An Interpreter for a Foundation Program Language:
First Version

Jack B. Dennis
Paul J. Fox

February 1971
An Interpreter for a Foundation Program Language: First Version

The Computation Structures Group of Project MAC at M.I.T. is working on the design and specification of a foundation program language. This memo is a first experimental version of an interpreter for this language. The purpose of designing an interpreter is to provide a precise definition of the language, and to provide a guide to the implementation of the language. We expect that an implementation in the form of a computer system would closely follow the logic of the interpreter.

This interpreter is essentially an elaboration of our earlier work [1], which outlined an execution model for parallel computations and proposed a machine organization suitable for implementing these computations. In view of the close similarity between this interpreter and the earlier work, and our interest in having some documentation of this work available, we have not included an expository discussion. We hope that familiarity with the concepts in [1] will give the reader sufficient basis for understanding this memo.

The present language and interpreter differ from the earlier work in that our concept of control accompanying data, including the use of start and done signals for controlling the use of pointers, has not been included. This was done because we wish to illustrate a more complete foundation language and were not prepared to specify a data-control formulation of the interpreter that could encompass iteration and conditional instructions. The result is that the determinacy of a computation is more the responsibility of the translator, rather than being assured by the structure of procedures as in the earlier work. Implementation of general data-control concepts in an interpreter is a subject of current research. The motivation behind the modified form of attach and execute used in this version of the foundation language is found in a recent memo by P. J. Fox [2]. Another significant change is that each leaf node now has an associated pointer so that sharing of single elementary objects is readily accomplished.

This memo first gives an abstract syntax for the foundation language using definition schemata similar to those used by the IBM Vienna Group [3]. The abstract syntax includes the formats of the various instruction types of the language. The interpreter itself is defined by means of a formal syntax for a set of states, a set of primitive instructions which cause transformations of these states, and a representative collection of interpreter routines for foundation language instructions, expressed using the state-transforming primitive instructions.
References


A. Basic Classes - Elementary Objects and Selectors

\[ R \] real numbers (representations)
\[ Z \] integers
\[ W \] strings on some finite alphabet
\[ E \] elementary objects
\[ E = R \cup Z \cup W \]
\[ S \] selectors
\[ S = Z \cup W \]

B. Procedures and Information Structures

\[
\text{procedure} = \{ (s; \text{proced-part}) | s \in S \}
\]
\[
\text{proced-part} = \text{instruction} \cup \text{info-structure} \cup \text{procedure} \cup E
\]
\[
\text{info-structure} = \{ (s; \text{info-part}) | s \in S \}
\]
\[
\text{info-part} = \text{procedure} \cup \text{info-structure} \cup E
\]
\[
\text{argument-structure} = \text{info-structure}
\]

[The abstract syntax does not ensure that programs are well-formed.]
C. Instruction Types

1. Real arithmetic
   binary operations: radd, rsub, rmul, rdiv
   unary operation: rneg
   zeroary operation: rcom[stnt]
   binary predicates: rgre, requ
   unary predicate: rpos, rzer

2. Integer arithmetic
   binary operations: iadd, isub, imul, iquo, irem
   unary operation: ineg
   zeroary operation: icon[stnt]
   binary predicates: igre, iequ
   unary predicates: ipos, izer

3. String manipulation
   binary operations: concat[ation]
   unary operations: first, last, head, tail [head is all characters but the last, tail is all but the first]
   zeroary operation: string
   binary predicate: same
   unary predicate: empty

4. Assignment and type conversion
   real and integer: rtoi, itor
   integer and strings of length l: ctoi, itoc
   length of string: lgth
   assignment: rename
5. Structure access and modification

fixed selector: .fsect, fapp[end], fdel[etc]
variable selector: vsel, vapp, vdel
elementary objects: fetch, assign
linking: attach, bind
predicates on pointer variables: isreal, isint, isstr, isval,
issstruct[ure]
predicate on pointer and selector: fdef, vdef[ined]

6. Control

sequencing: succ, join, case
procedure application: create, exec, end

D. The Object Class Instruction

An instruction has from one to five components accessed by selectors in \([1, 2, 3, 4, 5]\). The types of the component elementary objects are specified by the following formats:

1.

\[
\begin{array}{c}
1 \\
\left\{ rneg, ineg, head, tail, \right. \\
\left\{ first, last, rtoi, itor, ctoi, itoc, lgth, \right. \\
\left\{ fetch, \right. \\
\left\{ create, case \right. \\
\end{array}
\]

