm = (<s-output-id:is-'OUTPUT' v is-'OUTPUT STRING'>,
<s-output-terml:is-exp>,
<s-output-exp-list:is-exp-list>)

(A63) is-data-record = is-data-struct v is-init-val-stm-list

(A64) is-literal = is-number v is-string.

IV. THE STATES OF THE ABSTRACT MACHINE (9)

This section defines the set of states, $\overset{\Lambda}{\text{is-state}}$, which the abstract machine can assume. The set of states is defined to reflect the structure of SYMBOL. The initial states of the programs and the set of end states are also included in the set of states.

Other than the set of elementary objects (EO) and Set of Selectors (S) for the SYMBOL abstract syntax, the following sets of elementary objects and selectors are distinguished:

$\overset{\Lambda}{\text{is-n}}$ =	an infinite set of integers (used for the number of object string elements)
$\overset{\Lambda}{\text{is-a}}$ =	a set of integers (used to name the position of the elements of the space available list)
{s-flag,s-fl,s-bl}	Selectors to distinguish the components of block-link-node. They are flag field, forward link field and backward link field.
{s-cw,s-inf}	Selectors for the components of id-node. They are control-word field and information field.
$\overset{\Lambda}{\text{is-sta}}$	a set of integers (used to name the position of the stack elements)

$\hat{\text{is-flag}}$	a set of codes of block control word flag bits (See Appendix II)
$\hat{\text{is-ptr}}$	a set of pointers
$\hat{\text{is-ma}}$	a set of memory addresses
$\hat{\text{is-s}}$	a infinite set of combinations of selectors
$\hat{\text{is-para-no}}$	a set of integers (used to number the parameters of procedures)
$\hat{\text{is-lc}}$	a set of integers < 100 (used for the contents of the limit counter)
is-nf	a special code which indicates the entry is empty. (null field)

The set of the selectors for the states of abstract machine is:

{s-os, s-ntt, s-stk, s-hd, s-sal, s-dump, s-m, s-c}

(S1) is-state = (<s-os:is-os>,
<s-ntt: is-ntt>,
<s-stk: is-stk>,
<s-r: is-r>,
<s-sal:is-sal>,
<s-d: is-dump>,
<s-m:is-m>,
<s-c:is-c>)

where:

os: object string

ntt: name table tree

stk: stack

r: register

sal: space available list

dump: dump

m: memory

c: control

(S2) is-os = ({ <n:is-obj> || is-n(n) > })

(S3) is-ntt = ({ <b: is-blk-branch> || is-blk-no(b)})

(S4) is-blk-branch = (<s-blk-link:is-blk-link-node> ,
{ <id: is-id-node> || is-id(id)})

(S5) is-blk-link-node = (<s-flag: is-flag> ,
<s-fl:is-ptr> ,
<s-bl : is-ptr>)

(S6) is-id-node = (<s-cw: is-cw> ,
<s-inf : is-inf>)

(S7) is-inf = is-data v is-blk-no v is-s* v is-inf-str

*s = s_1-s_2 . . . $s_i(\xi)$; $i \geq 1$; s_1, s_2 . . . $s_i \in S$

(S8) is-cw = {DNT ,LS, DOS ,LABEL, PROC,
PARA, LINK, GLOBAL }

(S9) is-inf-str = (<s-blk-no: is-blk-no> , <s-obj-no : is-n>)v
(<s-pn: is-para-no> , <s-blk-no> ,
<s-obj-no:is-n>)

(S10) is-data = is-number v is-string

(S11) is-stk = ({<i: is-stk-wd> || is-sta(i)})

(S12) is-stk-wd= is-stk-id-node v is-blk-stk-wd

(S13) is-blk-stk-wd = is-blk-no v (<s-cbc:is-blk-no> <s-nbc:is-blk-no>)
v(<s-osa:is-n>,<s-stp:is-sta>)

(S14) is-r = (<s-osc: is-n>,
<s-stp:is-sta>,
<s-bc:is-blk-no>,
<s-salp:is-a>,
<s-lc:is-lc>,<s-hd:is-hd>)

(S15) is-m = ({<ma: is-m-el> || is-ma(ma)})

(S16) is-m-el = (<s-cw: is-m-cw>,
<s-inf: is-m-inf>,
<s-pred: is-ma>,
<s-succ: is-ma>)

(S17) is-sal = ({<a:is-ma> || is-a(a)})

(S18) is-dump = ({<b: is-st> || is-blk-no(b)})

(S19) is-c = control tree (6)

(S20) is-flag = {block in use, Ω}

(S21) is-stk-id-node = (<s-cw:is-stk-cw>,<s-inf:is-inf>)

(S22) is-stk-cw = is-cw v {RECIP, SUB, SUB-ADD,], ⊇, ∈, <, >,DATA}

(S23) is-m-cw = { LSS, EV, STRING }

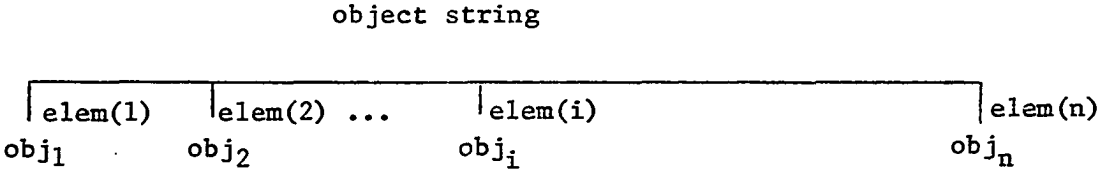
(S24) is-m-inf = id-data v is-s v is-Ω

A. Description of the Components of the State

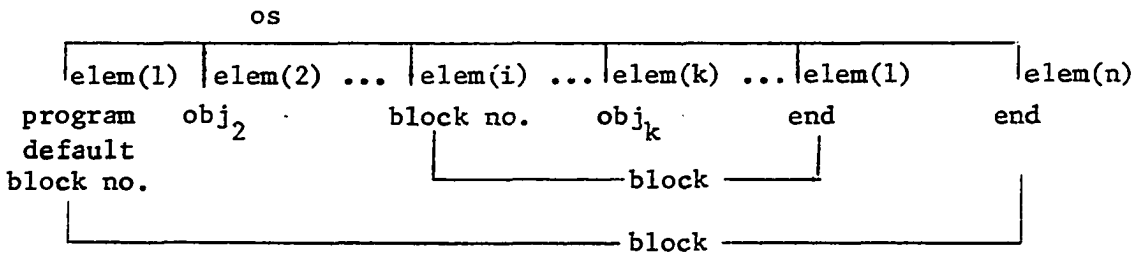
The elements of the set of states which we have defined are corresponding to the object string, name table, stack, registers, page list and memory of the SYMBOL structure.

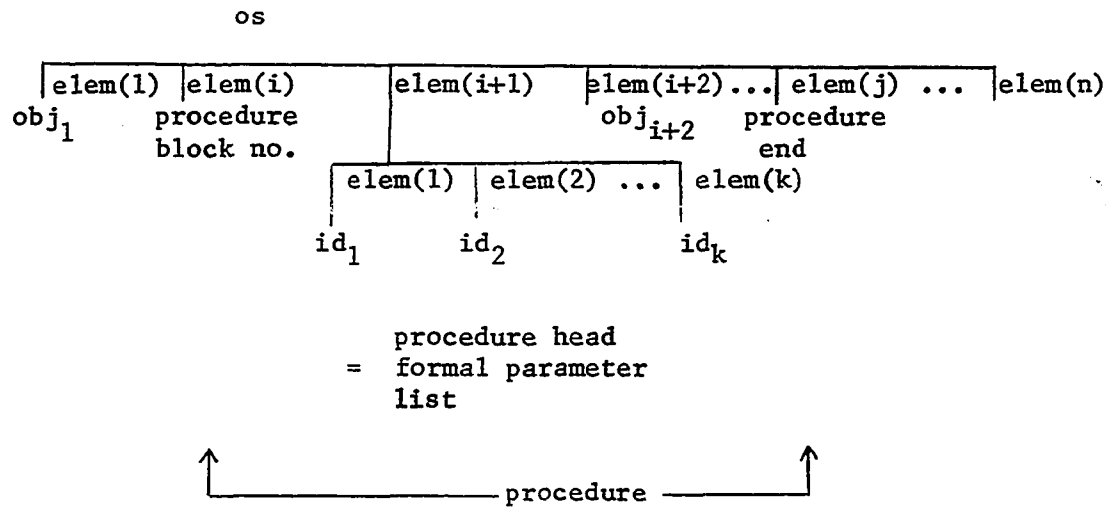
1. Object string (OS)

The object string of a program is a list of object elements. The object elements are defined in the abstract syntax which are elements of $\overset{\Lambda}{\text{is-obj}}$



The block structure and procedure structure in the object string are shown as follows:





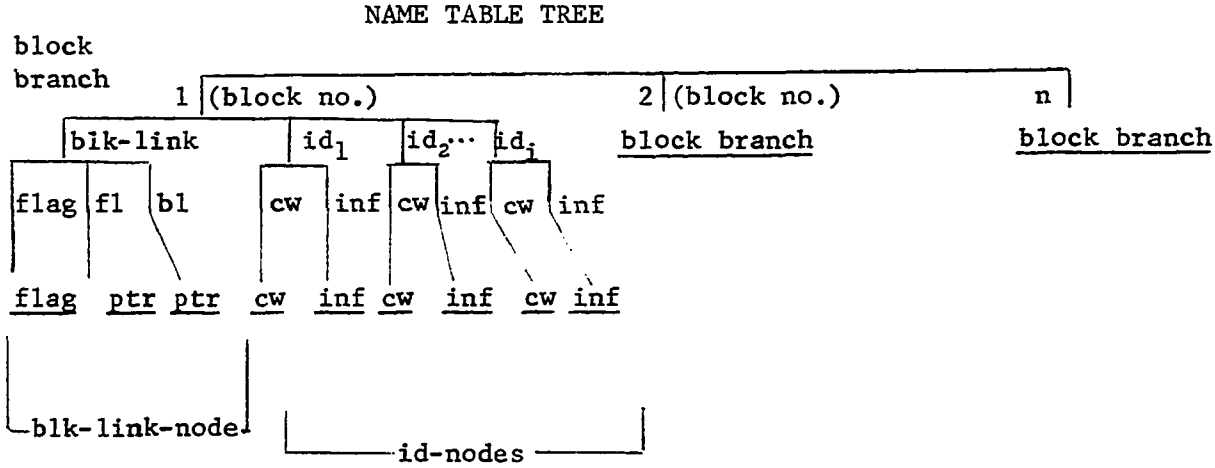
2. Name table tree (ntt)

For the program which is currently being executed by the system, there is a name table tree (ntt). In this name table tree, corresponding to each block or procedure occurring in the program, there is a branch which is call block branch. This block branch is identified by the block number of its corresponding block.

