DRAFT

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 58

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 <u>attach</u> and <u>execute</u> 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

1. J. B. Dennis, Programming Generality, Parallelism and Computer Architecture. Computation Structures Group Memo 32, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1968. [Also Information Processing 68, North-Holland, Amsterdam 1969, pp 484-492.]

2. P. J. Fox, Sharing in J. B. Dennis' Programming Generality, Parallelism and Computer Architecture. Computation Structures Group Memo 55, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1971.

3. P. Lucas and K. Walk, On the formal description of PL/I. Annual Review in Automatic Programming, Vol. 6, Part 3, Pergamon Press, 1969.

Abstract Syntax (Edition 1)

A. Basic Classes - Elementary Objects and Selectors

$\underline{R}$	real numbers (representations)
$\underline{Z}$	integers
$\underline{W}$	strings on some finite alphabet
$\underline{E}$	elementary objects
	$\underline{E} = \underline{R} \cup \underline{Z} \cup \underline{W}$
$\underline{S}$	selectors
	$\underline{S} = \underline{Z} \cup \underline{W}$

B. Procedures and Information Structures

$\underline{\text{procedure}} = \{\langle s;\ \underline{\text{proced-part}}\rangle \mid s \in \underline{S}\}$

$\underline{\text{proced-part}} = \underline{\text{instruction}} \cup \underline{\text{info-struct[ure]}} \cup \underline{\text{procedure}} \cup \underline{E}$

$\underline{\text{info-struct}} = \{\langle s;\ \underline{\text{info-part}}\rangle \mid s \in \underline{S}\}$

$\underline{\text{info-part}} = \underline{\text{procedure}} \cup \underline{\text{info-struct}} \cup \underline{E}$

$\underline{\text{argument-struct}} = \underline{\text{info-struct}}$

[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: rcon[stant]

binary predicates: rgre, requ

unary predicate: rpos, rzer

2. Integer arithmetic

binary operations: iadd, isub, imul, iquo, irem

urary operation: ineg

zeroary operation: icon[stant]

binary predicates: igre, iequ

unary predicates: ipos, izer

3. String manipulation

binary operations: concat[enation]

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 1: ctoi, itoc

length of string: lgth

assignment: rename

5. Structure access and modification

fixed selector: fsel[ect], fapp[end], fdel[ete]

variable selector: vsel, vapp, vdel

elementary objects: fetch, assign

linking: attach, bind

predicates on pointer variables: isreal, isint, isstr, isval,
isstruct[ure]

predicate on pointer and selector: fdef, vdef[ined]

6. Control

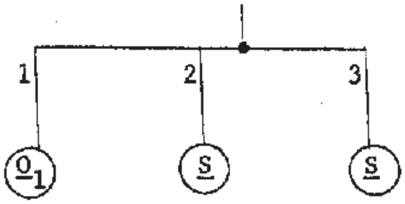
sequencing: succ, join, case

procedure application: create, exec, end

D. The Object Class Instruction

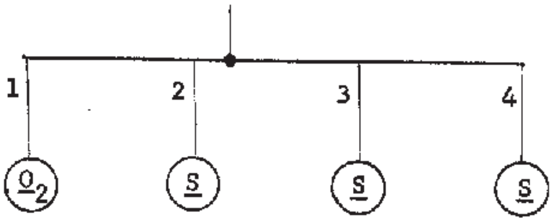
An instruction has from one to five components accessed by selectors in {1, 2, 3, 4, 5}. The types of the component elementary objects are specified by the following formats:

1.



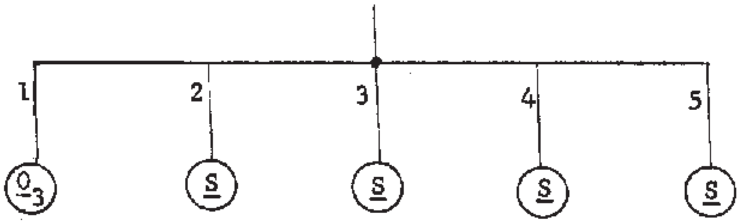
$$\underline{O}_1 = \left\{ \begin{array}{l} \text{rneg, ineg, head, tail,} \\ \text{first, last, rtoi, itor, ctoi, itoc, lgth,} \\ \text{fetch,} \\ \text{create, case} \end{array} \right\}$$

2.



