The Structure of SNOBOL4

by

Peter Wegner

Technical Report
No. 68-9
February 1968

Depart. of Computer Science
Cornell University
Ithaca, New York 14850

The Structure of SNOBOL4

Peter Wegner

Cornell University

February 1968

The SNOBOL4 programming language was developed by Griswold, Poage and Polonski (1). It combines facilities available in problem oriented languages with string manipulation facilities, pattern matching facilities and facilities for compilation during execution. The present description attempts to accomplish three objectives. It is an introduction to SNOBOL4 for the programmer with previous programming experience in some programming language but not necessarily previous experience with SNOBOL3. It is intended also to provide dynamic insights into the mechanisms for statement execution, and to describe source language structures in terms of the information structures to which they give rise during execution. It is felt that dynamic insights into the way in which source statements are executed help not only the system programmer but also the average user.

This description is based on reference 1 and on two very worthwhile days of discussion with the authors of SNOBOL4 at Holmdel. The author is indebted also to John Kelly for fruitful discussion of aspects of SNOBOL4 and for proofreading the manuscript.

THE STRUCTURE OF SNOBOL4

Table of Contents

		Page
1.	Integers, Numbers and Strings	1
2.	Variables and their Values	2
3.	Simple Assignment Statements	5
4.	String Representation and String Accessing	6
5.	Evaluation of Simple Expressions	8
6.	Arrays	12
7.	Indirect Addresses and Composite Names	17
8.	Input and Output	21
9.	Labels and Conditional Branching	26
10.	Simple Pattern Matching	27
11.	The Order of Pattern Matching	33
12.	Programmer-Defined Functions	37
13.	Programmer-Defined Data Types	45
14.	Compilation During Execution	48
15.	Primitive Pattern Functions	52
16.	Matching of String Positions	61
17.	Assignment During Pattern Matching	64
18.	Deferred Pattern Evaluation	67
19.	Pattern Charts and Syntactic Recognition	73
20.	The Representation of Patterns	81
21.	Relational Functions and Deferred Expression Evaluation	86
22.	Keywords	90
	References	93

1. Integers, Numbers and Strings

SNOBOL4 combines facilities for numerical computation with facilities for string manipulation and pattern matching. In the present section we shall introduce data elements of the type INTEGER , REAL and STRING and give some examples of simple assignment statements in which data elements of these types occur.

The SNOBOL4 alphabet consists of the 26 capital letters, the ten decimal digits, the six characters + - * / = &, the five characters . , ; : ? , the single quote ' and the double quote " , the four parenthesis characters ( ) < > , the characters $ | and ¬, and the blank character. There are 57 characters altogether.

Integers are denoted by strings of digits, and real (floating point) numbers are denoted by integer strings which contain a decimal point. Integers and real numbers may be assigned as the values of identifiers, just as in procedure oriented languages.

```
  Y = 3
Y35 = 5.6
```

The first statement assigns the integer 3 as the value of Y . The second statement assigns the number 5.6 as the value of Y35 . The two statements above are examples of assignment statements.

The right hand side of an assignment statement may consist of an expression involving constants and variables.

```
Y = 5
Z = Y + 2
```

Execution of the second statement involves evaluating Y , adding 2 to the value of Y and assigning the result to Z .

In the above examples we have encountered integer constants and decimal number constants. Integer constants are said to be of the type INTEGER while decimal number constants are said to be of the type REAL .

Types are associated not only with constants but also with variables. The type of a given variable is given by the constant which

constitutes the current value. In the above examples Y and Z have the type INTEGER and Y35 has the type REAL .

SNOBOL4 allows constants and variables to be of the type STRING . Since a string which occurs on the right hand side of an assignment statement is treated as a variable and evaluated, string constants must be enclosed in single or double quotes.

```
X = 'ABC'
X = "ABC"
X = ABC
```

The first and second statements assign the string constant ABC as the value of X . The third statement assigns to X the value of the string variable ABC .

SNOBOL4 permits a number of additional data types each of which has an associated set of data constants which may appear as literals or be assigned as the values of variables. Each data type has an associated set of operations which are applicable to data constants of that type.

The form of data constants and expressions for each data type is discussed in the body of this report. However this report aims to describe not only the SNOBOL4 source language but also the way in which SNOBOL4 data objects are represented during a computation. Data types of SNOBOL4 permit considerably richer classes of data structures than numerical languages, and SNOBOL4 therefore must organize the representation of data structures differently from conventional languages. The basic framework for the representation of values of data objects and for the representation of strings is given in the next section.

2. Variables and their Values

Any programming language permits computations on data elements of a number of different types. Numerical programming languages like FORTRAN and ALGOL emphasized computations on real (floating point) and integer data. SNOBOL4 permits computations on a much richer class

of data types, including data of the type STRING and of the type PATTERN.