Within the block branch, corresponding to each identifier (variables, labels and procedures) occurring in this block there is an identifier node (id-node). This identifier node is named by its identifier which implies identifiers must be distinct from each other within a block.

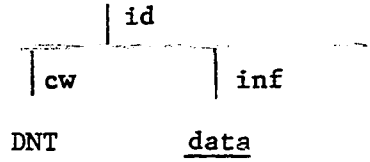
Each block branch also contains a block link node (blk-link-node). This node is used for block linking information storage. It consists of three components: flag field (s-flag), forward link (s-fl) and backward link (s-bl). Before the execution of the program (the initial state), the flag field stores block usage information, the forward link links to the block-link-node of the nesting block or the block immediately leading

current block, and the backward link links to the blk-link-node of the nested block or the block immediately following current block. If there does not exist a nested block or any block following current block, the backward link (bl) is left empty.



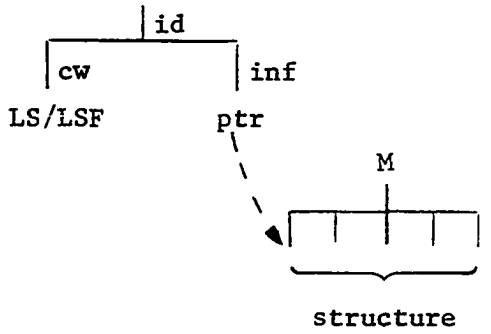
Each identifier node of the block branch again has two components; a control word (cw) and an information field (inf). For different kind of identifiers, their cw and inf are different. Before the initial state, all control words and information fields have been inserted with appropriate contents.

a. Data in name table



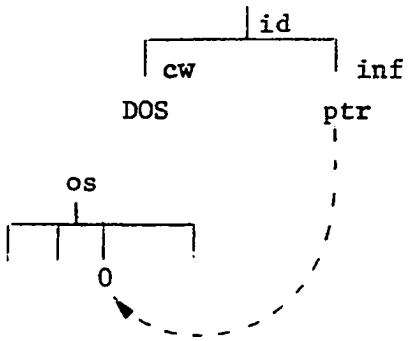
This indicates the identifier is a simple variable with its value stored in its inf. In the initial state, the data in the information field is empty (Ω).

b. Link to structure (LS)



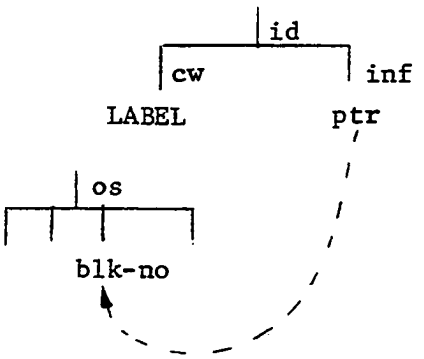
LS indicates this identifier of a structure. The information field contains a pointer which points to the first entry of the vector in the memory. In the initial state, no id-node has LS or LSF control word. The LS is inserted when a structure or vector is assigned to the identifier.

c. Data in object string (DOS)



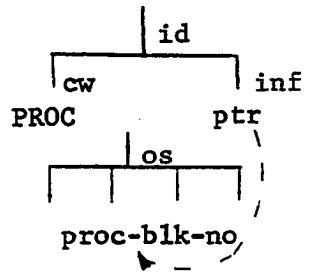
This cw indicates the identifier's data is in the object string. Its inf contains a pointer which points to the location of the data in the object string. Usually this occurs in the initial states. After the execution, the data in the o.s. will be stored into this id's node in the name table or in the memory.

d. Label



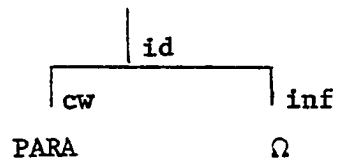
This is the control word of label. The inf contains a pointer which points to the location of this label in the object string. Remember, in the object string, this location is the block-no of this label.

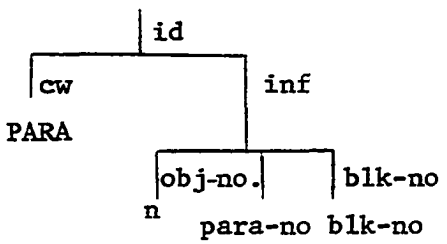
e. Procedure (PROC)



This is a procedure identifier. The inf contains a pointer which points to the first object of this procedure in the object string. This first object is the pro-blk-no of this procedure.

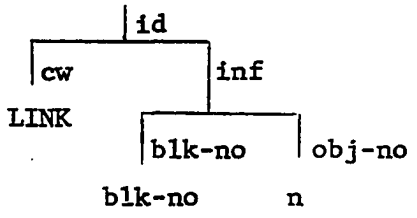
f. Formal parameter (PARA)



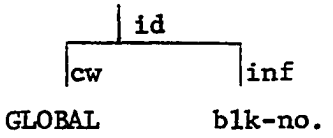


This is the control word of a procedure's formal parameter. In the initial state or before the procedure is executed, the inf is empty. After the procedure is called, the inf contains the information of its corresponding actual parameter and the calling statement. Remember, we have defined in the abstract syntax that the actual parameter list of its call statement or procedure call is in the O.S. The parameter linking process will insert a object number (obj-no), a parameter number (pn) and a block number (blk-no) into the inf to locate the actual parameter in the O.S. We have to have the object number of the call statement or procedure call in the O.S. We need to have the parameter number to locate this specific parameter from the actual parameter list. Also we need to have the block number of the calling statement in order to find the appropriate environment. Different calling statements will insert different contents into inf.

g. Link This is the control word and information field of a link identifier after this link statement is executed. The inf contains the blk-no and the obj-no of the Link statement.

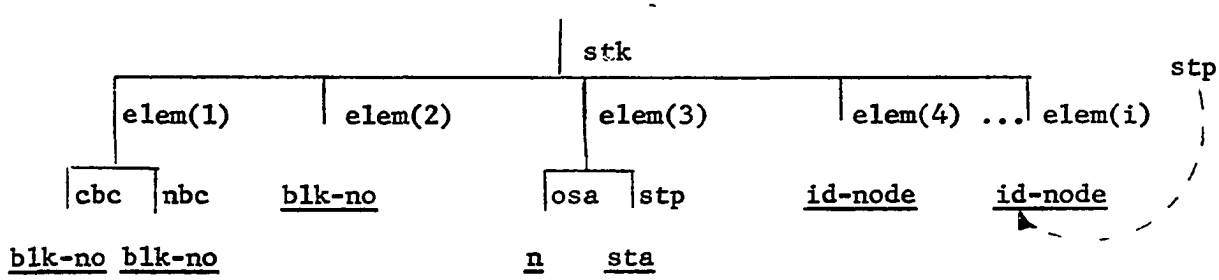


h. Global This is the control word of a global linked identifier. The inf contains the blk-no of the linked block.



3. Stack (stk)

The push-down stack is defined as a stack-word (stk-wd) list. The first three entries are reserved for storing block stack words (blk-stk-wd). They can be fetched and inserted randomly. The block stack words are information for block linking. Usually, the first blk-stk-wd always stores the current block number and nesting block number. The second blk-stk-wd stores a block number if we are entering this block or stores a procedure block number if we are entering this procedure block. The third blk-stk-wd stores current object string address (OSC) and current stack pointer (stp). These information are necessary to resume the original environment when the block is completed.



Other entries on the stack can be pushed down or popped up. These entries are id-nodes. Each id-node contains two components, a control word and a information field which we have discussed in the section of name table tree.

The push-down and pop-up operation are carried out by incrementing or decrementing the stack pointer (stp). This stack pointer is in the Headers.

4. Registers (r)

The registers contains six components which are object string counter (osc), stack pointer (stp), block counter (bc), space available list pointer (salp), limit counter (lc) and headers (hd).

The object string counter always points to the current executing object in the object string.

The stack pointer (stp) points to the top (empty) entry of the stack.

The block counter stores current executing block number.

The space available list pointer points to the top element of the space available list.

The limit counter contains an integer bound. In the initial state, this bound is set to 9.

The registers are for storage of block information.