\( O_1 = \)
2.

\[
0_2 = \{ \text{radd, rsab, rmul, rdiv, rpos, rzer, iadd, isub, imul, iquo, irem, ipos, iter, } \text{concat, empty, rename, fael, fapp, fdel, vael, vapp, vdel, assign, isval, isreal, isint, isstr, isstruct, succ.} \}
\]

3.

\[
0_3 = \{ \text{rge, requ, igre, iequ, same, } \text{attach, bind, fdef, vdef, exec.} \}
\]
The instruction that is enabled by completion of two predecessor instructions is `join`.
States of the Interpreter

A state is a finite collection of objects of the class state-item specified by the following syntax:

\[
\text{state-item} = \text{elem-obj-item} \cup \text{gen-struct-item} \\
\quad \cup \text{local-struct-item} \\
\quad \cup \text{process-item}
\]

\[
\text{elem-obj-item} = \langle \langle 1; \text{'real'} \rangle, \langle 2; P \rangle, \langle 3; R \rangle \rangle \\
\quad \cup \langle \langle 1; \text{'integer'} \rangle, \langle 2; P \rangle, \langle 3; Z \rangle \rangle \\
\quad \cup \langle \langle 1; \text{'string'} \rangle, \langle 2; P \rangle, \langle 3; W \rangle \rangle
\]

[The set \( P \) is a class of pointer codes which are the values taken on by pointer variables of a procedure. The codes may be assumed to be integers, that is, \( P \subseteq \mathbb{Z} \).]

[An item is said to be pointed to by the pointer code which is its second component.]

\[
\text{gen-struct-item} = \langle \langle 1; \text{'object'} \rangle, \langle 2; P \rangle, \langle 3; S \rangle, \langle 4; P \rangle \rangle
\]

[where the 4-component points to gen-struct-item's or elem-obj-item]
local-struct-item = (⟨1; 'rptr'⟩, ⟨2; F⟩, ⟨3; S⟩, ⟨4; F⟩) [read pointer]

[where the 4-component points to gen-struct-item's or elem-obj-item's.]

U (⟨1; 'wptr'⟩, ⟨2; F⟩, ⟨3; S⟩, ⟨4; F⟩) [write pointer]

[where the 4-component points to gen-struct-item's or elem-obj-item's.]

U (⟨1; 'pptr'⟩, ⟨2; F⟩, ⟨3; S⟩, ⟨4; F⟩) [procedure pointer]

[where the 4-component points to gen-struct-item's.]

U (⟨1; 'eop'⟩, ⟨2; F⟩, ⟨3; S⟩, ⟨4; F⟩) [elementary object pointer]

[where the 4-component points to a elem-obj-item.]

U (⟨1; 'istr'⟩, ⟨2; F⟩, ⟨3; S⟩, ⟨4; F⟩) [local structure]

[where the 4-component points to local-struct-item's.]

U (⟨1; 'loc'⟩, ⟨2; F⟩, ⟨3; F⟩)

[where the 3-component points to local-struct-item's.]

U (⟨1; 'prec'⟩, ⟨2; F⟩, ⟨3; F⟩)

[where the 3-component points to gen-struct-item's.]

U (⟨1; 'ret'⟩, ⟨2; F⟩, ⟨3; S⟩)

process-item = (⟨1; 'en1'⟩, ⟨2; F⟩, ⟨3; S⟩, ⟨4; F⟩)

U (⟨1; 'ena'⟩, ⟨2; F⟩, ⟨3; S⟩, ⟨4; F⟩)

U (⟨1; 'null'⟩, ⟨2; F⟩, ⟨3; S⟩, ⟨4; F⟩)

[where the 2-components points to local-struct-item's, and the 4-component points to gen-struct-item's.]
Representation of the Interpreter

The interpreter advances through a sequence of states as a result of executing instructions. For any state there will be some number of enabled process items indicating instructions ready for execution. A state transition consists in the execution of any one of these instructions. It is convenient to specify the state transitions for each instruction type by a routine expressed in a simple language. This language makes use of variables that range over pointer codes or elementary objects, simple assignment statements involving primitive operations on elementary data types, and simple conditional statements using primitive predicates. The statement *quit* is executed in situations where the result of executing an instruction is undefined. The following primitive statements permit manipulation of the state table of the interpreter.

