MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 85

The Semantic Specification of SNOBOL in the Common Base Language

by

Caleb Drake

This work was submitted for credit in Subject 6.534, "Semantic Theory for Computer Systems," Spring 1973.

June 1973

1. Introduction

The question of what constitutes a good semantic definition of a programming language (or a semantic definition at all !) is one which I will avoid. I will be using the common base language (CBL; see (D1)) because its goals seem to fit what I will be doing (especially translating out language features to acheive a uniformity across languages) and its formal object is quite suitable for the most important data type in SNOBOL. To the extent that CBL is a model for some future generation of machine architecture, what I present here may be considered as useful information for writing the code-generation portion of a SNOBOL translator; however, I have always aimed for clarity and precision rather than presenting implementation hints.

2. A Basis for String Manipulation

We can look at strings and string manipulation in SNOBOL as a set of structures built up from a set of elementary objects. The set of elementary objects is the set of characters, the objects "succeed" and "fail" and the null string.

We wil consider the following types (there are also labels, arrays, integers, fixed point and floating point numbers and programmer-defined data types):

1) Decisions: The elementary objects "succeed" and 'fail" are decisions. The logical truth values "true" and "false are included in their meaning. For example the predicate "greater than" returns a decision as its value and that decision may be used to direct contral flow in a program. They are also used to indicate whether a computation can be performed (e. g. when a function is not applicable to a list of arguments it may return the value "fail" or an array reference may fail if an index is out of bounds).

2) Characters: This is the alphabet which belongs to a particular implementation. The things that can be done to a character are to read it, write it, ask whether it is equal to another or compare it in lexicographic ordering.

3) Strings: A string is an ordered list of characters. Basic functions for composing and breaking up strings will be discussed below. More complicated operations on strings (e.g. searching for substrings with certain properties) are performed using pattern matching.

4) Pattern: A pattern is a structure that defines a set of strings accepted by that pattern. A string when it is used as a pattern accepts only strings that are character-for-character equal to itself; i.e. it specifies a set with one member.

We have selector and constructor primitives for composing strings out of characters (analogous to "car", "cdr" and "cons" in LISP, of course):

first(string): the first character of the string. (i.e. the logical type of this function is (strings →characters).) It fails if the string is the null string.
rest(string): the string which is "string" with its first character deleted. (logical type: (strings →strings).) This function returns the null string as its value if "string" has only one character and fails if "string" is the null string.
concat(character,string): the string which is "string" with "character" concatenated on its front end. (Logical type: (characters X strings →strings).)
The notion of the length of a string can be defined as the number of appliactions of "concat" in its definition.

The following usual axioms hold:

i) first(concat(char,str)) = char

ii) rest(concat(char,str)) = str

iii) concat(first(str),rest(str)) = str (unless str is the null string)

The primitives that are necessary are:

eqch?(char1,char2): succeeds if char1 and char2 are the same character; fails otherwise.

null?(str): succeeds if str is the null string.

We can define type conversion functions using the above:

string(char) = concat(char,null)

char(str) = if null?(rest(str)) then first(str) else fail

Here we note how concatenation and testing of equality of strings can be defined in terms of the above:

```
concats(str1,str2) =
   if null?(str1) then str2
      else concat(first(str1),concats(rest(str1),str2)
eqst?(str1,str2) =
   if null?(str1) then if null?(str2) then succeed  else fail
      else eqch?(first(str1),first(str2)) and
           eqst?(rest(str1),rest(str2))
```

where "and" in the above is defined for decisions analogous to the way it is defined for truth values.

As we mentioned above the type string is in some sense a special case of the type pattern. The function "concats" is of logical type (strings X strings →strings). The question is whether we want to define patterns as being composed of characters so that string concatenation is a special case of pattern concatenation. The distinction is that there is a certain amount of homogeneity that is present in concatenation that is not present for patterns. That is, when two strings are concatenated they behave as one string; when two patterns are concatenated the point of concatenation

has some independent importance in pattern-matching as a point at which a decision may have to be made or unmade (upon backtracking) or a side effect created.

3. SNOBOL

(All information obtained about SNOBOL and all references to the SNOBOL manual can be found in (GPP1).)

I have decided to work on the semantics of SNOBOL because work on semantics of computer languages has concentrated on numerical languages. I will not talk about instruction sequencing, definition and application of procedures, arithmetic operations recursion, etc. since these have been discussed in the literature. There are two features in SNOBOL that are considerably different from what we see in algorithmic languages. They are pattern matching and indirect reference through construction of a name.

Although a pattern is "really" just a specification of a set of strings and pattern matching is "really" just an examination of the subject string looking for a substring in that set these notions do not capture the structure of the situation involved. Patterns are constructed from other patterns by "alternation" (syntactically "|") and "concatenation" (syntactically juxtaposition). For example, in

P1 = 'ABC' (the string: concat(A,concat(B,concat(C,null))))

P2 = 'DE'

P = (P1 | P2) P1

P will match 'ABCABC' or 'DEABC' (the alternatives being tried in that order). If the value of P2 changes at some time later, P will remain the same. In addition to strings there a set of system-defined functions available (e.g. LEN(N) which matches any string N characters long) for use in pattern construction.