For simplification, these components may be directly referred by following selectors:

s-osc(ξ) = s-osc(s-r(ξ))

s-stp(ξ) = s-stp(s-r(ξ))

s-bc(ξ) = s-salp(s-r(ξ))

s-lc(ξ) = s-lc(s-r(ξ))

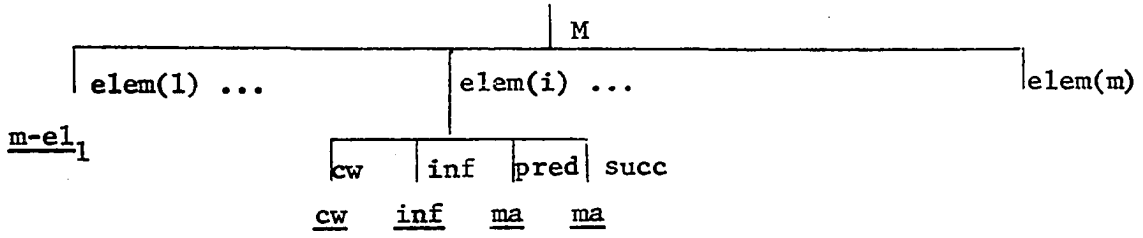
s-hd(ξ) = s-hd(s-r(ξ))

5. Memory (m)

Memory is a sequence of memory elements (m-el). Memory elements can form doubly linked lists. Each element contains four parts, they are control word (cw), information field (inf), predecessor (pred) and successor (succ).

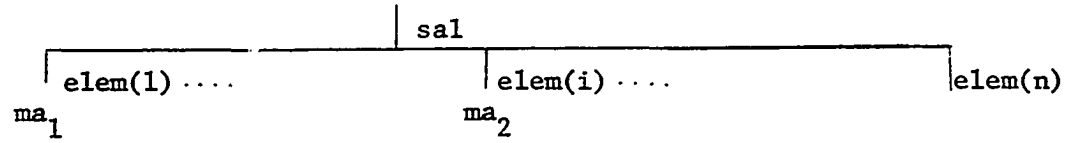
The memory control words are (Link to Substructure) LSS, STRING and EV (End of Vector). LSS indicates its inf contains a pointer which points to a substructure. STRING indicates its inf contains a string. And EV indicates this cell is the end of a vector, its inf is Ω. The pred can link to its predecessor and succ can link to its successor in the memory.

In the initial state all nodes of all memory elements are null (Ω).



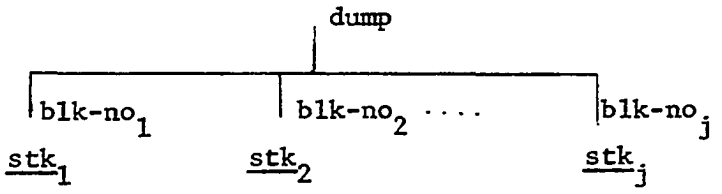
6. Space available list (sal)

Space available list is a sequence of memory addresses. In the initial state, the whole memory is linearly in this list. Memory assignment is by deleting memory address from the top of the list. Collecting free memory spaces is by adding their addresses on top of the list. The top is pointed by the salp in the header.



7. Dump (d)

Dump is simply a one level storage list for storing stacks. When we enter a nested block, the nesting block's stack is temporarily stored in dump. Stacks stored in dump are named by their own block numbers.



8. Control (c)

Control is a control tree. Please refer to (6) for detailed discussion.

B. Initial State

The initial state for any given program $t \in \overset{\Lambda}{\text{is-obj-string}}$ is

$$\mu_0(\langle \text{s-c:}\underline{\text{int-os}}(t)\rangle, \langle \text{s-osc:1}\rangle, \langle \text{s-stp:1}\rangle,$$
$$\langle \text{s-bc:}\Omega\rangle, \langle \text{s-salp:m}\rangle, \langle \text{s-lc:9}\rangle,$$
$$\langle \text{s-ntt:is-init-ntt}\rangle, \langle \text{s-stk:}\Omega\rangle,$$
$$\langle \text{s-sal: is-init-sal}\rangle, \langle \text{s-d :}\Omega\rangle,$$
$$\langle \text{s-m: } (\{\langle \text{ma:}\Omega\rangle \,||\, \text{is-ma(ma)}\}\langle \text{s-os:is-os}\rangle)$$

In the intital state, the object string counter (osc) is set to 1 which points to the first object of the O.S. The stack pointer (stp) points to the bottom of the stack. The block counter (bc) is set to Ω which means no block has been encountered. This limit counter (lc) is default to 9. The stack (stk) is empty. The dump (d) is empty. The memory elements are all empty. The object string (os) is a legal object string of a program.

Since no memory space is used, all memory elements are available for assignment. They are all in the initial space available list (init-sal).

$$\text{is-init-sal} = (\{\langle \text{i:i}\rangle \,||\, \text{is-a(i)},\text{is-ma(i)}\ i \leq M\})$$

where: M is the word number of the memory.

The space available pointer points the top of the list, so it is set to M.

In the initial state the initial name table tree (init-ntt) is prepared ready for execution. For different nodes in the name table tree, different entries have been inserted.

For block-link-node, the flag node is set empty, since no block has been executed. The fl is set to link the block branch of the leading block (or nesting block), and the bl is set to link the block branch of the following block (nested block) or Ω.

For id-nodes, their contents are based on their identifiers. For different identifiers of the program, their contents are shown below:

Identifier	Control word	Information field
procedure identifier	PROC	ptr: points to the first object of the procedure body is o.s.
label	LABEL	ptr: points to the location of this label in the o.s.
formal parameter identifier of initial	PARA	Ω
value statement (init-val-id)	DOS	pointer: points to this initial-value statement in the o.s.
All other identifiers	DNT	Ω

States ξ whose control part s-c(ξ) is Ω are <u>end-states</u>.

V. THE INTERPRETATION OF SYMBOL PROGRAMMING LANGUAGE

The Interpretation of SYMBOL Language defined an instruction schema int-os whose parameter is a SYMBOL program object string which we have defined in Abstract Syntax. The execution of this instruction int-os(t) simulates the execution of a program (t) by the SYMBOL system in terms of the abstract machine.

For convenient and better readability, abbreviations for the immediate components of a current state ξ are introduced. The left-hand side of the following list may always be replaced by the corresponding right hand side.

OS	s-os(ξ)	
NTT	s-ntt(ξ)	
STK	s-stk(ξ)	
R	s-r (ξ)	
SAL	s-sal(ξ)	
D	s-d(ξ)	
C	s-c(ξ)	
M	s-m(ξ)	
OSC	s-osc(ξ)	s-osc(s-r (ξ))
STP	s-stp(ξ)	s-stp(s-r (ξ))
BC	s-bc(ξ)	s-bc(s-r (ξ))
SALP	s-salp(ξ)	s-salp(s-r (ξ))
LC	s-lc(ξ)	s-lc(s-r (ξ))

If there cases for which the instruction definition is undefined, then this is implicated by an additional final line in the definition:

T → error

The following functions and instructions are not further specified:

1. push-down (t) =

   s-stk: μ(STK; <STP : t>)

   s-stp: STP + 1

2. int-mon-com (t,b) — Instruction which returns the result of monadic combination t in the environment of block b.

3. int-dya-com(t,b) — Instruction which returns the result of dyadic combination t in the environment of block b.

4. print(t) — Instruction which prints t.

5. output(m,t) — Instruction which outputs record t to terminal m.

The definition of instruction schema will be given in the following format:

(Ii) DEFINITION OF INSTRUCTION SCHEMA

where: list of abbreviation local to the definition

for: range of arguments of the schema printed

ref.: references

note: additional notes.

Any of these last four items may be omitted.

(I1) int-os =

is-'$\dagger$'(O_t) → Ω

T → s-osc:OSC + 1

s-c: int-os

int-obj(O_t)

where: O_t = elem(OSC, OS)

for: is-OS(OS)

note: OSC+1 means increment the OSC pointing to the next object;

$\dagger$ indicates the end of the o.s.

(I2) int-obj(t) =

is-blk-no(t) → int-blk(t)

is-proc-blk-no(t) → skip-to-end(t)

is-proc-hd(t) → null

is-stm(t) → int-stm(t)

is-blk-end(t) → int-blk-end(t)

is-proc-blk-end(t) → int-proc-blk-end(t)

is-if-exp(t) → push-down(μ_0(<s-cw : 'BOOL' >,

<s -inf: e >

e:int-exp(t,BC)

is-if-jump-ptr(t) → int-if-jump-ptr(t)

is-transf-ptr(t) → int-transfer(t)

for: is-object(t)

(I3) int-blk(t) =

is-EQUAL (BC,t) → null;

T → is <Ω>(BC) → enter-blk(t)

T → enter-nested-blk(t)

where: is-EQUAL (BC,t) test if BC equals t.