match (t, p, n, x) else S

The values of variables (or literals) t, p, n and x are compared with corresponding components of each item in the state table. If there is no match, statement S is obeyed. If there is exactly one match, this item of the state table becomes selected. If there is a match with each of several items, an arbitrary one of these becomes selected. If one of the variable names t, p, n, x is an asterisk, it is assumed to be matched regardless of the value of the corresponding component of a state-item.

read → (t, p, n, x)

The components of the selected item of the state table become the values of the designated variables. If a dash appears in place of a variable, the corresponding component of the selected item is not used.
write (t, p, n, x)

The components of the selected item are assigned the values of variables t, p, n, x. A dash in place of a variable name indicates that the corresponding component of the selected item retains its previous value.

item

Select a new item added to the state table.

purge

Delete the selected item from the state table.

code \rightarrow p

Generate a pointer code not previously used.

\textbf{find} \quad t1 \rightarrow t2 \ \textbf{else} \ S

An attempt is made to find an item in the state table of type t1. If unsuccessful statement S is obeyed. If successful the item (or an arbitrary one if there is more than one) is selected and changed to type t2. In other words the indivisible execution of:

\textbf{match} (t1, *, *, *) \ \textbf{else} \ S

\textbf{write} (t2, -, -, -)

\textbf{ifrest} (t, p, n, x) \ \textbf{goto} \ k

The indivisible execution of:

\textbf{if} \ \textbf{match} (t, p, n, x) \ \textbf{goto} \ k

item

\textbf{write} (t, p, n, x)
The Interpreter

A few macro instructions are handy to shorten the routines of the interpreter.

1. To access a pointer by a pointer and a selector, and verify its type:

   \[ \text{obtain } (p, n, r) \rightarrow pl \]

   means:

   \[
   \text{match } (t, p, n, *) \text{ else quit}
   \]

   \[
   \text{read } \rightarrow (-, -, -, pl)
   \]

   \[
   \text{end}
   \]

2. To get a selector (which may be either an integer or a string)
   from a local data structure:

   \[ \text{get-selector } i \rightarrow sel \]

   means:

   \[
   \text{match } ('obj', p-ins, 'i', *) \text{ else quit}
   \]

   \[
   \text{read } \rightarrow (-, -, -, p-sel)
   \]

   \[
   \text{match } (*) \text{ else quit}
   \]

   \[
   \text{read } \rightarrow (\text{type, }, -, \text{ sel})
   \]

   \[
   \text{if type } \neq \text{ 'int' and type } \neq \text{ 'str' then quit}
   \]

   \[
   \text{end}
   \]

3. A routine to fetch an elementary object of type t2 from a structure
   of type t1, by means of pointer p and selector s.

   \[ \text{get } (p, s, t1, t2) \rightarrow x \]

   means:

   \[
   \text{match } (t1, p, s, *) \text{ else quit}
   \]

   \[
   \text{read } \rightarrow (-, -, -, p-sel)
   \]

   \[
   \text{match } (t2, p-sel, *) \text{ else quit}
   \]

   \[
   \text{read } \rightarrow (-, -, x)
   \]

   \[
   \text{end}
   \]
4. A routine to update or append an elementary object of type t2 to a structure of type t1, by means of pointer p and selector s.

\[ \text{put} (p, s, t1, t2, y) \]

means:

\[ \text{match}(*, p, s, *) \text{ else goto 1} \]

\[ \text{read} \to (\text{type, -}, -, p-y) \]

\[ \text{if type} \neq t1 \text{ then quit} \]

\[ \text{goto 2} \]

1: \[ \text{code} \to p-y \]

\[ \text{item} \]

\[ \text{write} (t1, p, s, p-y) \]

2: \[ \text{match} (*, p-y, *) \text{ else goto 3} \]

\[ \text{goto 4} \]

3: \[ \text{item} \]

4: \[ \text{write} (t2, p-y, y) \]

\[ \text{end} \]
comment: Pick an enabled process item and retrieve the associated procedure item. [See Diagram]

interp:

\[ \text{find} \ 'ena' \rightarrow 'null' \ \text{else goto interp} \]
\[ \text{read} \rightarrow (-, \ p\text{-loc}, \ s\text{-ins}, \ p\text{-pr}) \]
\[ \text{purge} \]
\[ \text{obtain} \ (p\text{-pr}, \ s\text{-ins}, \ 'obj') \rightarrow p\text{-ins} \]
\[ \text{get} \ (p\text{-ins}, \ 'l', \ 'obj', \ 'str') \rightarrow \text{opcode} \]
\[ \text{case} \ (\text{opcode}) \]

comment: The case statement gives control to the routine labelled by the value of opcode.