The intuitive explanation of how pattern matching is done in the SNOBOL manual is that of a "needle" finding its way through a "bead diagram". For example the pattern:

(('P' | 'Q') 'R') ('S' | 'T')) ('U' | 'V')

which matches the strings 'PRU', 'QRU', 'PRV', 'QRV', 'SU', 'SV', 'TU' and 'TV' has the bead diagram in figure 1. The way pattern matching works is that the "cursor" is pointing at the first letter of the subject string; the needle attempts to find a path through the diagram such that "if a bead matches the needle passes through and moves upward as far as it can go without crossing a horizontal line. If a bead does not match, the needle moves down to an alternate bead provided one exists. Downward movement may not cross a horizontal line. If no alternative exists, the needle is pulled back through the last successfully matched bead and an alternative is sought there." If this all fails the cursor is moved right one position and the

process starts again. Pattern matching succeeds if the needle finds a path and fails if the cursor runs out of characters. The reader no doubt understands exactly what is happening here, but we would like to explain it to a computer (for program verification, for example) and relate it to other notions having to do with computation.

The other feature of SNOBOL we will consider is its indirect reference operator (syntactically "$"). For example:

```
S1 = 'HOT'
S2 = 'DOG'
HOTDOG = 'NITRATE'
S3 = $(S1  S2)
```

assigns the string 'NITRATE' as the value of the variable S3 (string concatenation is also symbolized as juxtaposition). This operation is obvious enough as a manipulation of the kind of information that a computer will have available in translating and executing a programming language, but as a semantic notion it is somewhat peculiar. When defining programming languages names are usually considered to be atomic (like variable names in mathematics) or are not considered at all since the only mappings involving them are assignment and evaluation and these can be translated out using notions like "store" and "state-vector". It is the fact that names can be constructed that makes this situation different from that of indirect referencing involving pointers. It seems that the type "identifier" is not only a part of the meta-language, but is a type for objects in the language in the sense that "$" has as its range the set of valid identifiers. However, the endpoint of evaluation is still an object in the language or a memory location or state (depending on whether it occurs in a context which requires an L-value or an R-value) since the identifier is evaluated as soon as it is determined.

4. In the Base Language

In drawing up the translation functions I have found that it is easiest to write them as recursive functions. This is not in the spirit of the base language; however, one could apply Amerasinghe's (A1) technique for handling recursive procedures to these definitions to obtain texts in the base language which could then be executed by the interpreter. We will augment the set of elementary objects with the objects STR, CHAR and PTRN which will be used to mark objects of types string, character and pattern respectively.

We will use the following primitives (all paths must begin in the local struc-

ture corresponding to that procedure activation):

a) const elementary object, path

My usage of this is different from that of Dennis (D1) in that here if some portion of the path does not exist, it is created.

b) share path1, path2

Path1 is assumed to exist. The object pointed to by the last selector of path1 is now pointed to by the last selector of path2. If part of path2 does not exist, appropriate selectors are created; if there are selectors below path2 already, they are deleted and the structure below might be garbage collected. If at some time later the structure pointed by both paths is modified (by a const or copy) through either path. both paths will point to the new object (hence the name "share"; see figure 7 of (D1)). Two previous instructions are special cases of share:

select p,n,q = share p.n,q

link p,n,q = share q,p.n

c) copy path1,path2

This operation copies the structure below path1 to below path2. (Again arcs of path2 may have to be created.)

d) test path

This predicate succeeds if the path exists, fails otherwise.

e) eq elementary-object1, elementary-object2

This predicate succeeds if the two objects are the same.

f) if predicate then operations1 else operations2

g) return path

This indicates a constructed path returned by a function. Presumably, it will be translated out to an argument being passed when these recursive procedures are translated into standard CBL applications.

[redacted] text, arg

[redacted] use these as my primitive operations; it is likely that some of them [redacted] composed of more elementary operations (e.g. an operation to create selectors [redacted] missing in a path or for a copied object). We will have occasion to write [redacted] composition whose value is a selector path as an argument to one of the [redacted] can be eleminated by using some conversion functions introduced later [redacted] temporaries in the local structure.

[redacted] string 'ABC' will represented as in figure 2. We have the following defini- [redacted] altering operations:

```
concat(char,string) =
   if eq char.type, CHAR  then
        if eq string.type, STR then
             const STR, L.$temp.type
             copy char, L.$temp.value.first
             copy string, L.$temp.value.rest
             return L.$temp
          else FAIL
      else FAIL
```

In the above, "L" refers to the local structure within which this function is applied.

```
first(string) =
   if eq string.type, STR  then string.value.first  else FAIL
rest(string) =
   if eq string.type, STR  then string.value.rest  else FAIL
concats(string1,string2) =
   if eq string1.value, NULL  then string2
      else concat(first(string1),concats(rest(string1),string2))
```

Note that the predicates eqch? and null? as well as type checking are all translated into uses of eq with the appropriate elementary object. For the remainder of this paper, strings wil be drawn in diagrams and written in functions in the same fashion as elementary objects (to save space and effort).