(I4) enter-blk(t) =

set-bc(t)

stack-I(μ_0(<s-cbc:t>,<s-nbc:BC>))

init-blk

save-stk

note: These four instructions may be executed in parallel as:

s-bc:t

s-stk: μ_0(s-elem(1):(μ_0(<s-cbc:t>,<s-nbc:BC>)))

s-stp:4

s-d: μ(D;<BC : STK>)

(I5) Save-stk =

s-d: μ(D; <BC : STK>)

(I6) init-blk =

s-stk : Ω

s-stp : 4

(I7) stack-I(t) =

s-stk:μ(STK;<s-elem(1): t >)

(I8) set-bc(t) =

s-bc : t

(I9) enter-nested-blk(t) =

enter-blk(t)

stack-III(μ_0(<s-osa:OSC>, <s-stp:STP>))

stack-II(BC)

note: This instruction can be modified as

s-c:enter-blk(t)

s-stk: μ(STK; <s-elem(2): BC >,

<s-elem(3): μ_0(<s-oas:OSC>, <s-stp:STP>)>)

(I10) stack-II(t) =

s-stk: μ(STK; <s-elem(2): t>)

(I11) stack-III(t) =

s-stk: μ(STK; <s-elem(3): t>)

(I12) skip-to-end =

is-proc-blk-end(O_t) → s-osc:OSC + 1

T → s-c: skip-to-end

s-osc:OSC + 1

where : O_t = elem(OSC, OS)

(I13) int-stm(t) =

is-assign-stm(t) → int-assign-stm(t)

is-goto-stm(t) → int-goto-stm(t)

is-call-stm(t) → int-call-stm(t)

is-dummy-stm(t) → int-dummy-stm(t)

is-comment-stm(t) → int-comment-stm(t)

is-init-val-stm(t) → int-init-val-stm(t)

is-ret-stm(t) → int-ret-stm(t)

is-link-stm(t) → int-link-stm(t)

is-sw-stm(t) → int-sw-stm(t)

is-input-stm(t) → int-input-stm(t)

is-output-stm(t) → int-output-stm(t)

for: is-stm(t)

(I14) int-assign-stm(t) =

is-exp(s-assign-elem(t)) → int-exp-assign-stm(t))

is-assign-struct(s-assign-elem(t)) → int-struct-assign-stm(t)

(I15) int-exp-assign-stm(t) =

assign

stack-assign-exp(int-exp(s-assign-elem(t),BS))

stack-recip-list(s-recip-list(t))

(I16) stack-recip-list(t) =

is-<>(t) → null

T → stack-recip-list(s-tl(t))

stack-recip(s-hd(t))

for: is-recip-list(t)

(I17) stack-recip(t) =

is-'LIMIT'(t) → push-down(μ_0(<s-cw:RECIP >,

<s-inf:LIMIT>))

is-recip(t) → push-down (μ_0(<s-cw:RECIP >,

<s-inf: R >)

R: int-recip (t,BC)

for: is-recip(t) v is-'LIMIT'(t)

note: LIMIT is the recipient for assigning limit counter (lc)

(I18) int-recip(t,b) =

is-id(t) → get-LHS-simp-add(t,b)

is-component(t) → get-sub-add(t,b)

is-proc-call(t) → int-recip(int-proc-call(t,b),b)

(I19) get-LHS-simp-add(t,b) =

is-LABEL(cw_t) → error

is-PROC (cw_t) → error

is-LINK(cw_t) → get-LHS-link-add(inf_t,b)

is-DNT(cw_t) → get-LHS-simp-add(t,inf_t)

is-LS(cw_t) → PASS:t.b.ntt

is-PARA(cw_t) → get-LHS-para-add(inf_t)

is-DOS(cw_t) → get-LHS-simp-add(t,b)

gen-os-val(t,b)

where: cw_t = s-cw.t.b.ntt(ξ)

inf_t = s-inf.t.b.ntt(ξ)

note: PASS = t.b.ntt passes a pointer which is t.b.ntt, it is not passing the contents which is pointed by this pointer.

(I20) get-LHS-link-add(t,b) =

is-prim-exp(exp_t) → gla-LHS-prim(exp_t,bc_t)

is-mon-exp(exp_t) v is-dya-exp(exp_t) → error

is-paren-exp(exp_t) → get-LHS-link-add(s-exp(exp_t),b)

where: exp_t = s-exp(s-obj-no(t).(OS))

bc_t = s-blk-no(t)

(I21) gla-LHS-prim(e,b) =

is-literal(e) → error

is-id(e) → get-LHS-simp-add(e,b)

is-component(e) → get-sub-add(e,b)

is-proc-call(e) → int-recip(int-proc-call(e,b),b)

is-LIMIT(e) → PASS:LIMIT

is-part-ref(e) → error

is-in-comp(e) → error

(I22) get-LHS-para-add(t) =

is-id(e_t) → get-LHS-simp-add(e_t, b_t)

is-component(e_t) → get-sub-add (e_t,b_t)

is-proc-call(e_t) → int-recip(int-proc-call(e_t,b_t),b_t)

where: e_t = s-pn(t)(s-obj-no(t).OS)

b_t = s-blk-no(t)

(I23) gen-os-val(t,b) =

is-switch-stm(stm_t) → gen-label-struct(t,b)

is-init-val-stm(stm_t) → gen-init-val(t,b)

where: stm_t = (s-inf(t))·OS

(I24) get-sub-add(t,b) =

ga-sub

reverse-stp

stack-comp-list(s-comp-list(t))

push-down(μ_0(<s-cw:'SUB-ADD'>,

<s-inf: a>))

a: get-LHS-simp-add(s-comp-id(t),b)

note: 'SUB-ADD' is a control word temporarily stored on stack to indicate that its inf is the id of a subscribed address.

(I25) stack-comp-list(t) =

is-<>(t) → push-down(μ_0(<s-cw:']'>, <s-inf:Ω>))

T → stack-comp-list(s-tl(t))

push-down(μ_0(<s-cw:'SUB'>,

<s-inf:s>))

s:int-exp(s-hd(t),BC)

for: is-number(s)

note: 'SUB' indicates inf contains an subscript of a component.

(I26) reverse-stp =

is-'SUB-ADD'(cw_{st}) → null

T → s-c:reverse-stp

s-stp:STP-1

where: cw_{st} = s-cw(elem(STP,STK))

(I27) ga-sub =

s-c:is-'DNT'(cw_a)

→ is-<> (inf_a) → gen-struct(sub_{stk},add_{stk})

T → error*

is-'LS'(cw_a) → scan-sub(sub_{stk},add_{stk})

s-stp:STP+2

where: add_{stk} = s-inf·elem(STP,STK)

sub_{stk} = s-inf·elem(STP+1,STK)

cw_a = s-cw(add_{stk}(ξ))

inf_a = s-inf(add_{stk}(ξ))

note: add_{stk} simply is a pointer and add_{stk} (ξ) is the content which is pointed by add_{stk}.

ref: (I24) and (I19)

*: error is caused by trying to alter the content when the address is referred.

(I28) gen-struct(s,a) =

s-c: scan-nullfield-sub(s,c_t)

enter-ntt-link(a,c_t)

s-salp: SALP-1

where: c_t = elem(SALP, SAL)·m

ote: c_t is only a pointer (not the content)

(I29) enter-ntt-link(a,c) =

s-a: μ_0(<s-cw:'LS'>,<s-inf:c>)

note: a is a pointer. This is equivalent to s-ntt: μ(NTT; a:μ_0(<s-cw:'LS'>,<s-inf:c>))

ref: (I27)

(I30) scan-nullfield-sub(s,a) =

s-c:is-'1'(s) → is-']'(cw_{new}) → PASS:a

reverse-stp

insert-ev(c_t)

insert-nf-succ(a,(t)

is-'SUB'(cw_{new}) → s-c:gen-sub-struct(inf,a)
new

insert-ev(c_t)

insert-nf-succ(a,c_t)

s-stp:STP+1

T → scan-nullfield-sub(s-1,c_t)

insert-nf-succ(a,c_t)

s-salp:SALP-1

where: cw_{new} = s-cw·elem(STP,ST)

inf_{new} = s-inf·elem(STP,ST)

c_t = elem(SALP, SAL)·m

(I31) gen-sub-struct(s,a) =

s-c: scan-nullfield-sub(s,c_t)

insert-sub-link(a,c_t)

s-salp: SALP-1

where: c_t = elem(SALP,SAL)·m

(I32) insert-sub-link(a,c) =

μ(s-a(ξ); <s-cw:LSS>, <s-inf:c>)

note: 'LSS' means link-to-substructure

(I33) insert-ev(c) =

μ(s-c(ξ); <s-cw:EV>, <s-succ:Ω>)

note: 'EV' indicates the end of vector

(I34) insert-nf-succ(a,c) =

μ(s-a(ξ); <s-cw:STRING>, <s-inf:Ω>, <s-succ:c>)

μ(s-c(ξ); <s-pred:a>)

(I35) scan-sub(s,a) =

is-<1>(s) → is-<STRING>(cw_a) → is-<]>(cw_{new}) → PASS:a
reverse-stp

T →
is-<>(inf_a) → s-c:gen-sub-struct(sub_{new}, a)

s-stp:STP + 1

T → error*

is-<LSS> (cw_a) → is-<]>(cw_{new}) → PASS:a

reverse-stp +

T → s-c:scan-sub(sub_{new},inf_a)

s-stp:STP+1

is-<EV> (cw_a) → is-<]> (cw_{new}) → s-c:PASS:a

reverse-stp

insert-ev(c_t)

s-salp:SALP-1

T → s-c:gen-sub-struct(sub_{new},a)

insert-ev(c_t)

insert-nf-succ(a,c_t)

s-stp:STP+1

s-salp:SALP-1

T → is-<EV> (cw_a) → s-c:scan-nullfield-sub(s,c_t)

insert-nf-succ(a,c_t)

s-salp:SALP-1

T → scan-sub(s-1,$succ_a$)

where: cw_a = s-cw·a(ξ)

inf_a = s-inf·a(ξ)

sub_{new} = s-inf·elem(STP,STK)

cw_{new} = s-cw·elem(STP,STK)

c_t = elem(SALP, SAL)·m

$succ_a$ = (s-succ·a(ξ))

note: *ref: (I27)

+Garbage collect routine will collect the deleted structure if a new assignment happens.

(I36) int-exp(t,b) =

is-prim-exp(t) → int-prim-exp(t,b)

is-mon-com(t) → PASS:μ_0(<s-cw:STRING>,<s-inf:int-mon-com(t,b)>)

is-dya-com(t) → PASS:μ_0(<s-cw:STRING>,<s-inf:int-dya-com(t,b)>)

is-paren-exp(t)→ int-exp(s-exp(t),b)

(I-37) int-prim-exp(t,b) =

is-literal(t) → PASS:μ_0(<s-cw:STRING>,<s-inf:t>)

is-id(t) → get-RHS-simp-add(t,b)

is-component(t) → pass: a(ξ)

a: get-sub-add(t,b)

is-proc-call(t) → int-exp(int-proc-call(t,BC),b)

is-<LIMIT>(t) → PASS:μ_0(<s-cw:LIMIT>,<s-inf:LC>)

is-part-ref(t) → int-part-ref(t,b)

is-in-comp(t) → int-in-comp(t,b)

(I-38) get-RHS-simp-add(t,b) =

is-<LINK>(cw_t) → get-RHS-link-add(inf_t,b)

is-<DOS>(cw_t) → get-RHS-simp-add(t,b)

gen-os-val(t,b)

is-<GLOBAL>(cw_t) → get-RHS-simp-add(t,inf_t)

is-<PARA>(cw_t) → get-RHS-os-add(inf_t)

T → PASS: nt_t

where: nt_t = t·b·ntt(ξ)

cw_t = s-cw(nt_t)

note: T condition is obtained if cw_t is LABEL, PROC, LS, and DNT.