comment: Routine for enabling successor instructions; \( \text{conj}(\text{opcode}) = \text{true} \) if the successor instruction requires completion of two predecessor instructions (i.e. join).

enable:

\[ \text{obtain} \ (p\text{-pr}, \ s\text{-suc}, \ 'obj') \rightarrow p\text{-ins} \]
\[ \text{get} \ (p\text{-ins}, \ 'l', \ 'obj', \ 'str') \rightarrow \text{opcode} \]
\[ \text{if conj (opcode)} \ \text{goto 3} \]
\[ \text{match} \ (*, \ p\text{-loc}, \ s\text{-suc}, \ *) \ \text{else goto 1} \]
\[ \text{goto 2} \]

1: \ item

2: \ write \ ('ena', \ p\text{-loc}, \ s\text{-suc}, \ p\text{-pr}) \]
\[ \text{goto interp} \]

3: \ match \ (*, \ p\text{-loc}, \ s\text{-suc}, \ *) \ \text{else goto 4} \]
\[ \text{read} \rightarrow \text{type, -, -, -} \]
\[ \text{if type} \neq \ 'enl' \ \text{then purge} \]

4: \ iftest \ ('enl', \ p\text{-loc}, \ s\text{-suc}, \ p\text{-pr}) \ \text{goto 2} \]
\[ \text{goto interp} \]
comment: Binary operations on elementary objects [See Diagram]

radd: get-selector 2 \rightarrow r1
get-selector 3 \rightarrow r2
get-selector 4 \rightarrow s-suc
get (p-loc, r1, 'eop', 'real') \rightarrow x1
get (p-loc, r2, 'eop', 'real') \rightarrow x2
RADD (x1, x2) \rightarrow y
put (p-loc, s-ins, 'eop', 'real', y)
goto enable

comment: Instructions treated similarly:
rsub, rmul, rdiv, iadd, isub, imul, iquo, irem, concat,

comment: Unary operations on elementary objects

rneq: get-selector 2 \rightarrow r1
get-selector 3 \rightarrow s-suc
get (p-loc, r1, 'eop', 'real') \rightarrow x
rneq(x) \rightarrow y
put (p-loc, s-ins, 'eop', 'real', y)
goto enable

comment: Instructions treated similarly:
ineg, head, tail, first, last, rtoi, itor, ctoi, itoc, lgth.
comment: Conditionals with binary predicates:

rgre: get-selector 2 \to r1
      get-selector 3 \to r2
      get (p-loc, r1, 'eop', 'real') \to x1
      get (p-loc, r2, 'eop', 'real') \to x2
      if RGRE (x1, x2) then goto 1
      get-selector 5 \to s-suc
      goto enable

1: get-selector 4 \to s-suc
      goto enable

comment: Instructions treated similarly:
requ, igre, iequ, same

comment: Instructions for manipulating structures:

tsel: get-selector 2 \to s-ptr
      get-selector 3 \to sel
      get-selector 4 \to s-suc
      match (*, p-loc, s-ptr, *) else quit
      read \to (type, -, -, ptr)
      if type \neq 'wptr' and type \neq 'rptr' then quit
      match ('obj', ptr, sel, *) else quit
      read \to (-, -, -, pl)
      match (*, p-loc, s-ins, *) else goto 1
      goto 2

1: item
2: write ('rptr', p-loc, s-ins, pl)
      goto enable
[See Diagram]

vapp[end]:

get-selector 2 → s-ptr
get-selector 3 → s-sel
get-selector 4 → s-suc

match ('eop', p-loc, s-sel, *) else quit
read → (-, -, -, p-sel)

match (*, p-sel, *) else quit
read → (type, -, sel)

if type ≠ 'int' and type ≠ 'str' then quit

match ('wptr', p-loc, s-ptr, *) else quit
read → (-, -, -, ptr)

match (*, ptr, sel, *) else goto 1
read → (type1, -, -, pl)

if type1 ≠ 'obj' then quit
goto 3

1: match (*, ptr, *) else goto 2

purge

2: code → pl

item

write ('obj', ptr, sel, pl)