Data elements may be manipulated by operations of the programming language, and may be assigned as the values of variables. Different types of data elements give rise to different kinds of storage requirements during execution. Real and integer data types are usually representable in a fixed information field such as a memory register. However, operations on data objects of the type STRING and PATTERN frequently change not only the value but also the size of the information field required to represent an object. The form of representation of data objects in SNOBOL4 is governed by the fact that the data objects may be of unpredictable size.

In numerical languages it is usual to fix the data type associated with any given variable prior to execution. Fixing of the data type is accomplished either by a declaration for the variable which explicitly specifies the data type, or implicitly by the context in which the variable occurs. In SNOBOL4 , the data type associated with a variable is determined dynamically during execution whenever a value is assigned to the variable during execution. A given variable may have values of different types assigned to it at different points of execution.

SNOBOL4 relaxes both the restriction that data objects of a given type are represented by fixed size information structures and the restriction that the type of a given variable remains fixed throughout its execution. These two features of SNOBOL4 require a different representation of variables and their values than in traditional numerical languages. The form of representation used in SNOBOL4 for variables and their values is described below.

SNOBOL4 represents the values of all data objects by a fixed length information item called a descriptor. The descriptor is one of the most fundamental information structures in SNOBOL4 Every descriptor

contains a value field for specifying either a literal value or a pointer to the value, a type field for specifying the type of the value, and a mode field for specifying usage information useful to the SNOBOL4 system.

Pointer to Value or literal value	Usage information for garbage collection	Type of Value
value field	mode field	type field

Figure 1. Format of a Descriptor

Values of certain fixed length classes of data objects (such as integers) are represented by literals in the value field, while values of variable size data objects are represented by pointers which point to a representation of the value. The representation pointed to by a descriptor is referred to as the data reference block of the object being represented.

Values in SNOBOL4 may be of the type INTEGER , REAL , STRING , ARRAY, PATTERN , NAME , CODE , or of a type defined by the programmer. Each of the above data types has an associated set of operators by means of which data objects of that type are manipulated. Each of the types also has a differently structured data reference block by means of which objects of that type are represented. In subsequent sections we shall describe the structure of data reference blocks for a number of the above data types, and describe the effect of certain operations in terms of how they transform the descriptors and data reference blocks of the data objects which constitute their arguments.

The notion of a variable is fundamental to any programming language and should be rigorously defined. The term variable is used in SNOBOL4 to denote any object to which a value may be assigned. Since all values in SNOBOL4 are uniformly represented by descriptors, representation of a variable must contain a descriptor field in which the descriptor specifying the value of the variable may be stored.

The distinction between values and variables is of special importance in SNOBOL4 since strings and certain other data objects may be

both variables and data objects which are the values of other variables. All SNOBOL4 variables are data objects and may be assigned as values of other variables. However, a data object is a variable only if it possesses a value attribute (descriptor field.)

3. Simple Assignment Statements

Simple assignment statements have the form V = E where V is a variable, and E is an expression whose value and type is to be assigned to the named string. Execution of the assignment statement V = E is accomplished by evaluation of V , followed by evaluation of E , followed by assignment of the descriptor which characterizes E as the value of a string specified by V .

X = 3.5 This assignment statement creates a descriptor for the number 3.5 and assigns this descriptor as the value of the string X . The descriptor for 3.5 contains a pointer to the constant 3.5 in the value field and a code which specifies the type REAL in the type field.

X = 'ABC' This assignment statement creates a descriptor whose value field points to the string ABC and whose type field specifies the type STRING , and assigns this descriptor as the value of X .

In both of the above examples the value field of the assigned descriptor will point to the data reference block of a data object. Reference blocks for data objects of type REAL will be called number reference blocks while reference blocks for objects of type STRING will be called string reference blocks. The structure and mode of accessing for string reference blocks is further discussed in the next section.

When the right hand side of a simple assignment statement is a single variable, statement execution merely involves copying of a descriptor.

Y = X This assignment statement takes the descriptor which specifies the value of X and assigns the descriptor to Y . After execution of this statement Y and X will contain identical descriptors as their values.

The value field of a descriptor contains a literal rather than a pointer to the value when the object is of type integer.

```
X = 5
Y = X
```

The first statement creates a descriptor with the literal 5 in the value field and the type specification INTEGER in the type field, and assigns this value to X. The second statement assigns this descriptor to Y.

Note that in this example two copies of the literal value are created. If the value of X had been the string ABC or the number 3.5, execution of Y = X would have merely created an additional pointer to the value rather than a further copy of the value itself.

The value field of a descriptor is a literal when it contains the null string:

```
X =
Y = X
```

The first statement creates a descriptor for the null string and assigns this descriptor to X. The second statement assigns this descriptor to Y. This descriptor for the null string is of type STRING but contains the literal 0 in its value field rather than a pointer to the null string.

Integers and the null string are the only objects that are represented by literals in the value field of the associated descriptor. All other objects are represented by pointers to the data reference blocks which constitute their values.*

4. String Representation and String Accessing

In the present section the structure and mode of accessing of string reference blocks will be considered. SNOBOL4 associates a unique string reference block with every created BCD string, although certain fields of a string reference block such as the descriptor field may be pushed down on function evaluation and subsequently restored (see section 12). Every string reference block contains a block heading field, a descriptor field, an attribute pointer field, a hash address field, and a BCD name field, as illustrated in figure 2.