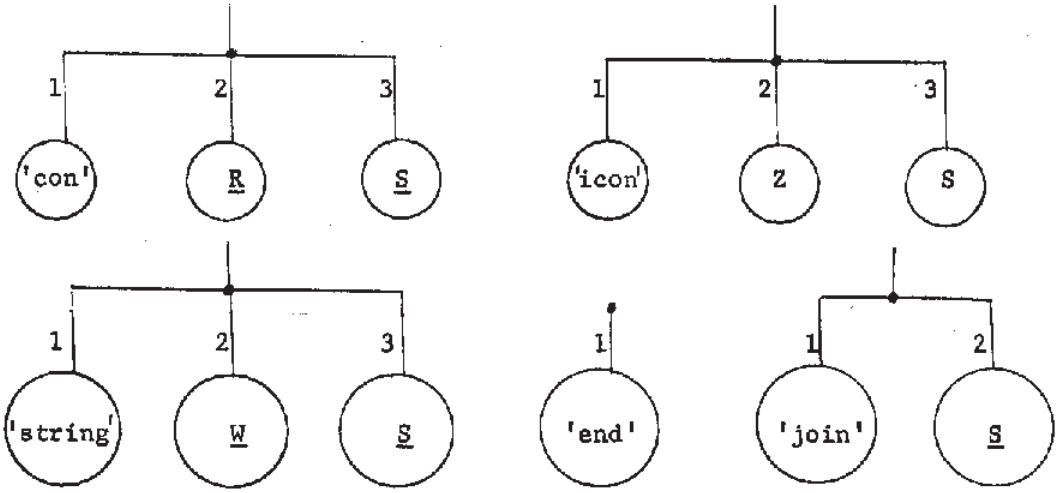
$$\underline{Q}_2 = \left\{\begin{array}{l} \text{radd, rsub, rmul, rdiv, rpos, rzer} \\ \text{iadd, isub, imul, iquo, irem, ipos, iter} \\ \text{concat, empty, rename} \\ \text{fsel, fapp, fdel, vsel, vapp, vdel, assign} \\ \text{isval, isreal, isint, isstr, isstruct} \\ \text{succ.} \end{array}\right\}$$

3.



$$\underline{Q}_3 = \left\{\begin{array}{l} \text{rgre, requ, igre, iequ, same,} \\ \text{attach, bind, fdef, vdef, exec.} \end{array}\right\}$$

4.



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:

state-item = *elem-obj-item*
$\cup$ *gen-struct-item*
$\cup$ *local-struct-item*
$\cup$ *process-item*

elem-obj-item = $(\langle 1;\ \text{'real'}\rangle,\ \langle 2;\ \underline{P}\rangle,\ \langle 3;\ \underline{R}\rangle)$
$\cup\ (\langle 1;\ \text{'int[eger]'}\rangle,\ \langle 2;\ \underline{P}\rangle,\ \langle 3;\ \underline{Z}\rangle)$
$\cup\ (\langle 1;\ \text{'str[ing]'}\rangle,\ \langle 2;\ \underline{P}\rangle,\ \langle 3;\ \underline{W}\rangle)$

[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, $\underline{P} \subseteq \underline{Z}$.]

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

gen-struct-item = $(\langle 1;\ \text{'obj[ect]'}\rangle,\ \langle 2;\ \underline{P}\rangle,\ \langle 3;\ \underline{S}\rangle,\ \langle 4;\ \underline{P}\rangle)$

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

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

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

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

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

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

∪ (⟨1; 'eop'⟩, ⟨2; P⟩, ⟨3; S⟩, ⟨4; P⟩ [elementary object pointer

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

∪ (⟨1; 'lstr'⟩, ⟨2; P⟩, ⟨3; S⟩, ⟨4; P⟩) [local structure]

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

∪ (⟨1; 'loc'⟩, ⟨2; P⟩, ⟨3; P⟩)