(I-39) get-RHS-link-add(t,b) = int-exp(exp_t,b_t)

where: exp_t = s-exp·((s-obj-no(t))·OS)

b_t = s-blk-no(t)

ref: p. 27 Link.

(I40) get-RHS-os-add(t) = int-exp(e_t,b_t)

where: e_t = (S-pn(t))·((s-obj-no(t)) (OS))

b_t = s-blk-no(t)

ref: p. 26 formal parameter

(I41) stack-assign-exp(t) =

is-<STRING v LIMIT v DNT>(cw_t)

→ push-down(μ_0(<s-cw: DATA>,<s-inf:inf_t>))

is-<LS v LSS>(cw_t) → stack-struct(t)

T → error

where: cw_t = s-cw(t)

(I42) stack-struct(t)

s-c:fetch-struct(inf_t,c_t)

push-down(μ_0(<s-cw: ⊜>,<s-inf:Ω>))

start-level-ptr(c_t)

s-salp:SALP-1

where: c_t = elem(SALP, SAL)·m

inf_t = s-inf·t(ξ)

(I43) start-level-ptr(t) =

s-t: μ_0(<s-cw:INIT>, <s-pred:Ω>)

note: insert 'INIT' into the level pointer.

(I44) fetch-struct(t,s)

is-<EV>(cw_t) → int-ev-source(s)

is-<LSS>(cw_t) → int-link-source(t,s)

T → fetch-struct($succ_t$,s)

pushdown(μ_0(<s-cw:DATA>, <s-inf:inf_t>))

where: cw_t = s-cw·t(ξ)

inf_t = s-inf·t(ξ)

$succ_t$ = s-succ-t(ξ)

note: t(ξ) is a pointer which links to an entry in the memory.

(I45) int-link-source(t,s) =

s-c: fetch-struct(inf_t,c_t)

pushdown(μ_0(<s-cw:<>, <s-inf:Ω>))

insert-level-ptr($succ_t$,s,c_t)

s-salp: SALP-1

where: inf_t = s-inf·t(ξ)

c_t = elem(SALP, SAL)·m

$succ_t$ = s-succ·t(ξ)

(I46) insert-level-ptr(t,s,c) =

s-m: μ(M;s:(<s-succ:c>),

c:(<s-cw:DATA>, <s-inf: t >, <s-pred:s>

<s-succ:Ω>))

(I47) int-ev-source(t,s) =

is-<INIT>(cw_s) → delete(s)

pushdown(μ_0(<s-cw:⊛>,

<s-inf:Ω>))

T → fetch-struct(inf_s, $pred_s$)

pushdown(μ_0(<s-cw: >,

<s-inf:Ω>))

where: inf_s = s-inf·s(ξ)

$pred_s$ = s-pred·s(ξ)

cw_s = s-cw·s(ξ)

(I48) delete(s) =

is-<>(s) → null

T → s-c: delete($succ_s$)

s-sal:μ{SAL;elem(SALP + 1) : s }

s-SALP: SALP + 1

where: $succ_s$ = s-succ·S(ξ)

(I49) delete-sub-struct(t) =

is-<LSS>(cw_t) → delete-sub-struct($succ_t$)

delete-sub-struct(inf_t)

is-<DATA>(cw_t) → delete-sub-struct($succ_t$)

delete(t)

is-<EV>(cw_t) → delete(t)

where: cw_t = s·cw·t(ξ); inf_t = s-inf-t(ξ):

$succ_t$ = s-succ·t(ξ)

(I50) assign =

find-as-recip

scan-as-el

(I51) scan-as-el =

is-<⊜> (cw_t) → back-up-stp

is-<LINK EXP V DATA > (cw_t) → s-stp: STP-1

where: cw_t = s-cw·elem(STP-1,STK)

(I52) back-up-stp =

is-<∈> (cw_s) → s-stp:STP-1

T → s-c:back-up-stp

s-stp:STP-1

where: cw_s = s-cw·elem(STP-1,STK)

(I53) find-as-recip =

is-<RECIP> (cw_t) → assign

null-recip(STP-1

assign-recip(inf_t)

is-< > (cw_t) → s-c: find-as-recip

s-stp: STP-1

T → null

where : cw_t = s-cw·elem(STP-1), STK)

inf_t = s-inf·elem(STP-1,STK)

(I54) null-recip(t)

s-STK: μ(STK; t : <s-cw: Ω>, <s-inf:Ω>)

(I55) assign-recip(t) =

is-<LS v LSS> (cw_t) → assign-field(t)

look-as

delete-sub-struct(inf_t)

is-$\prec$DNT$\succ$ (cw_t) → assign-int-field(t)

look-as

is-$\prec$DATA$\succ$ (cw_t) → assign-m-field(t)

look-as

where: cw_t = s-cw·t(ξ)

inf_t = s-inf·t(ξ)

(I56) look-as =

is-$\prec\ \succ$ (cw_t) → s-c:look-as

s-stp:STP+1

T → null

where: cw_t = s-cw·elem(STP·STK)

(I57) assign-m-field(t)

is-$\prec$ⓔ$\succ$ (cw_s) → assign-struct(t)

is-$\prec$DATA$\succ$ (cw_s) → assign-simp-var(t)

is-$\prec$LINK EXP$\succ$ (cw_s) → assign-link(t)

where: cw_s = s-cw·elem(STP,ST)

ref: (I115)

(I58) assign-simp-var(t)

s-m: μ(M; t: ($\prec$s-cw: DATA$\succ$, $\prec$s-inf:$inf_s\succ$,

$\prec$s-pred:Ω$\succ$, $\prec$s-succ:Ω$\succ$))

where: inf_s = s-inf·elem(STP,STK)

(I59) assign-nt-field(t) =

is-$\prec$ⓔ$\succ$ (cw_s) → assign-nt-struct(t)

is-<DATA> (cw_s) → assign-nt-simp-var(t)

is-<LINK EXP> (cw_s) → assign-link(t)

where: cw_s = s-cw·elem(STP∘STK)

(I60) assign-nt-simp-var(t) =

s-ntt: μ(NTT; t :(<s-cw: DNT>, <s-inf:inf_s>))

where: inf_s = s-inf·elem(STP,STK)

(I61) assign-struct(t) =

s-c: creat-struct(c_1,c_2)

est-link(t,c_1)

s-m: μ(M;c_2:<s-pred:Ω>)

where: c_1 = elem(SALP, SAL)·m

c_2 = elem(SALP-1,SAL)·m

(I62) est-link(t,c) =

s-m: μ(M;t:(<s-cw:LSS>, <s-inf:c>),

c:(<s-cw:Ω>, <s-inf:Ω>, <s-pred:Ω>,

<s-succ:Ω>))

s-salp: SALP-1

(I63) assign-nt-struct(t) =

s-c: creat-struct (c_1,c_2)

est-nt-link(t,c_1)

s-m: μ(M; c_2:<s-pred:Ω>)

where: c_1 = elem(SALP, SAL)·m

c_2 = elem(SALP-1,SAL)·m

(I64) est-nt-link(t,c) =

s-ntt:μ(NTT; t:(<s-cw:LS>,<s-inf:c>))

s-m:μ(M;c:(<s-cw:Ω>,<s-inf:Ω>,

<s-pred:Ω>,<s-succ:Ω>))

s-salp:SALP-1

(I65) creat-struct(a,s) =

s-c:is-<$\circledast$>(cw_s) → creat-struct(a,s)

is-< >(cw_s) → LG-struct(a,s)

is-< >(cw_s) → RG-struct(a,s)

is-<$\circledast$>(cw_s) → RSG-struct(a,s)

is-<DATA>(cw_s) → string-struct(a,s,P_s)

s-stp:STP+1

s-salp:SALP-1

where: cw_s = s-cw·elem(STP,STK)

P_s = elem(STP,STK)

(I66) LG-struct(a,s) =

s-c:creat-struct(c_1,c_3)

set-level-ptr(s,c_2,c_3)

est-link(a,c_1)

when c_1 = elem(SALP, SAL)·m

c_2 = elem(SALP-1,SAL)·m

c_3 = elem(SALP-2,SAL)·m

(I67) set-succ(a,c)

s-m:μ(M;a:<s-succ:c>,

c:(<s-cw:Ω>,<s-inf:Ω>,<s-pred:a>,

<s-succ:Ω>))

s-salp:SALP-1

(I-68) set-level-ptr(s,b,c)

s-m:μ(M;s:(<s-cw:Ω>,<s-inf:b>,<s-succ:c>),

c: <s-pred:s>)

s-salp:SALP-1

(I69) RG-struct(a,s) =

creat-struct(a_a, s_a)

blank(s,s_a)

set-ev(a)

where: s_a = s-pred-s(ξ)

a_a = s-inf·s_a

(I-70) set-ev(a) =

s-m:μ(M;a:<s-cw:EV>)

(I71) blank(s,p)

s-m: μ(M; P:<s-succ:Ω>)

s-c: delete(s)