A. Assignment and Indirection

As we saw in the last section we want to structure our environment in such a way that we can reference values by the character string for the associated identifier. The object in figure 3 represents the environment after preforming the following assignments:

XL = 'AH' A = 'BK' AHA = 'P' BK = XL XC = A 'R' AHA

That is, each time a variable is introduced the letters in its name are used as selectors; finally, the selector "id" gives the type-value pair corresponding to that variable. One may think of this as the relevant portion of a tree that exists in full (SNOBOL defaults all variables to the null string) or as that structure which is built up so far. We will operate within the framework of the former. Now execution of the statement:

$(XC) = $(A) $(XL 'A')

will sprout a selector "r" on the path env.b.k, then a selector "p" on the result and will assign 'AHP' as its value (with type STR).

Clearly to be able to do this we must be able to talk about paths and in particular to construct them. We will have the data type indicators SLC (for selector) and PATH (for path) and the constructor function "concatp(path, ~~selector~~ string)". The definitions of these are analogous to those of CHAR, STR and concat except that where strings form a right- branching structure, paths form a left-branching structure and selector.value is a typr value pair for a string while first.value is a character object. We will use the elementary object NULL to indicate the front end of a path.

concatp(path, ~~selector~~ string) =

```
if eq path.type, PATH
   then if eq selector.type, STR  then
            const PATH, L.$temp.type
            copy path, L.$temp.value.path
            const SLC, L.$temp.value.selector.type
            copy string, L.$temp.value.selector.value
            return L.$temp
         else FAIL
   else FAIL
```

We must draw a distinction between the data-type PATH and a path through an object (with infix concatenation operator ".") which points to a subobject or an elementary object. So we need a conversion function which takes an argument of type PATH and produces the corresponding pointer (a path from the local structure) to an object. To define such a function we need a primitive type conversion function which we will call "selector(string)" whose value is the selector which is identified by the string.value.

evalpath(path) =

```
if eq path.type, PATH then
     if eq path.value, NULL then NULL
          else if eq evalpath(object.path), NULL  then selector(path.selector)
                    else evalpath(object.path).selector(path.selector)
```

We will assume that syntactic routines can be written which translate a SNOBOL assignment statement into a composition of the functions "assign", "name" and "concats". The first and third are obvious. "$('AB')" is translated to "name('AB')", as is "AB". "$(AB)" is translated to "name(name('AB'))". The statement at the bottom of page 6 is translated to:

```
assign(name(name('XC')),
       concats(name(name('A')),name(concats(name('XL'),'A'))))
```

The texts to be interpreted for each function are:

```
concats(obj1,obj2) =
   if eq obj1.type, PATH  then concats(evalpath(obj1),obj2)
      else if eq obj2.type, PATH  then concats(obj1,evalpath(obj2))
              else if eq obj1.value, NULL  then obj2
                      else concat(first(obj1),concats(rest(obj1),obj2))
name(obj) =
   if eq obj type, PATH  then name(evalpath(obj))
      else if eq obj.type, STR
              then const PATH, L.$temp.type
                   const NULL, L.$temp.value
                   concatp(concatp(concatp(L.$temp,'L'),'ENV'),nm(obj))
              else output ERROR: ILLEGAL DATA TYPE
nm(string) =
   if eq string.value, NULL
      then const PATH, L.$temp.type
           const NULL, L.$temp.value
           concatp(L.$temp,'ID')
      else const STR, L.$temp.type
           const NULL, L.$temp.value
           concatp(nm(rest(string)),concat (first(string),L.$temp))
assign(path,obj) =
   if eq obj.type, PATH  then assign(path,evalpath(obj))
         else copy object,evalpath(path)
```

"concats" has been modified so that either of its arguments may be a path. This is to provide for concatenating identifier values and literal strings with the same function. Similarly "name" does this so that we can use the same function for the operator "$" and normal identifiers. "assign" does it so that the right-hand side may be an identifier or a value. (The alternative of haveing "name" do an "evalpath" before passing its value does not work because we need an identifier for the left-hand side of "assign".) The sequence "const ... , const ... " in "name" and "nm" forms a null path or string to use as a starting point for the recursively constructed object. "nm(string)" constructs an object of type PATH with selectors corresponding to the letters of the string whose last selector is "id" (for retreiving the identified object). "name(obj)" takes this object and forms a PATH object corresponding to an access path from the local structure fo the identified object. Note that all created objects (in particular the one for the right-hand side of "assign") go away

upon function exit (except if they occur in a return operation) since the local structure for that function call will be deleted.

B. Patterns and Pattern Matching

We might look at an attempt to match the pattern in figure 1 as an attempt to successfully find a path through the graph of figure 4. The relevance of a CBL object (as a tree with the possibility of sharing subtrees) is immediately apparent; alternation is represented by branching and alternation by sequence. When an alternation is concatenated with another pattern then the alternatives must share the following structure.

We will not discuss the operation of the many system-provided pattern matching functions; we will leave a place for them in the pattern structure and assume that each text is bound to the appropriate path in the environment. Figure 5 illustrates the object we will be constructing for the example pattern:
('A' | 'C') LEN(4)
The "next" selector is the way we have of translating out concatenation of patterns. Patterns which are alternations do not need this selector because it will be present in each of the alternatives.