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

∪ (⟨1; 'prc'⟩, ⟨2; P⟩, ⟨3; P⟩)

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

∪ (⟨1; 'ret'⟩, ⟨2; P⟩, ⟨3; S⟩)

process-item = (⟨1; 'enl'⟩, ⟨2; P⟩, ⟨3; S⟩, ⟨4; P⟩)
∪ (⟨1; 'ena'⟩, ⟨2; P⟩, ⟨3; S⟩, ⟨4; P⟩)
∪ (⟨1; 'null'⟩, ⟨2; P⟩, ⟨3; S⟩, ⟨4; P⟩)
[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 → p

Generate a pointer code not previously used.

find t1 → t2 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:

```
match (t1, *, *, *) else S
write (t2, -, -, -)
```

iftest (t, p, n, x) goto k

The indivisible execution of:

```
if match (t, p, n, x) goto k
item
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:

```
obtain (p, n, t) → p1
```

means:

```
        match (t, p, n, *) else quit
        read → (-, -, -, p1)
        end
```

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

```
get-selector i → sel
```

means:

```
        match ('obj', p-ins,'i,' *) else quit
        read → (-, -, -, p-sel)
        match (*, p-sel, *) else quit
        read → (type, -, sel)
        if type ≠ 'int' and type ≠ 'str' then quit
        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.

```
get (p, s, t1, t2) → x
```

means:

```
        match (t1, p, s, *) else quit
        read → (-, -, -, p-el)
        match (t2, p-el, *) else quit
        read → (-, -, x)
        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.

```
put (p, s, t1, t2, y)
means:
    match(*, p, s, *) else goto 1
    read → (type, -, -, p-y)
    if type ≠ t1 then quit
    goto 2
1:  code → p-y
    item
    write (t1, p, s, p-y)
2:  match (*, p-y, *) else goto 3
    goto 4
3:  item
4:  write (t2, p-y, y)
    end
```

comment: Pick an enabled process item and retrieve the associated procedure item. [See Diagram]

```
interp:  find 'ena' → 'null' else goto interp
         read → (-, p-loc, s-ins, p-pr)
         purge
         obtain (p-pr, s-ins, 'obj') → p-ins
         get (p-ins, '1', 'obj', 'str') → opcode
         case (opcode)
```

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

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

```
enable:  obtain (p-pr, s-suc, 'obj') → p-ins
         get (p-ins,'1,' 'obj', 'str') → opcode
         if conj (opcode) goto 3
         match (*, p-loc, s-suc, *) else goto 1
         goto 2
     1:  item
     2:  write ('ena', p-loc, s-suc, p-pr)
         goto interp
     3:  match (*, p-loc, s-suc, *) else goto 4
         read → (type, -, -, -)
         if type ≠ 'enl' then purge
     4:  iftest ('enl', p-loc, s-suc, p-pr) goto 2
         goto interp
```

```
comment:   Binary operations on elementary objects [See Diagram]

     radd:   get-selector 2 → r1
             get-selector 3 → r2
             get-selector 4 → s-suc
             get (p-loc, r1,  'eop', 'real') → x1
             get (p-loc, r2,  'eop', 'real') → x2
             RADD (x1, x2) → 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

     rneg:   get-selector 2 → r1
             get-selector 3 → s-suc
             get (p-loc, r1, 'eop',  'real') → x
             RNEG(x) → 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 → r1
             get-selector 3 → r2
             get (p-loc, r1, 'eop', 'real') → x1
             get (p-loc, r2, 'eop', 'real') → x2
             if RGRE (x1, x2) then goto 1
             get-selector 5 → s-suc
             goto enable
         1:  get-selector 4 → s-suc
             goto enable

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

comment:  Instructions for manipulating structures:

     fsel:   get-selector 2 → s-ptr
             get-selector 3 → sel
             get-selector 4 → s-suc
             match (*, p-loc, s-ptr, *) else quit
             read → (type, -, -, ptr)
             if type ≠ 'wptr' and type ≠ rptr then quit
             match ('obj',  ptr, sel, *) else quit
             read → (-, -, -, p1)
             match (*, p-loc, s-ins, *) else goto 1
             goto 2
         1:  item
         2:  write ('rptr', p-loc, s-ins, p1)
             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, -, -, p1)
            if type1 ≠ 'obj' then quit
            goto 3
        1:  match (*, ptr, *) else goto 2
            purge
        2:  code → p1
            item
            write ('obj', ptr, sel, p1)
        3:  match (*, p-loc, s-ins, *) else goto 4
            goto 5
        4:  item
        5:  write ('wptr', p-loc, s-ins, p1)
            goto enable
```