3: match (*, p-loc, s-ins, *) else goto 4
goto 5

4: item

5: write ('wptr', p-loc, s-ins, pl)
goto enable
fdel:  
\texttt{get-selector 2 \rightarrow s.ptr} \\
\texttt{get-selector 3 \rightarrow sel} \\
\texttt{get-selector 4 \rightarrow s.suc} \\
\texttt{match ('wptr', p.loc, s.ptr, \_)} \texttt{else quit} \\
\texttt{read \rightarrow (-, -, -, ptr)} \\
\texttt{match ('obj', ptr, sel, \_)} \texttt{else quit} \\
\texttt{purge} \\
\texttt{goto enable} \\
\texttt{comment: Instructions vsel, fapp, and vdel are performed by variations on the above routines.}

attach:  
\texttt{get-selector 2 \rightarrow s.pl} \\
\texttt{get-selector 3 \rightarrow sel} \\
\texttt{get-selector 4 \rightarrow s.p2} \\
\texttt{get-selector 5 \rightarrow s.suc} \\
\texttt{match (*, p.loc, s.pl, \_)} \texttt{else quit} \\
\texttt{read \rightarrow (tl, -, -, p1)} \\
\texttt{if tl \neq 'rptr' and tl \neq 'wptr' then quit} \\
\texttt{match ('wptr', p.loc, s.p2, \_)} \texttt{else quit} \\
\texttt{read \rightarrow (-, -, -, p2)} \\
\texttt{match (*, p2, sel, \_)} \texttt{else goto 1} \\
\texttt{goto 2} \\
1: \texttt{item} \\
2: \texttt{write ('obj', p2, sel, pl)} \\
\texttt{goto enable} \\
\texttt{comment: A similar routine performs bind instructions.}
assign:  
\[ \text{get-selector} \ 2 \to s\text{-ptr} \]
\[ \text{get-selector} \ 3 \to s\text{-val} \]
\[ \text{get-selector} \ 4 \to s\text{-suc} \]
\[ \text{match} \ ('\text{eop}', \ p\text{-loc}, \ s\text{-val}, \ *) \ \text{else quit} \]
\[ \text{read} \to (-, -, -, \text{ptr}) \]
\[ \text{match} \ (*, \text{ptr}, \*) \ \text{else quit} \]
\[ \text{read} \to (\text{type}, -, \text{value}) \]
\[ \text{if type} \neq \ '\text{str}' \ \text{and type} \neq \ '\text{int}' \ \text{and type} \neq \ '\text{real}' \ \text{then quit} \]
\[ \text{match} \ ('\text{wptr}', \ p\text{-loc}, \ s\text{-ptr}, \*) \ \text{else quit} \]
\[ \text{read} \to (-, -, -, \text{ptr}) \]
\[ \text{match} \ (*, \text{ptr}, \*) \ \text{else goto} \ 1 \]
\[ \text{goto} \ 2 \]

1:  \text{item} 

2:  \text{write} \ (\text{type}, \text{ptr}, \text{value}) 
    \[ \text{goto} \ \text{enable} \]

rename:  
\[ \text{get-selector} \ 2 \to \text{sel1} \]
\[ \text{get-selector} \ 3 \to \text{sel2} \]
\[ \text{get-selector} \ 4 \to \text{s-suc} \]
\[ \text{match} \ (*, \ p\text{-loc}, \ \text{sel1}, \*) \ \text{else quit} \]
\[ \text{read} \to (\text{type}, -, -, \text{ptr}) \]
\[ \text{match} \ (*, \ p\text{-loc}, \ \text{sel2}, \*) \ \text{else goto} \ 1 \]
\[ \text{goto} \ 2 \]

1:  \text{item} 

2:  \text{write} \ (\text{type}, \ p\text{-loc}, \ \text{sel2}, \text{ptr}) 
    \[ \text{match} \ (\text{type}, \ p\text{-loc}, \ \text{sel1}, \text{ptr}) \ \text{else quit} \]
    \text{purge} 
    \[ \text{goto} \ \text{enable} \]
comment: Sequencing control instructions.

join:  get-selector 2 \rightarrow s-suc

            goto enable

succ:  get-selector 2 \rightarrow s-suc
            get-selector 3 \rightarrow s-sucl
            obtain (p-pr, s-sucl, 'obj') \rightarrow p-ins
            get (p-ins,'i', 'obj', 'str') \rightarrow opcode
            if conj (opcode) goto 3
            match (*, p-loc, s-sucl, *) else goto 1
            goto 2