(I72) RSG-struct(a,s) =

delete(s)

set-ev(a)

(I73) string-struct(a,s,p)

creat-struct(c,s)

set-string(a,c,p)

where: c = elem(SALP, SAL)·m

(I74) set-string(a,c,p) =

s-m:μ(M; a:(<s-cw:cw_p>,<s-inf:inf_p>,<s-succ:c>),

c:(<s-cw:Ω>,<s-inf:Ω>,<s-pred:a>,

<s-succ:Ω>))

s-salp:SALP-1

where: cw_p = s-cw·p

inf_p = s-inf·p

(I75) int-struct-assign-stm(t) =

assign

stack-assign-struct(s-assign-struct(t))

stack-recip-list(s-recip-list(t))

(I76) stack-assign-struct(t) =

pushdown(μ_0(<s-cw:⊋>,<s-inf:Ω>))

stack-assign-as-list(s-as-list(t))

stack-as(s-as(t))

pushdown(μ_0(<s-cw:⊊>,<s-inf:Ω>))

(I77) stack-as(t) =

is-<>(t) → null

T → stack-field-list(t)

(I78) stack-field-list(t) =

is-<>(t) → null

T → stack-field-list(s-field-tl(t))

stack-assign-exp(int-exp(s-field-hd(t),BC))

(I79) stack-assign-as-list(t) =

is-< >(t) → null

T → stack-assign-as-list(s-as-tl(t))

stack-assign-as-pair(s-as-hd(t))

(I80) stack-assign-as-pair(t) =

stack-as(s-as(t))

stack-as-struct(s-as-struct(t))

(I81) stack-as-struct(t) =

pushdown(μ_0(< s-cw: > > , < s-inf:Ω >))

stack-assign-as-list(s-as-list(t))

stack-as(s-as(t))

pushdown(μ_0(< s-cw: < > , < s-inf:Ω >))

(I82) int-proc-call(t,b) = check-proc-ret(int-procedure(t,b))

(I83) check-proc-ret(t) =

is-< > (t) → error

T → null

(I84) int-procedure (t,b) =

enter-proc-blk(P)

link-para-list(P,1)

check-recursion(P)

stack-act-para-list(t)

P:check-proc-id(get-RHS-simp-add(s-id(t),b))

(I85) check-proc-id(t) =

is-<PROC>(cw_t) → PASS: inf_t

T → error

where: cw_t = s-cw•t(ξ)

inf_t = s-inf•t(ξ)

(I86) stack-act-para-list(t) =

stack-act-para(t,n)

n: count-act-para(s-act-para-list(t))

(I87) count-act-para(t) =

is-<>elem(1,t) → error

T → count-para(1,t)

note: no nonparameter procedure call is allowed

(I88) count-para(n,t) =

is-<>elem(n+1,t) → PASS: n

T → count-para(n+1,t)

(I89) stack-act-para(t,n) =

is-<1>(n) → pushdown(μ_0(<s-cw:ACT PARA>,

<s-inf:μ(<s-obj-no:t>,

<s-pn: n>,<s-blk-no:BC>)))

T → stack-act-para (t,n-1)

pushdown(μ_0(<s-cw:ACT PARA>,

<s-inf:μ_0(<s-obj-no:t>,

<s-pn:n>,

<s-blk-no:BC>)))

(I90) check-recursion(t) =

is-<BIU> ($flag_t$) → recursion

T → s-$flag_t$(ξ): BIU

where: proc-body-address = t

proc-blk-no = elem(t,OS) =bc_t

proc-nt = s-bc_t·ntt(ξ) = nt_t

$flag_t$ = s-flag·s-blk-link·nt_t

note: recursion will call software to step in.

BIU means block in use

(I91) link-para-list(t,n) =

is-<PARA> (cw_f) → is-<ACT-PARA> (cw_s) → s-c:link-para-list(t,n+1)

s-stp:STP-1

s-ntt:μ(NTT;P_f:<s-inf:

inf_s>)

T → error

is-<> (cw_f) → is-<ACT-PARA> (cw_s) → error

T → null

where: add = t·os body address

bc_t = add(ξ) = elem(t,OS) proc-blk-no

nt_t = bc_t·NTT procedure's name table

$head_t$ = elem(t+1, OS) procedure head

$para_f$ = elem(i, $head_t$) . ith formal parameter

P_f = $para_f$·nt_t

cw_f = s-cw·P_f

cw_s = s-cw·elem(STP-1,STK)

inf_s = s-inf·elem(STP-1,STK)

(I92) enter-proc-blk(p) =

s-c: enter-nested-blk(bc_p)

stack-III(OSC,STP)

stack-II(bcp)

s-osc: P+1

where: bc_p = p·OS

(I93) int·blk-end =

is-<>(bc_t) → STOP

T → s-bc: bc_t

s-stk:bc_t(D)

s-stp:s-stp·elem(3,bc_t(D))

where: bc_t elem(1,STK)

note: stop means complete execution

(I94) int-proc-end =

back

int-blk-end

clear-flag

(I95) clear-flag =

s-ntt:μ(NTT;f:Ω)

where: f = s-flag·s-blk-link.BC·ntt

(I96) back =

s-osc:s-osa(elem(3,STK))

(I97) int-ret-stm(t) =

is-<>(exp_t) → PASS:Ω

int-proc-end

T → PASS:e

int-proc-end

e: int-exp(exp_t, BC)

where: exp_t = s-ret-exp(t)

(I98) int-goto-stm(t) =

match-block(1)

1: int-label(int-prim-exp(s-goto-destn(t),BC))

(I99) int-label(t) =

is-<LABEL>(cw_t) → PASS: inf_t

T → error

where: cw_t = s-cw·t(ξ) inf_t = s-inf·t(ξ)

(I100) match-block(1) =

is-(BC = bc_1) → s-OSC:1

T → is-<>(BC) → STOP

T → match-block(1)

int-blk-end

clear-flag

where: bc_1 = elem(1,OS) ; is-(BC = bc_1) = is-EQUAL(BC,bc_1)

(I101) int-call-stm(t) = int-proc-call(s-proc-call(t))

(I102) int-dummy-stm(t) = null

(I103) int-comment-stm(t) = print(t)

(I104) gen-init-val(t,b) =

is-string(val_t) → gen-init-string(t,b)

is-data-struct(val_t) → gen-init-data-struct(t,b)

where: val_t = s-init-val(elem(s-inf(t),OS))

(I105) int-init-val-stm(t) =

s-ntt:μ(NTT; i_t:<s-cw:DOS>,<s-inf:OSC>)

where: i_t = s-init-val-id(t)·BC·ntt

(I106) gen-init-string(t,b)

s-ntt: μ(NTT; id_t: μ_0(<s-cw:DNT>,<s-inf:s_t>))

where: id_t = t·b·ntt

$string_t$ = s-init-val(elem(inf_t,OS))

inf_t = s-inf(t)

(I107) gen-init-data-struct(t,b) =

assign-nt-struct(P_t)

back-up-stp

stack-data-struct($struct_t$)

where: inf_t = s-inf(t) $struct_t$ = s-init-val(elem(inf_t,OS))

P_t = t·b·ntt

(I108) stack-data-struct(t) =

push-down(μ_0(<s-cw:⋑>,<s-inf:Ω>))

stack-data-ds-pair-list(s-ds-list(t))

stack-ds(s-ds(t))

pushdown(μ_0(<s-cw:⋐>,<s-inf:Ω>))

(I109) stack-ds(t) =

is-<>(t) → null

T → stack-string-fd-list(t)

(I110) stack-string-fd-list(t) =

is-<>(t) → null

T → stack-string-fd-list(s-sf-tl(t))

pushdown(μ_0(<s-cw:DATA>,

<s-inf: s-sf-hd(t)>))

(I111) stack-data-ds-pair-list(t) =

is-<>(t) → null

T → stack-data-ds-pair-list(s-ddp-tl(t))

stack-data-ds-pair(s-ddp- hd(t))

(I112) stack-data-ds-pair(t) =

stack-ds(s-ds(t))

stack-da-struct(s-data-struct(t))

(I113) stack-da-struct(t) =

pushdown(μ_0(<s-cw:>> , <s-inf:Ω>))

stack-data-ds-pair-list(s-ds-list(t))

stack-ds(s-ds(t))

pushdown(μ_0(<s-cw:<>,<s-inf:Ω>))

(I114) int-link-stm(t) =

assign

pushdown(μ_0(<s-cw: LINK-EXP>,

<s-inf:(<s-blk-no:BC>,<s-obj-no:OSC>)>))

stack-recip-list(s-recip-list(t))

(I115) assign-link(t) =

s-ntt:μ(NTT; t : (<s-cw:LINK>,<s-inf:inf_s>))

where: inf_s = s-inf(elem(STP,STK))

ref: (I57), (I59)

(I116) int-sw-stm(t) =

s-ntt:μ(NTT; i_t:μ_0(<s-cw:DOS>,<s-inf:OSC>))

where: i_t = s-sw-id(t)·ntt

(I117) gen-label-struct(t,b) =

assign-nt-struct(n_t)

back-up-stp

stack-label-struct($struct_t$,b)

where: $struct_t$ = s-sw-struct(elem(inf_t,OS))

inf_t = s-inf(t)

n_t = t·b·ntt

(I118) stack-label-struct(t,b) =

pushdown(μ_0(<s-cw:⊵>,<s-inf:Ω>))

stack-label-ls-pair-list(s-label-list(t),b)

stack-ls(s-ls(t),b)

pushdown(μ_0(<s-cw:⊴>,<s-inf:Ω>))

(I119) stack-ls(t,b) =

is-< >(t) → null

T → stack-id-list(t,b)

(I120) stack-id-list(t,b) =

is-< >(t) → null

T → stack-id-list(s-id-tl(t))

pushdown(i_t)

where: i_t = (s-hd(t))·b·NTT

for: is-label(s-hd(t))

(I121) stack-label-ls-pair-list(t,b) =

is-<>(t) → null

T → stack-label-ls-pair-list(s-llp-tl(t),b)

stack-label-ls-pair(s-llp-hd(t0,b)

(I122) stack-label-ls-pair(t,b) =

stack-ls(s-ls(t),b)

stack-la-struct(s-label-struct(t),b)

(I123) stack-la-struct(t,b) =

pushdown(μ_0(<s-cw:>>,<s-inf:Ω>))

stack-label-ls-pair-list(s-label-list(t),b)

stack-ls(s-ls(t),b)

pushdown(μ_0(<s-cw:<>,<s-inf:Ω>))

(I124) int-if-jump-ptr(t) =

s-c: jump(s_t, t)

test(s_t)

s-stp: STP-1

where: s_t = elem(STP-1,STK)

(I125) test(t) =

is-<0 v 1>(inf_t) → null

T → error

where: inf_t = s-inf(t)

note: 0 is false, 1 is true.