```
    fdel:  get-selector 2 → s-ptr
           get-selector 3 → sel
           get-selector 4 → s-suc
           match  ('wptr', p-loc, s-ptr, *) else quit
           read →  (-, -, -, ptr)
           match ('obj', ptr, sel, *) else quit
           purge
           goto enable
```

comment: Instructions vsel, fapp, and vdel are performed by variations on the above routines.

```
    attach:  get-selector 2 → s-p1
             get-selector 3 → sel
             get-selector 4 → s-p2
             get-selector 5 → s-suc
             match (*, p-loc, s-p1, *) else quit
             read → (t1, -, -, p1)
             if t1 ≠ 'rptr' and t1 ≠ 'wptr' then quit
             match ('wptr', p-loc, s-p2, *) else quit
             read → (-, -, -, p2)
             match (*, p2, sel, *) else goto 1
             goto 2
          1: item
          2: write ('obj',  p2, sel, p1)
             goto enable
```

comment: A similar routine performs bind instructions.

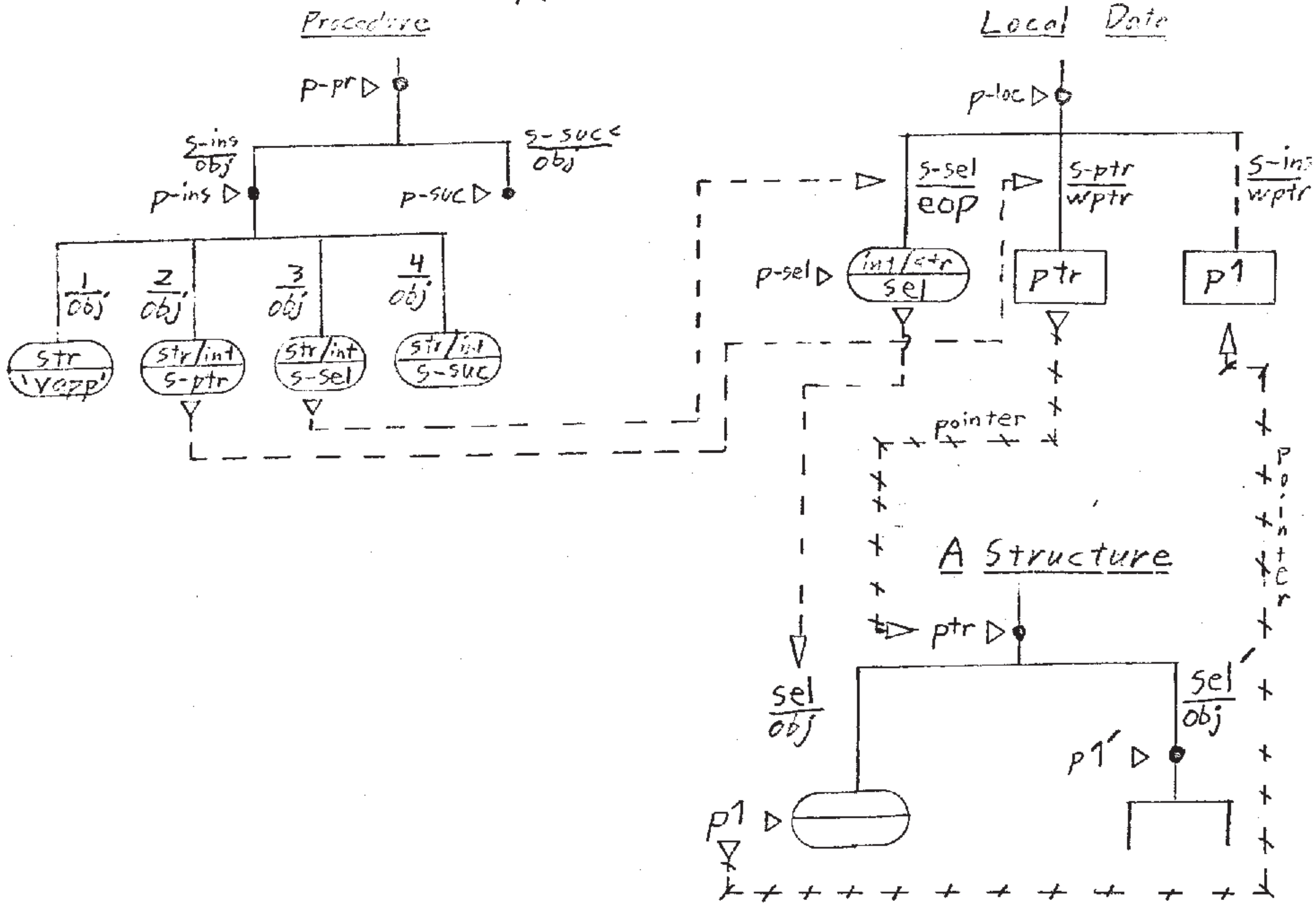
```
assign:   get-selector 2 → s-ptr
          get-selector 3 → s-val
          get-selector 4 → s-suc
          match ('eop', p-loc, s-val, *) else quit
          read → (-, -, -, ptr)
          match (*, ptr, *) else quit
          read → (type, -, value)
          if type ≠ 'str' and type ≠ 'int' and type ≠ 'real' then quit
          match ('wptr', p-loc, s-ptr, *) else quit
          read → (-, -, -, ptr)
          match (*, ptr, *) else goto 1
          goto 2
    1:    item
    2:    write (type, ptr, value)
          goto enable
rename:   get-selector 2 → sel1
          get-selector 3 → sel2
          get-selector 4 → s-suc
          match (*, p-loc, sel1, *) else quit
          read → (type, -, -, ptr)
          match (*, p-loc, sel2, *) else goto 1
          goto 2
    1:    item
    2:    write (type, p-loc, sel2, ptr)
          match (type, p-loc, sel1, ptr) else quit
          purge
          goto enable
```