An important feature of SNOBOL we would like to include is the possibility of inducing side effects by a pattern matching operation. Syntactically this is represented by the binary operators "." and "$" as in the following:
((P | Q) . V1 'R') $ V2 ('S | T)
"." (conditional assignment) causes the substring matched by the pattern on the left to be assigned to the variable on the right if the entire pattern match succeeds. Here if the match succeeds, that portion which matched P or Q (whichever was matched) is assigned to V1. "$" (immediate assignment) causes the substring matched by the pattern on the left to be assigned to the variable on the right as soon as this match occurs. In the above if P 'R' matches V2 will be assigned a value: if at this point neither 'S' nor 'T' matches, the alternative Q and then 'R will be tried. If this matches, the value of V2 will be updated again.

Again we will assume the existence of syntactic routines which parse the pattern into a composition of semantic routines. They will be "alt", "conc", "openc" and "closec" (for the beginning and end of a conditional assignment; the first argument is the pattern corresponding to the beginning or one after the ending of the sequence of patterns to be associated with that variable (the end of the pattern implicitly closes all variables that are to be assigned up to the end), the second is the variable), "openi" and "closei" (similarly for immediate assignment) and "func" (for special functions; the first argument is the name of the function, the second is the argument or the name of the argument). For example, the above pattern translates

to:

```
conc(openi(conc(openc(alt(name('P'),name('Q')),name('V1')),
                closec('R',name('V1')),name('V2')),
     closei(alt('S',name('T'),name('V2')))
```

We will need a list of the alternatives constructed in the first argument of a "conc" so that each "next" may be linked to the structure resulting from the second argument. We will use a stack for this because we will need it for backtracking in pattern matching and that way we don't have to introduce two new types. I won't go into defining the stack (it is straightforward enough as the CBL equivalent of a linked list with an access pointer at one end) but will assume that we have the usual operations push stack, object, pop stack and top(stack). We will use NULL for the bottom element and write const NULL, stack to initialize the stack.

The definitions are (we will not write clauses to take care of improper arguments):

```
pattern(string) = const PTRN, L.$temp.type
                  const EQST, L.$temp.value.func
                  copy string.value, L.$temp.value.arg
                  return L.$temp
func(path,arg) =
   if eq arg.type, PATH  then func(path,evalp(arg))
      else copy path, L.$temp.value.func
           copy arg, L.$temp.value.arg
           const NULL, L.$temp.value.next
           const PTRN, L.$temp.type
           return L.$temp
openc(pat,name) =
   if eq pat.type, PATH  then openc(evalp(pat),name)
      else if eq pat.type, STR  then openc(pattern(pat),name)
              else copy name, pat.openc
                   return pat
```

"pattern(string)" converts a string to the corresponding pattern. This will happen every time that a string is used in a context which requires a pattern (that is, whenever anything besides concatenation with another string happens). We have used EQST as an elementary object which will trigger a matching routine because this operation is not seen explicitly by the user and we don't want modification of the environment to remove this function. (The user is allowed to redefine the names used for system functions.) "function" creates the appropriate pattern structure. The "next" selector in both cases will be reset by other routines; "evalp" will be

defined below. "openc" attaches a selector by that name which points to a copy of the path name to the pattern involved. "closec", "openi" and "closei" are defined similarly. Note that by this definition only one variable of each type may be opened or closed at any point in the pattern (except at the very end). This can be generalized so that these selectors point to a list but we wil not do this here because the extra machinery required would not further illuminate the task at hand.

```
alt(obj1,obj2) =
   if eq obj1.type, PATH  then alt(evalp(obj1),obj2)
      else if eq obj2.type, PATH  then alt(obj1,evalp(obj2))
            else if eq obj1.type, STR  then alt(pattern(obj1),obj2)
                  else if eq obj2.type, STR  then alt(obj1,pattern(obj2))
                        else const PTRN, L.$temp.type
                             const ALT  L.$temp.value.func
                             copy obj1, L.$temp.value.arg1
                             copy obj2, L.$temp.value.arg2
                             return L.$temp
conc(obj1,obj2) =
   if eq obj1.type, PATH  then conc(evalp(obj1),obj2)
      else if eq obj2.type, PATH  then conc(obj1,evalp(obj2))
            else if eq obj1.type, STR  then if eq obj2.type, STR  then concats(obj1,obj
                                                else conc(pattern(obj1),obj2)
                  else if eq obj2.type, STR  then conc(obj1,pattern(obj2))
                        else link(obj1,obj2)
link(obj1,obj2) =  const NULL, L.stack
                   links(obj1,stack)
                   connect(stack,obj2)
links(obj1,stack) =
   if eq obj1.value.func, ALT  then links(obj1.value.arg1,stack)
                               links(obj1.value.arg2,stack)
      else if eq obj1.value.next, NULL  then push stack, path(obj1.value.next)
           else links(obj1.value.next,stack)
connect(stack,obj2) =
   if eq top(stack), NULL then
      else snare obj2, evalpatn(top(stack))
           pop stack
           connect(stack,obj2)
```

"alt" constructs an object as in figure 5. "conc" uses "concats" if both objects are strings ("concats may revert back to its former definition since "conc" does the path evaluation) otherwise it calls "link". "link" points all of the free "next" pointers (collected by "links") in obj1 to obj2 (using "connect"). The primitive function "path" takes a path composed of selectors and produces the corresponding object of type PATH. The function "evalp" is used in all the above procedures because when retreiving a pattern from the environment all the final "next" selectors point to SUCCEED. We want a pattern in which these point to NULL. So we have the following:

```
evalp(path) =
   if eq evalpath(path).type, PTRN  then copy evalpath(path),L.$temp
                                         nullify(L.$temp)
                                         return L.$temp
      else evalpath(path)
nullify(pat) =
   if eq pat.value.func, ALT  then nullify(pat.value.arg1)
                                   nullify(pat.value.arg2)
      else if eq pat.value.next, SUCCEED  then const NULL, pat.value.next
             else nullify(pat.value.next)
```