(I126) jump(l,t) =

is-< 1 > (inf_e) → null

T → s-OSC:a_t

where: inf_e = s-inf(e)

a_t = s-jump-prt(t)

note: s-OSC:a_t set the object string counter to the object which will be executed next. We can assume the incrementing of OSC is over-ride.

(I127) int-transfer(t) =

s-OSC: a_t

where: a_t = s-transf-ptr(t)

(I128) int-input-stm(t) =

is-<INPUT STRING> (id_t) → int-input-string-list(t)

is-<INPUT> (id_t) → int-input-list(t)

where: id_t = s-imput-id(t)

(I129) int-input-list(t) =

is-< > ($list_t$) → null

T → int-input-list(s-recip-tl($list_t$),m)

int-input(s-recip-hd($list_t$),m)

m: int-exp(s-input-term((t))

where: $list_t$ = s-recip-list(t)

for: is-terml(m)

(I130) int-input(t,m) =

assign

stack-input(m)

stack-recip(t)

(I131) stack-input(m)

is-data-struct(t_m) → stack-data-struct(t_m)

T → pushdown(μ_0(<s-cw:DATA>, <s-inf:t_m>))

where: t_m is the input record supplied from terminal m.

(I132) int-input-string-list(t) =

is-<> ($list_t$) → null

T → int-input-string-list(s-recip-tl($list_t$),m)
int-input-string(s-recip-hd($list_t$),m)
m: int-exp(s-input-terml(t))

where: $list_t$ = s-recip-list(t)

for: is-input-terml(m)

(I133) int-input-string(t,m) =

assign
stack-input-string(m)
stack-recip(t)

(I134) stack-input-string(m) =

pushdown(μ_0(<s-cw: DATA>, <s-inf:t_m>))

where: t_m is the input record stored in the input buffer of terminal m.

(I135) int-output-stm(t) =

output(m,t)
m:int-exp(s-out-terml(t),BC)
stack-output-list(s-output-exp-list(t))

(I136) stack-output-list(t) =

is-<>(t) → null

T → stack-output-list(s-exp-tl(t))

stack-assign-exp(e)

e:int-exp(s-exp-hd(t),BC)

(I137) int-in-comp(t,b) =

in-test(c_c)

c: get-sub-add(s-in-comp(t),b)

where: $c_c = c(\xi)$

(I138) in-test(c)

is-<>(c) → PASS: 0

T → PASS: 1

(I139) int-part-ref(t,b) =

is-<>(idx_t) → int-simp-part-ref(t,b)

T → int-comp-part-ref(t,b)

where: idx_t = s-part-ref-idx(t)

(I140) int-simp-part-ref(t,b) =

pass-part-ref(p,d,l)

l: int-exp(s-part-ref-length(t),b)

d: int-exp(s-part-ref-bound(t),b)

p: get-RHS-simp-add(S-part-ref-id(t),b)

(I141) int-comp-part-ref(t,b) =

pass-part-ref(p_t,d,l)

l: int-exp(s-part-ref-length(t),b)

d: int-exp(s-part-ref-bound(t),b)

p: get-sub-add(s-part-ref-id(t)s-part-ref-idx(t),b)

where: $P_t = p(\xi)$

(I142) pass-part-ref(p,d,l) =

PASS: r_p

where: r_p is the string taken out of p which starts from d^{th} character and contains l characters.

VI. CONCLUSION

One of the objectives of this description is to provide a conceptual description of the SYMBOL IIR computer system, especially focusing on its Central Processor (CP). In this description, all the execution of the system which is handled by the CP is faithfully implemented in the interpretation. Execution which is handled by other processors is purposely simplified.

It should be noted that certain differences exist between the SYMBOL system described here and the SYMBOL system as it actually exists. These differences need defending. The fundamental difference between the real computer and the conceptual description is that the latter is defined without concern for the word size in the computer. In the description, each node of a tree represents one logical entry without regarding for its size. The reasons for this assumption are:

(1) The concept of the abstract syntax is based on the existance of another translator which takes care of the parsing, scanning, segmenting and eliminating of unnecessary elements. Only the relevent program structure is concerned with the abstract syntax.

(2) In the SYMBOL system, memroy management is automatically handled by the memory Controller and are transparent to the actions of the CP.

The fundamental structure of the memory, i.e., the ability to form a doubly linked list, has, however, been preserved in the description.

In order to clarify the ideas, in the abstract machine, stack, name table tree and object string were defined seperate from the memroy.

But because of this arrangement, an additional dump had to be defined and some extra pointers had to be added.

In the SYMBOL system, when a new block is entered before the old block is exited, the old block's stack is simply left untouched in the memory. In the abstract machine, the stack is defined independently of the memory and only one stack exists. In the interpretation, when a new block is entered before the old block is exited, it has to provide a temporary storage for storing the old block's stack. Consequently a dump is implemented. This gives a clearer picture of the relation between a block and its stack.

A detailed description could have been done. The Reference Processor of the Central Processor has been actually defined elsewhere. Because of the complication of such a description, an unfamiliar reader finds it very hard to understand and to follow. Except for documentation, it is not practical as an introduction for the system.

This description was made after the SYMBOL system was already completed. If we apply a similar mechanism, we can also define a formal description of a new system, which will formalize and improve the design procedure of computing systems. For this purpose, newer languages and more efficient techniques have to be developed not only for description of programming language semantics but also for defining computing systems.

When the work started in late 1971, no one had tried to use the VDL to describe a processor. Later in the summer of 1972, a description of a mini-computer PDP-8 was reported (10), but no attempt

of describing a computer as complicated as SYMBOL was published.

The second objective was to test the applicability of the Vienna Definition Language (VDL) for describing complex computer systems. The result is positive. The Vienna Definition Language shows its strength in its powerful tools for the selection of tree components, the construction of new trees from their components and assignment of new values to vertices of trees. It is particularly suitable to describe SYMBOL. But because of the abstract machine is defined in such a way, sometimes composite selectors are too complicated to follow. This weakness is due to the definition of the abstract machine not to the VDL.

This paper demonstrates the availability of VDL for real applications. It also demonstrates the versatility of VDL for describing a processor in different levels. In order to let the reader gain more insight and philosophy of the designing of SYMBOL system, it is necessary to supply a document of the SYMBOL system which emphasizes the concept and logic structure but in a simplified and concise form. It is hoped that this paper has accomplished that object.

Some additional work might have been done concerning the translation between the actual source program and its corresponding abstract syntax, it can be done easily. The description of the syntax parsing and translation can be simply transformed from their corresponding hardware circuits in the Translator. A detailed description of SYMBOL might be continued as which has been done on the RP. This detailed description can be treated as a standard document of SYMBOL. Also, the VDL can be

applied in implementing the Translator of translating other programming languages into SYMBOL object string.

VII. BIBLIOGRAPHY

1. McCarthy, J. "Towards A Mathematical Science of Computation." Proc. IFIP Congress, 1962. Amsterdam: North Holland Publ. Co., 1963.

2. Naur, Peter, Editor, "Revised report on the algorithmic language ALGOL 60." Comm. ACM, 6, No. 1 (January, 1963), 1-17.

3. Landin, P. J. "A formal description of Algol 60." Proc. IFIP working conference on Formal Language Description Languages. Amsterdam, North Holland Publ. Co., 1966.

4. Landin, P. "A correspondence between ALGOL 60 and church's Lambda-Notation." Comm. ACM, 8, No. 2, (Feb., 1965) 89-101.

5. Bohm, C. "The CUCH as a formal and descriptive language." IFIP Working Conf., Baden, Sept. 1964.

6. Church, A. "The calculi of lambda-conversion." Am. Math. Studies, 6, 1941.

7. Wirth, Niklaus and Helmut Weber. "EULER: A Generation of ALGOL, and its Formal Definition." Comm. ACM, 9, No. 1, (Jan., 1966) 13-23.

8. Neuhold, E. J. "The formal description of programming languages. IBM SYST J. 10, No. 2 (1971), 86-112.

9. Lucas, P., P. Lauer and H. Stigleitner. "Method and notation for the formal definition of programming languages." IBM Laboratory Vienna, Technical Report TR 25.087, June 28, 1968.

10. Lee, John A. N. Computer Semantics. New York: Van Nostrand Reinhold Company, 1972.

11. Wegner, Peter. "The Vienna Definition Language." Comp. Surveys, 4, No. 1 (1972), 5-63.

12. Chesley, Gilman D. and William R. Smith. "The Hardware-Impremented High-Level Machine Language for SYMBOL." Proceedings SJCC, 1971.

13. Lucas, P. and K. Walk. "On the Formal Description of PL/1." Annual Review in Automatic Programming, 6, No. 3, (1969), 105-181.