```
comment: Sequencing control instructions.

   join: get-selector 2 → s-suc
         goto enable

   succ: get-selector 2 → s-suc
         get-selector 3 → s-sucl
         obtain (p-pr, s-sucl,  'obj' ) → p-ins
         get (p-ins,'1', 'obj', 'str') → 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 → (type, -, -, -)
         if type ≠ 'enl' then purge
     4:  iftest ('enl', p-loc, s-sucl, p-pr) goto 2
         goto enable
```

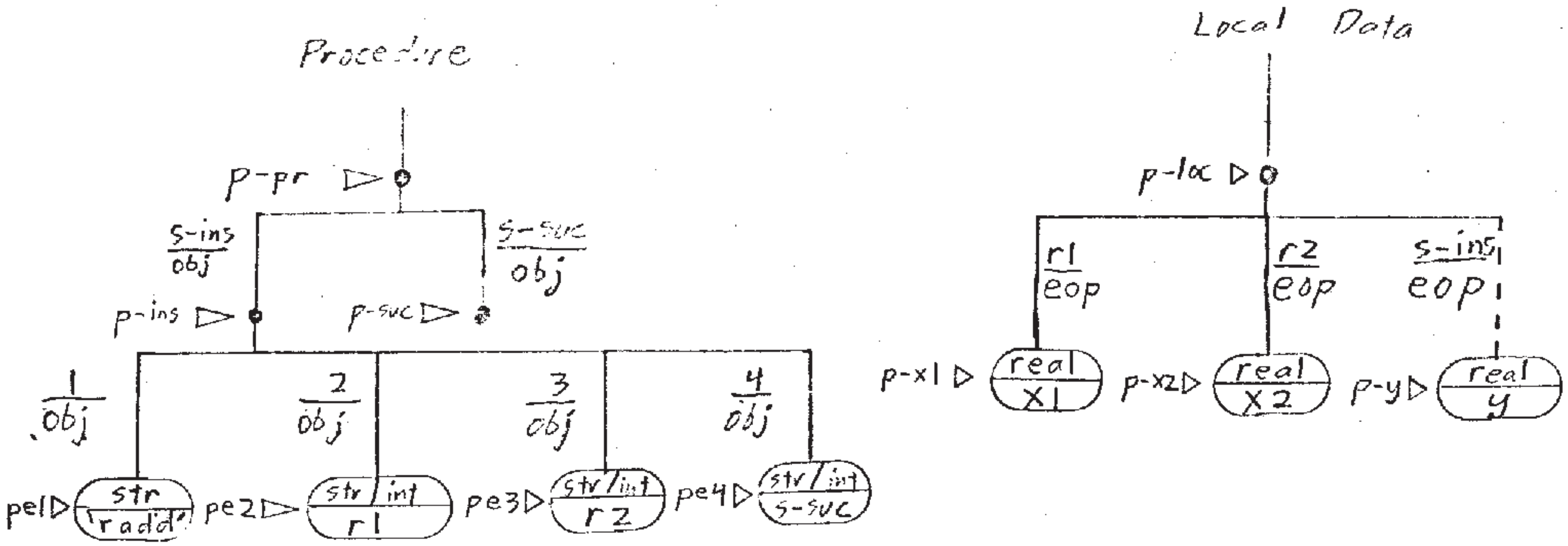
```
comment:  Control instructions for procedure activation and return.