Now that al this is finished we still have a pattern with some "next" selectors pointing to NULL. Also, where is the front of the pattern tied down? A pattern may be used in two types of statements: the pattern match statement and the assignment statement. We will discuss the former below. In the latter the root of the pattern will be attached to the local structure for the "assign" procedure. This procedure will also link the unlinked selectors to SUCCEED. This must be done by the assignment procedure because no other can know where the pattern ends since they all operate on such local evidence. The modified "assign" procedure is:

```
assign(path,obj) =
   if eq obj.type, PATH  then assign(path,evalp(obj))
      else if eq obj.type, PTRN  then link(obj,SUCCEED)
                                      copy obj, evalpath(path)
             else copy obj, evalpath(path)
```

The basic match routines (without side effects) are:

```
pattern-match(string,pattern) =
   const NULL, L.stack
   if match(string,pattern, L.stack)  then SUCCEED
      else if eq string.value, NULL  then FAIL
             else match(rest(string),pattern,L.stack)
```

```
match(string,pattern,stack) =
   if eq pattern, SUCCEED  then SUCCEED
      else if eq pattern.value.func, ALT
             then copy string, L.$temp.string
                  copy path(pattern.value.arg2), L.$temp.alt
                  push stack, L.$temp
                  match(string,pattern.value.arg1,stack)
             else if eq pattern.value.func, EQST
                    then if eq prefix(string,pattern.value.arg).decision, FAIL
                           then backtrack(stack)
                           else match(prefix(string,pattern.value.arg).left,
                                      pattern.value.next,stack)
                    else if eq (apply evalpath(pattern.value.func), pattern.value.arg)
                                .decision, FAIL
                           then backtrack(stack)
                           else match((apply evalpath(pattern.value.func),
                                             pattern.value.arg).left,
                                      pattern.value.next,stack)
backtrack(stack) =
   if eq top(stack), NULL  then FAIL
      else copy top(stack), L.$temp
           pop stack
           match(L $temp.string,evalpath(L.$temp alt),stack)
prefix(string1,string2) =
   const SUCCEED, L.$temp.decision
   const STR, L.$temp.matched.type
   const NULL, L.$temp.matched.value
   pfx(string1,string2,L.$temp)
   return L.$temp
pfx(string1,string2,obj) =
   if eq string1.value, NULL  then const obj.decision, FAIL
      else if eq string2.value, NULL  then copy string2, object.left
             else if eq first(string1).value, first(string2).value
                    then copy concats(obj.matched,concat(first(string1),NULL)),
                              obj.matched
                         pfx(rest(string1),rest(string2),obj)
                    else const FAIL, obj.decision
```

"pattern-match" is the top level routine; it creates a stack which is always passed as an argument to "match" and "backtrack" so that all invocations of these use this same stack; the level of recursive call of "match" within "pattern-match" corresponds to the cursor. "match" stacks alternatives (which consist of an object with selectors "string" for the string to be matched upon backtrack and "alt" for the pattern to be used) for ALT and handles decisions passed up from "prefix" or system functions. "backtrack"checks the stack for alternative paths through the pattern; if "match" fails it is because "backtrack failed to find alternatives upon failure of the basic matching functions. "prefix" returns an object with selectors "decision" (i.e. SUCCEED or FAIL), "matched" (the portion of the subject string matched) and "left" (the portion of the subject string still to be matched if necessary). Note that if "decision" is FAIL, the other two are never examined. It is assumed that system functions return this same type of object.

We will now discuss the modifications necessary to do the conditional and immediate assignments. (The detailed procedure would certainly not be worth the effort.) We will keep two stacks: one for conditionals and one for immediates. That these are stacks is not necessary, but it is convenient since the last variable opened will be the next variable closed (since they have to be nested or disjoint; in the general case where a group may be closed at once,the variable only has to be among this group). Each stack will be an object with a selector "name" and a selector "value" (which is initialized to the null string). A pair is entered onto the appropriate stack upon the occurance of an "openi" or "openc" selector on any of the three types of subpatterns. (We need the CBL primitive test to check for this.) When a "closei" is encountered, the pair is popped off the stack and the assignment is made (the name is necessary only in the general case). When a "closec" is encountered the pair is put on another list of variables to be assigned by "pattern-match" if "match" returns SUCCEED. Values are built up when either EQST or a system function match some substring; that is, the string pointed to by the "matched" selector of the object returned by the function call is concatenated onto all the "value" selectors of all objects in both stacks just before going on to the "next" part of the pattern. When the function ALT in encountered, a marker is placed on the stack so that all variables opened along that alternative may be closed if it fails. Similarly a marker must be concatenated on to the "value" of each pair so that characters added on after a decision was made may be deleted. This closing of variables and deletion of characters is handled by the function "backtrack". We also need a marker on the list of completed conditional assignments because if a conditional assignment variable is opened and closed along a path which eventually fails, we don't want to make the assignment. All this would be straightforward but tedious to encode in CBL.