14. Smith, William R., Rice, R., Chesley, G., Laliotis, T., Lundstrom, S., Calhoun, M., Gerould, L., Cook, T. "SYMBOL: A large experimental system exploring major hardware replacement of software." Proc. of AFIPS 1971 SJCC Vol. 38.

15. Richards, Hamilton, Jr. "SYMBOL Programming Reference Manual." Cyclone Computer Laboratory, Iowa State University, Ames, Iowa, 1971.

16. "TRANSLATOR logic description, EM0062." Cyclone Computer Laboratory, Iowa State University, Ames, Iowa, ca. 1969.

17. "Instruction Sequencer Logic Description EM0063." Unpublished Internal Report. Cyclone Computer Laboratory, Iowa State University, Ames, Iowa, ca. 1969.

18. "Reference Processor, EM0067." Unpublished Internal Report, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa, ca. 1969.

19. "IS Flow Chart." Unpublished Internal Report, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa, ca. 1969.

20. "RP Flow Chart." Unpublished Internal Report, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa, ca. 1969.

21. Mullery, A. P., R. F. Schauer, and R. Rice. "ADAM - a problem-oriented SYMBOL processor." IBM Research Paper RC-840. IBM Thomas J. Watson Research Center, Yorktown Heights, New York, December 20, 1962.

VIII. ACKNOWLEDGEMENTS

The author wishes to express his thanks to Dr. A. V. Pohm for his encouragement and continuous support and to Dr. C. T. Wright who read this paper since its early stages and made many constructive comments and guidance.

The author also expresses his appreciation to Dr. R. J. Zingg, Dr. R. M. Stewart, H. Richards, Jr. and R. Luckeroth for their comments and suggestions.

This work was supported by the SYMBOL project at Iowa State University.

IX. APPENDIX I

SYMBOL IIR INTERNAL CODE SET

	8	9	A	B	C	D	E	F
0	INPUT	BLOCK	TO	FROM		ADDRESS TO NT ENTRY	LINK TO SIMPLE VARIABLE	NUMERIC + +
1	OUTPUT	LOOP	DATA	BY	AG		LINK TO STRUCTURE	NUMERIC + -
2	DISABLE	ON		THRU	IG			NUMERIC - +
3	ENABLE	PROCEDURE	STRING	FOR	FT		LINK TO LABEL	NUMERIC - -
4	GOBAL	IF	EX	WHILE	FF	DIRECT PARAMETER	LINK TO DATA IN NT	NUMERIC TRUE ZERO
5	SWITCH	GO	EM EMpirical	THEN IF FALSE	FR	COMPLEX PARAMETER		STRING START
6	NOTE	PAUSE	CALL CONTINUE THAN	ELSE	FL	RETURN JUMP	LINK TO TEMPORARY DATA	STRING END
7	RETURN	SYSTEM	TRAP	END	FD	TRANSFER		END VECTOR
8	BEFORE	LIMITED	FALSE	IN	SA	PARAMETER RETURN	LINK TO STRUCTURE FIELD	

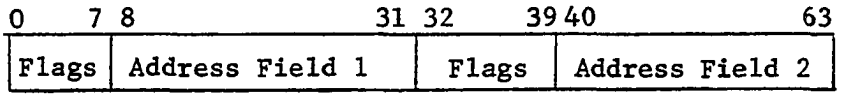
SYMBOL IIR INTERNAL CODE SET

	8	9	A	B	C	D	E	F
9	SAME	ABS Absolute	TRUE	NAME	SO	SOURCE POINTER		
A	AFTER	LTE Less Than or Equal	*	:	SI	INTEGER- IZE	LINK TO FIELD(IN)	
B	NOT	GTE Grtr Than or Equal	+	;	SD	NEGATE	LINK TO STRUCTURE (IN)	
C	AND	LESS	,	LIMIT	DE	|	LINK TO SUB- STRUCTURE	LEFT GROUP
D	OR	NEQ Not Equal	-	EQUAL	DS	]		LEFT SUPER GROUP
E	JOIN	GREATER		FORMAT	DL			RIGHT GROUP
F	MASK	→	/	INTERRUPT	ST	←		RIGHT SUPER GROUP

CHARACTER 0 AND 4 OF OBJECT STRING ONLY

X. APPENDIX II

NAME TABLE CONTROL WORD



Block Control Word (Bit 1 = 1)

Flag Bits

0 - Control Word

1 - Block Start

2 - Block End

3 -

4 - Privileged Procedure

5 - Global Linking Done

6 - Forward Link Active

7 - Backward Link Active

37 - Block In Use

38 - Block Recursed

Address Field 1

Forward Link (threads all blocks in the program).

Address Field 2

Backward Link (threads nested blocks only).

Identifier Control Word (Bit 1 = 0)

Flag Bits

0 - Control Word

1 - Block Start

2 - Block End

3 -

See Table Below

4 -

5 - Data In Object String

6 - Structured Data/Field

7 - Extended Entry

32 - ON Enabled Indicator

34 - Label Indicator

35 - Procedure Indicator

36 - Parameter Indicator

37 - ON Reference

Address Fields

Variable

Field 1 - Start Address

Field 2 - Current Address

Variable With ON

Field 1 = Start Address

Field 2 = Link to ON Block

Procedure or Label

Field 1 = Link to Extended Entry

Field 2 = Link to ON Block

Flag Bits 3-4

00 - Default New (ordinary variables)

01 - One-Level Global (GLOBAL)

10 - Declared New

11 - Multi-Level Global (Procedure Call)

Flag Bits 0-3

1111 = Data in Name Table

XI. APPENDIX III

SYMBOL SYNTAX

```
digit ::= 1|2|3|4|5|6|7|8|9|0

letter ::= A|B|C| --- |Y|Z|a|b|c| --- |y|z

character ::= _any character except ≠_

spacer ::= _a carriage-return, tab, or blank_

identifier ::= letter⌈[letter|digit|spacer]
               ...(letter|digit)]

decimal-number ::= digit...⌈.]|⌈digit...].digit...

exponent-part ::= 10⌊+|-][digit]digit

number ::= decimal-number|exponent-part
         |decimal-number exponent-part

string-number ::= ⌈+|-][(digit|,)...]number[EX|EM]

string ::= _sequence of zero or more of any characters
          except <>| and ≠_

ds ::= [List"|"string]

data-structure ::= <ds[data-structure ds]...>

ls ::= [List"|"identifier]

label-structure ::= <ls⌈label-structure ls]...>

as ::= ⌈List"|"exp]

assignment-structure ::= <as[assignment-structure as]...>

component ::= identifier"⌈"List,exp"]"

partial-reference ::= identifier"⌈"List,exp:exp"]"

procedure-call ::= identifier["("List,exp")"]

literal ::= number|"|"string"|"

recipient ::= identifier|component|procedure-call
```

```
arithmetic-op ::= +|-|*|/

string-op ::= JOIN|FORMAT|MASK

boolean-op ::= AND|OR

arithmetic-relation ::= GREATER[ THAN]|GTE|EQUAL[S]|NEQ
                      |LTE|LESS[ THAN]|<|≤|=|≠|≥|>

string-relation ::= BEFORE|SAME|AFTER

relational-op ::= arithmetic-relation|string-relation

dyadic-op ::= relational-op|arithmetic-op|string-op
            |boolean-op

monadic-op ::= +|-|ABS|NOT

primitive-exp ::= literal|identifier|component|procedure-call
                LIMIT|partial-reference|IN component

monadic-com ::= monadic-op(primitive-exp|dyadic-com
              |"("exp")")

dyadic-com ::= exp dyadic-op(primitive-exp|dyadic-com
             |"("exp")")

exp ::= primitive-exp|monadic-com|dyadic-com|"("exp")"

assignment-stm ::= List,(recipient|LIMIT)← exp|
                 assignment-structure

link-stm ::= List,recipient ← LINK exp

go-to-stm ::= GO[ TO ](identifier|component|procedure-call)

call-stm ::= [CALL]procedure-call

pause-stm ::= PAUSE

dummy-stm ::= [CONTINUE]

comment-stm ::= NOTE _any characters except :_

output-stm ::= OUTPUT([TO exp,]List,exp|STRING
             [TO exp,]List,exp|DATA[ TO exp,]
             List,identifier)
```

```
input-stm ::= INPUT([EX|EM][FROM exp,]List,recipient
              |STRING[FROM exp,]List,recipient
              |DATA[FROM exp])

initial-value-stm ::= identifier ("|"string"|"|data-structure)

switch-stm ::= SWITCH identifier label-structure

conditional-stm ::= IF exp THEN body [ELSE body] END

on-element-list ::= List,(identifier|INTERRUPT)

on-head ::= ON on-element-list

on-stm ::= on-head body END

on-control-stm ::= (DISABLE|ENABLE)on-element-list

procedure-head ::= PROCEDURE identifier
                   ["("List,identifier")"];

procedure-stm ::= procedure-head body END

return-stm ::= RETURN [exp]

block-stm ::= BLOCK body END

environment-stm ::= block-stm|on-stm|procedure-stm

scope-stm ::= GLOBAL List,identifier

compound-stm ::= conditional-stm|environment-stm

memory-op ::= AG|DE|DL|DS|FD|FF|FL|FR|FT|IG|SA|SD|SI|SO|ST

memory-stm ::= memory-op identifier

break-stm ::= SYSTEM|TRAP

restricted-stm ::= break-stm|memory-stm

stm ::= identifier:stm|assignment-stm|go-to-stm|call-stm
       |dummy-stm|comment-stm|output-stm|input-stm
       |initial-value-stm|on-control-stm|return-stm
       |pause-stm|switch-stm|scope-stm|restricted-stm
       |compound-stm|link-stm

body ::= [List;stm]

program ::= body╪
```