create:   get-selector 2 → sel

          get-selector 3 → s-suc

          match (*, p-loc, sel, *) else goto 1

          goto 2

      1:  item

      2:  code → p1

          write ('wptr', p-loc, sel, p1)

          goto enable

exec:     get-selector  2 → s-pr1  [See Diagram]
          get-selector 3 → s-warg
          get-selector 4 → s-rarg
          obtain (p-loc, s-pr1, 'pptr') → p-pr1
          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-loc1

          write ('lstr', p-loc, s-ins, p-loc1)

          item

          write ('loc', p-loc1, p-loc)

          item

          write ('prc', p-loc1, p-pr)
          item
          write ('ret', p-loc1, s-ins)
          item
          write ('wptr', p-loc1, 0w, p-warg)
          item
          write ('rptr', p-loc1, 0r, p-rarg)
          item
          write ('ena', p-loc1, '1', p-pr1)
          goto interp
```

```
end:  match ('loc', p-loc, *) else quit
      read → (-, -, p-loc0)
      match ('prc', p-loc, *) else quit
      read → (-, -, p-pr)
      match ('ret', p-loc, *) else quit
      read → (-, -, s-ins)
      match ('lstr', p-loc0, s-ins, p-loc) else quit
      purge
      p-loc0 → p-loc
     get-selector 5 → s-suc
     goto enable
```

vapp[end] in Ver 1



An Instruction in Ver 1
Lang 'radd'



Syntax	Initial State-item's			
process-item	'ena'	p-loc	s-ins	p-pr
gen-struct-item	'obj'	p-pr	s-ins	p-ins
gen-struct-item	'obj'	p-ins	'1'	p-e1
elem-obj-item	'str'	p-e1	'radd'	
	'obj'	p-ins	'2'	p-e2
	'str/int'	p-e2	r1	
	'obj'	p-ins	'3'	p-e3
	'str/int'	p-e3	r2	
	'obj'	p-ins	'4'	p-e4
	'str/int'	p-e4	s-suc	
	'obj'	p-pr	s-suc	p-suc

Initial State-item's

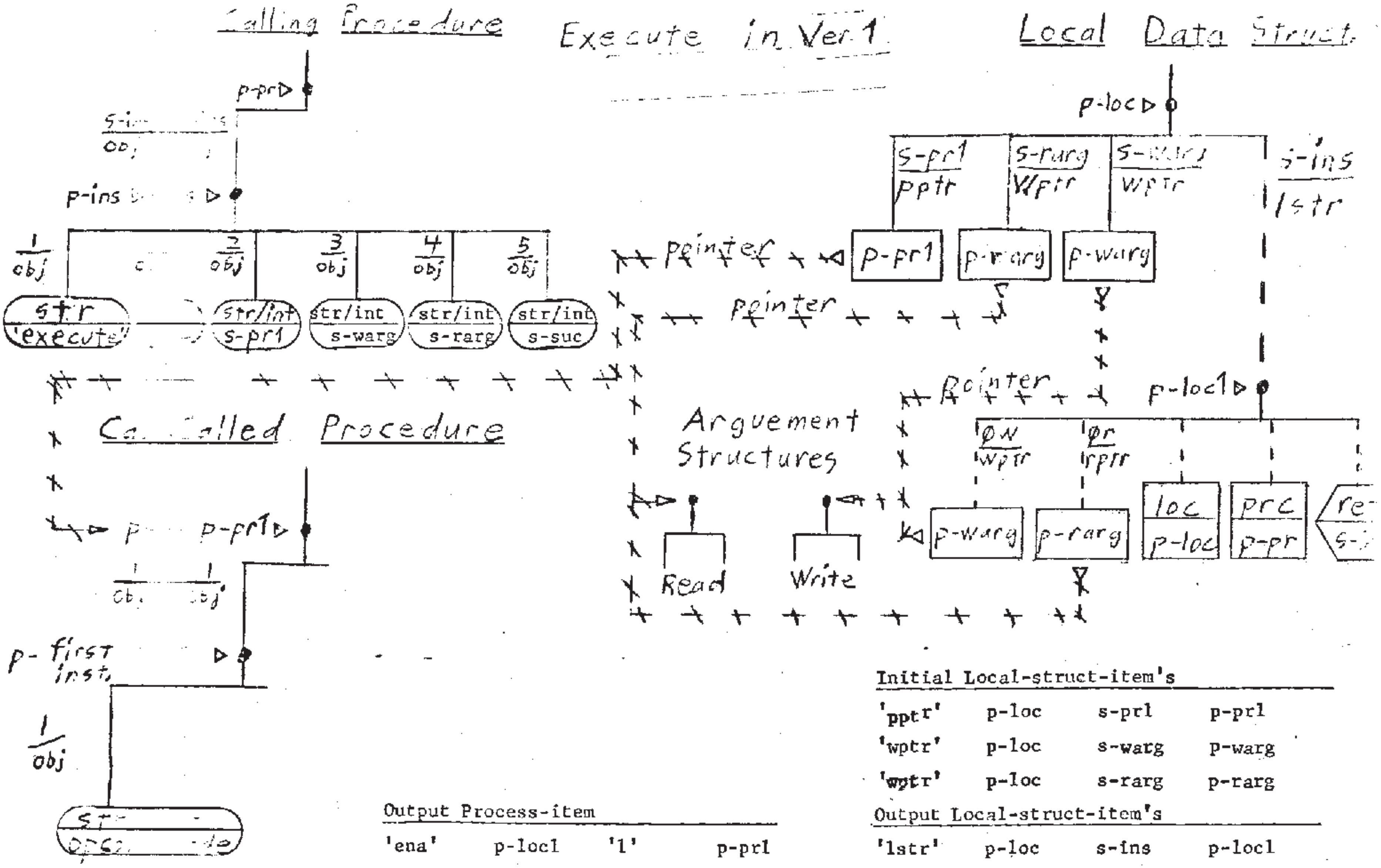
'eop'	p-loc	r1	p-x1
'real'	p-x1	x1	
'eop'	p-loc	r2	p-x2
'real'	p-x2	x2	

Output State-item's

'eop'	p-loc	s-ins	p-y
'real'	p-y	y	

PJF 3/26/71

Execute in Ver1



Output Process-item

'ena'	p-locl	'1'	p-prl

Initial Local-struct-item's

'pptr'	p-loc	s-prl	p-prl
'wptr'	p-loc	s-warg	p-warg
'wptr'	p-loc	s-rarg	p-rarg

Output Local-struct-item's

'lstr'	p-loc	s-ins	p-locl
'loc'	p-locl	p-loc	
'prc'	p-locl	p-pr	
'ret'	p-locl	s-ins	
'wptr'	p-locl	'Øw'	p-warg

P J F
2/28/71