5. Loose Ends

There are many ways that these basic procedures can be complicated to take into account other features of SNOBOL. For example there are many system functions which which use a numerical value for the position of the cursor in the string (e.g. TAB(N) matches up to the Nth character). So we would want an index variable in the loop in "pattern-match". There is a function "FENCE" which succeeds matching the null string but fails when backed into. (Perhaps this could be implemented by having FENCE clear the backtrack stack.) There are many functions with implicit alternatives (e.g. ARBNO(P matches an arbitrary number of concatenations of the pattern P) which must be handled similarly to ALT. There is the possibility of constructing a pattern from other patterns which will be evaluated analogously to call-by-name evaluation of procedure arguments. This allows for an implicitly changing pattern (e.g. for use in a loop). At first it seems that we might just use share instead of copy in the definitions of "conc" and "alt" for this. However, we don't want to be changing the final "next" pointer of a pattern in the environment (especially since more than one pattern may have this pattern as an unevaluated component). This also gives one the ability to write recursive patterns; it is not easy to see how this would fit in. One might also want to prove the correctness of these algorithms (although it may not be so easy to state their intentions), prove that "concats" as defined is associative, prove that 'BC' 'BD' has the same effect as 'B' ('C' 'D'), etc. It also possible that some study of these functions and representations may point out features of SNOBOL that may be executed concurently (or suggest extensions of the language in which the order of evaluation might be less restricted). In any case, I hope that I have shed some light on the semantic notions embodied in SNOBOL and the structure of the base language as a model for the meaning of computer languages.

Bibliography:

(A1) Amerasinghe, M. "Translation of BLKSTRUC Programs to the Base Language" Unpublished mimeo, Project MAC, MIT

(D1) Dennis, J. B. "On the Design and Specification of a Common Base Language" Computation Structures Group Memo 60, Project MAC, MIT

(GPP1) Griswold, Poage and Polonsky "The SNOBOL 4 Programming Language" Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1968