1:  item

2:  write ('ena', p-loc, s-sucl, p-pr)
            goto enable

3:  match (*, p-loc, s-sucl, *) else goto 4
            read \rightarrow (type, -, -, -)
            if type \neq 'enl' then purge

4:  iftest ('enl', p-loc, s-sucl, p-pr) goto 2
            goto enable
control: Control instructions for procedure activation and return.

create: get-selector 2 -> sel
        get-selector 3 -> s-suc
        match (*, p-loc, sel, *) else goto 1
        goto 2

1: item

2: code -> pl
        write ('wptr', p-loc, sel, pl)
        goto enable

exec: get-selector 2 -> s-prl [See Diagram]
        get-selector 3 -> s-warg
        get-selector 4 -> s-rarg
        obtain (p-loc, s-prl, 'p.ptr') -> p-prl
        obtain (p-loc, s-warg, 'wptr') -> p-warg
        obtain (p-loc, s-rarg, 'wptr') -> p-rarg
        match (*, p-loc, s-ins, *) else goto 1
        goto 2

1: item

2: code -> p-local
        write ('lstr', p-loc, s-ins, p-local)
        item
        write ('loc', p-local, p-loc)
        item
        write ('prc', p-local, p-pr)
        item
        write ('ret', p-local, s-ins)
        item
        write ('wptr', p-local, Ow, p-warg)
        item
        write ('rptr', p-local, Or, p-rarg)
        item
        write ('ena', p-local, 'l', p-prl)
        goto interp
end:  \textbf{match} ('loc', p-loc, *) \textbf{else} \textbf{quit}

\textbf{read} \rightarrow (\_, \_, p-loc0)

\textbf{match} ('proc', p-loc, *) \textbf{else} \textbf{quit}

\textbf{read} \rightarrow (\_, \_, p-pr)

\textbf{match} ('ret', p-loc, *) \textbf{else} \textbf{quit}

\textbf{read} \rightarrow (\_, \_, s-ins)

\textbf{match} ('istr', p-loc0, s-ins, p-loc) \textbf{else} \textbf{quit}

\textbf{purge}

p-loc0 \rightarrow p-loc

\textit{get-selector} 5 \rightarrow s-suc

\textbf{goto} enable
vappend in Ver 1

Procedure:

- p-prd
- p-ins
- p-suc

Local Data:

- s-sel
- s-ptr
- s-ins

A Structure:

- sel
- obj

p1

pointer

ptr

- p1

+ sel

+ obj
### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Initial State-item's</th>
<th>Initial State-item's</th>
</tr>
</thead>
<tbody>
<tr>
<td>process-item</td>
<td>'ena' p-loc s-ins p-pr</td>
<td>'eop' p-loc r1 p-xl</td>
</tr>
<tr>
<td>gen-struct-item</td>
<td>'obj' p-pr s-ins p-ins</td>
<td>'real' p-xl x1</td>
</tr>
<tr>
<td>gen-struct-item</td>
<td>'obj' p-ins 'l' p-el</td>
<td>'eop' p-loc r2 p-x2</td>
</tr>
<tr>
<td>elem-obj-item</td>
<td>'str' p-e1 'radd'</td>
<td>'real' p-x2 x2</td>
</tr>
<tr>
<td></td>
<td>'obj' p-ins '2' p-e2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'str/int' p-e2 r1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'obj' p-ins '3' p-e3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'str/int' p-e3 r2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'obj' p-ins '4' p-e4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'str/int' p-e4 s-suc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'obj' p-pr s-suc p-suc</td>
<td></td>
</tr>
</tbody>
</table>

### Local Data

<table>
<thead>
<tr>
<th>Local Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-loc &gt; p-xl</td>
</tr>
<tr>
<td>r2 eop</td>
</tr>
<tr>
<td>s-ins eop</td>
</tr>
</tbody>
</table>

### Procedure


<table>
<thead>
<tr>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-loc &gt;</td>
</tr>
<tr>
<td>p-suc &gt;</td>
</tr>
</tbody>
</table>

### Output State-item's

<table>
<thead>
<tr>
<th>Output State-item's</th>
</tr>
</thead>
<tbody>
<tr>
<td>'eop' p-loc s-ins p-y</td>
</tr>
<tr>
<td>'real' p-y y</td>
</tr>
</tbody>
</table>

---

P J F 3/26/71