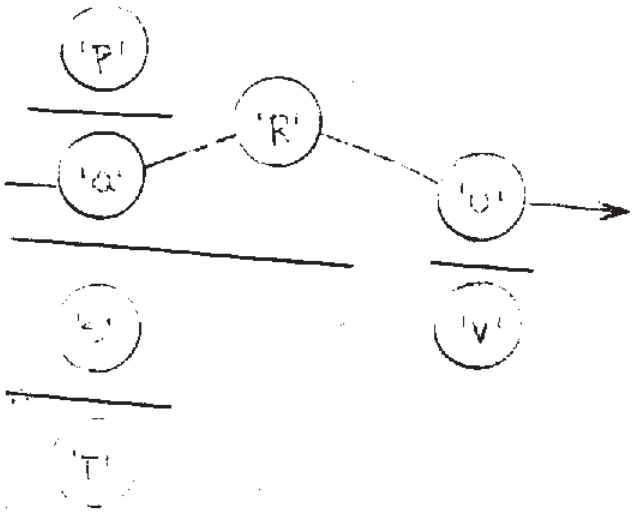

figure 1

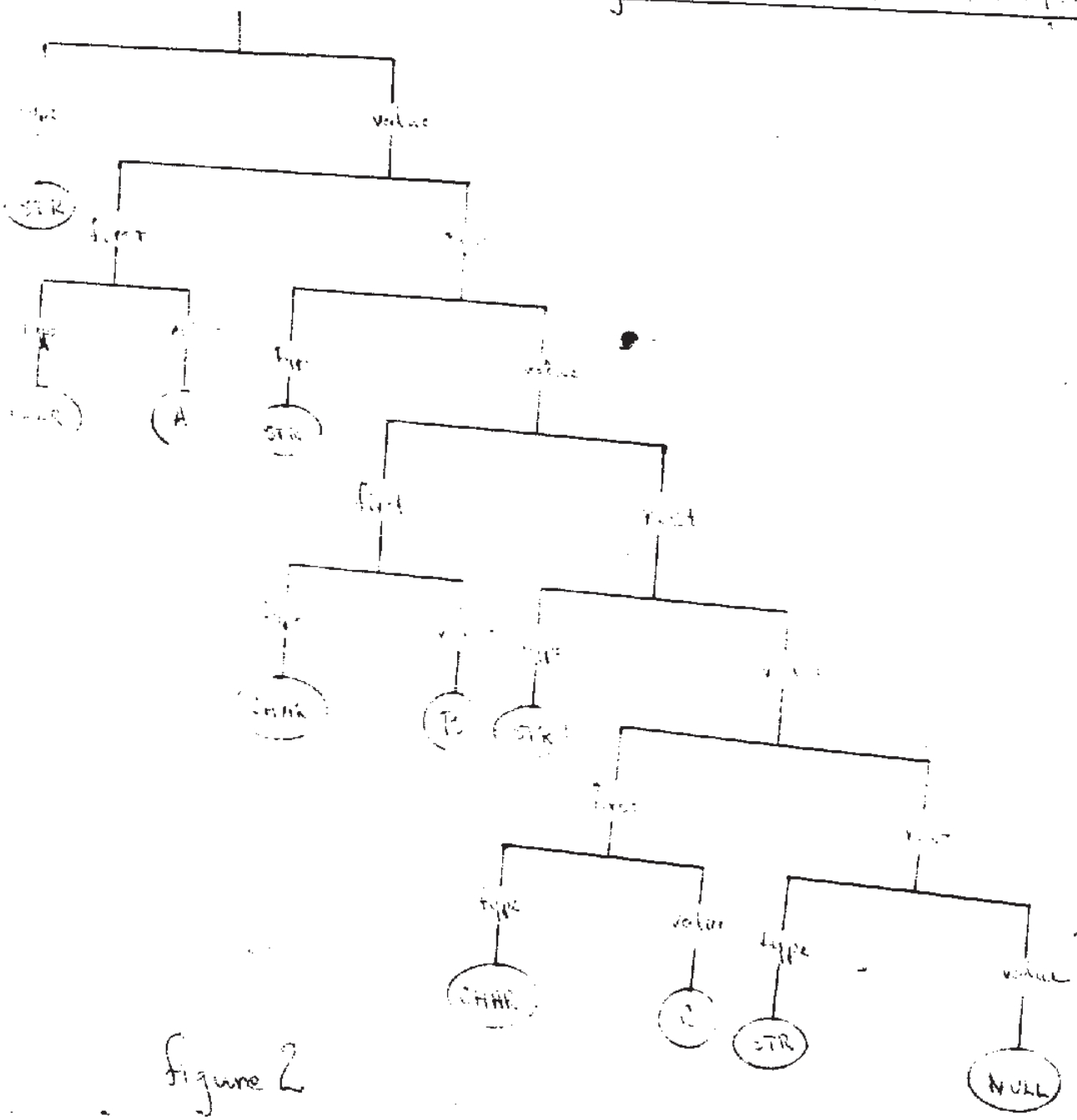

figure 2

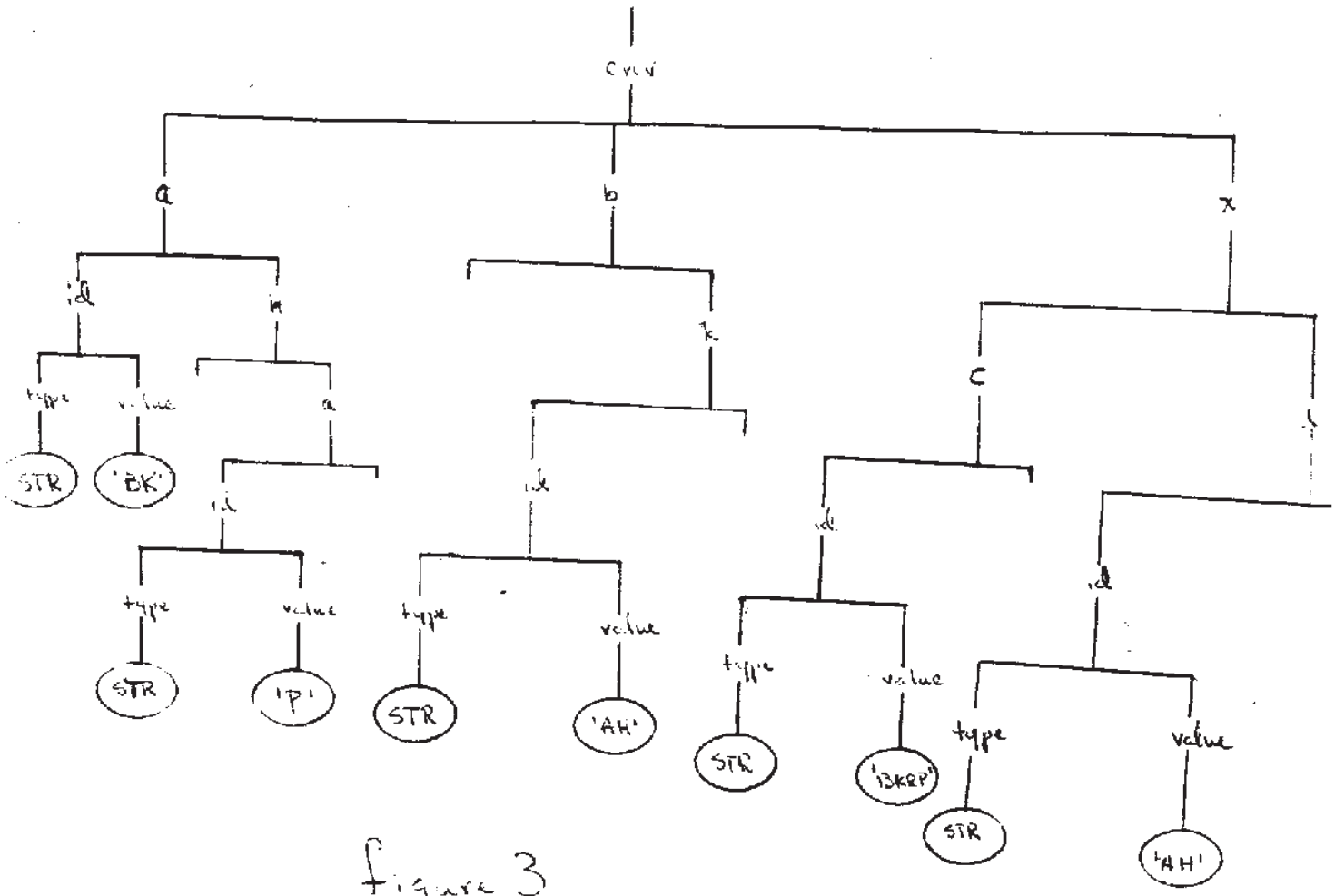

figure 3

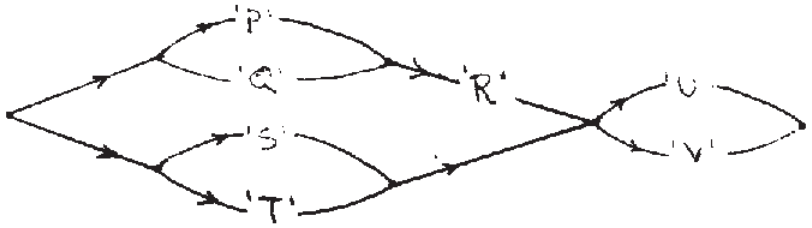

figure 4

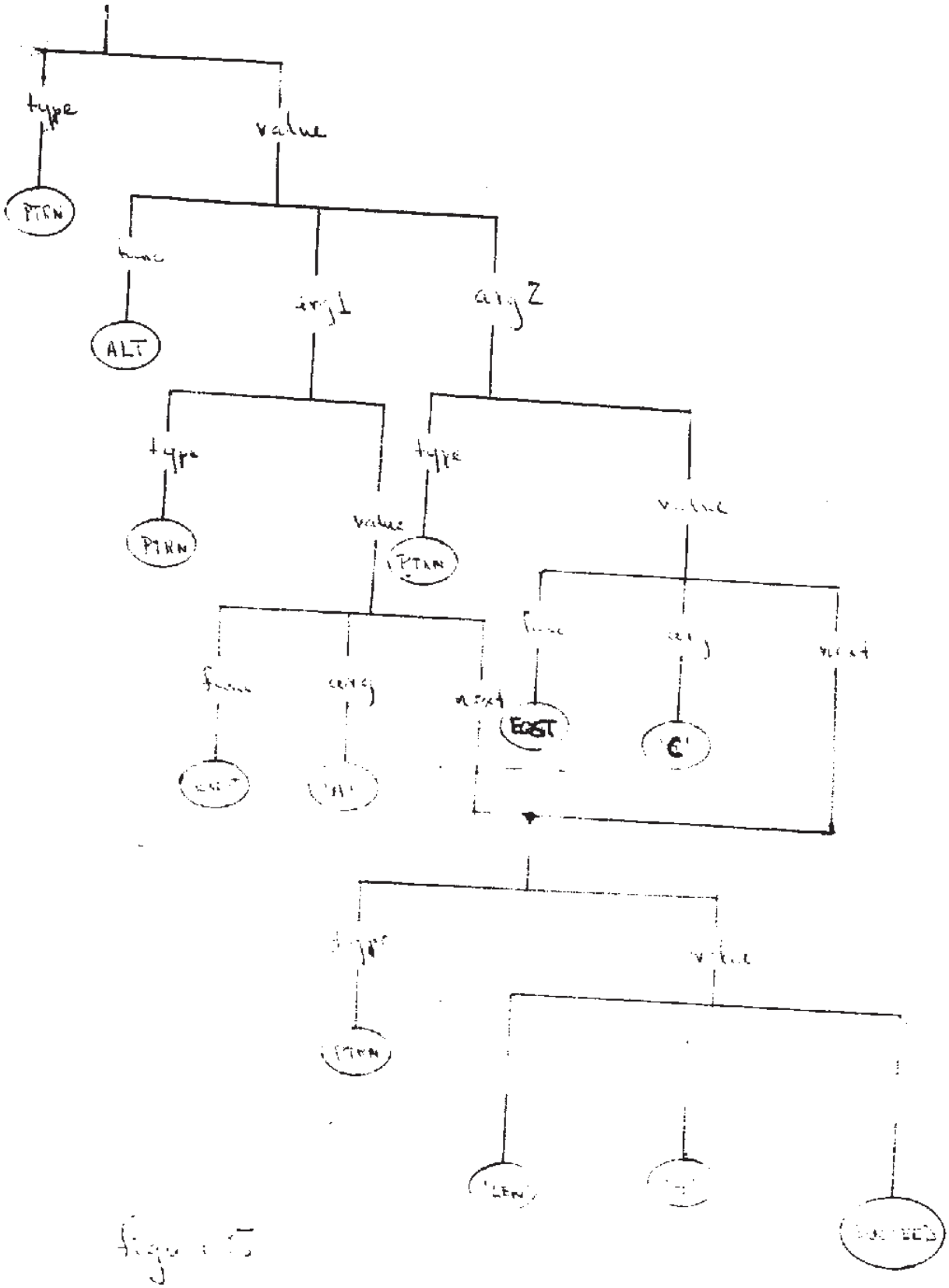
type
value
PTRN
func
arg1
arg2
ALT
type
type
value
PTRN
PTRN
EAST
'C'
'LEN'

Figure 5