---

*Data objects of type NAME are also represented by a literal in the value field of the descriptor (see section 7). However, in this case the data object is itself a pointer to another data object.

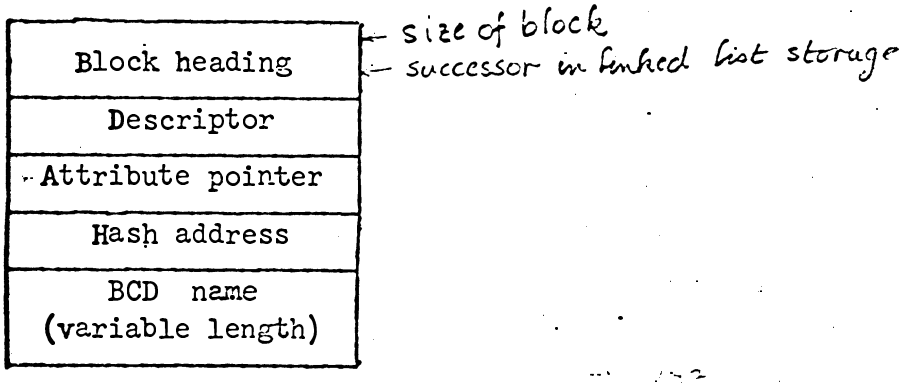


Figure 2. Format of a String Reference Block

The block heading field specifies the size of the string reference block and a pointer to its successor in linked list storage; the descriptor field specifies the value of the string; the attribute pointer field points to an attribute list which specifies whether the string is also a function name or a label; the hash address field is used to speed up accessing; and the BCD name field contains the literal string name.

During execution a given string may be accessed either through a descriptor which points to the string reference block or by explicit occurrence of the BCD name. When a string-valued variable is encountered during execution, the string reference block which constitutes the value is accessed through its descriptor. When a BCD string is formed during program execution, the literal BCD name is used for access purposes. If the BCD name of the newly created string corresponds to that of an already existing string the created string is identified with the existing string. Otherwise a new string reference block is created having the new string name in its BCD name field.

String reference blocks are stored in hash storage* in 256 linked lists referred to as buckets. The hash storage structure is determined by a 24 bit hash code which is computed from the BCD string name. 8 bits of the hash code are used to determine a bucket, and the remaining 16 bits

* The basic concepts of hashed storage are discussed on page 113 of (3).

are used to order entries within a given bucket.

When searching hash storage for a given string name the 24 bit hash code is computed, a bucket is selected using 8 bits of the hash code, and the 16 bit hash address field of successive string reference blocks in the bucket are compared against the 16 bits of the computed hash code until the bucket hash code exceeds the computed hash code or until a match is found. In the former case the BCD string is not in hash storage and is inserted into the bucket at the current point of the linked list. If matching of 16 bit hash codes occurs then BCD strings are matched. If the BCD string matches the BCD name field in the bucket, then the new BCD string is associated with this string reference block. Otherwise searching in the bucket continues through successive entries having the same 16 bit hash code and terminates in failure if an entry with a greater hash code is encountered. In practice matching of the 16 bit hash code almost always results in matching of BCD strings. The 16 bit hash code greatly speeds up the searching process both because ordering of entries reduces the average search time by a factor of 2 and because matching of individual bucket entries on their 16 bit hash codes is faster than matching on BCD strings.

5. Evaluation of Simple Expressions

The right hand side of an assignment statement consists of an expression containing operators and operands. Operators are characterized

by their degree which specifies the number of arguments of the operator, by a domain in which arguments take their values and by a range which specifies the set of values which result from application of an operator to its operands. The domain and range of an operator can usually be specified in terms of the primitive data types permitted in the language. Each primitive data type has an associated set of operators which can be used to manipulate objects of that data type.

The binary arithmetic operators + - * / and ** are available for operations on data objects of type INTEGER and of type REAL . These operations are subject to the usual precedence rules. Parentheses are permitted for overriding precedence in the usual manner. Binary arithmetic operators must be separated from their arguments by at least one blank.

NOTE

`X = 3 + 4 * 5`	Execution of this statement results in assignment to X of a descriptor whose value field contains the literal 23 .
`X = (3.5 + 1.4) * 2.0`	Execution of this statement results in assignment to X of a descriptor whose value field points to the constant 9.8 .

Unary minus operators are permitted and are written immediately preceding their arguments.

`X = 12 + 2 * -3` (No space!) `Y = X * -2`	The first statement assigns 6 as the value of X while the second statement assigns -12 as the value of Y .

Two strings can be formed into a single string by means of the concatenation operator. Concatenation of two strings is specified merely by juxtaposition of the literal strings or of the corresponding string names, and there is no explicit symbol in SNOBOL4 for string concatenation. However concatenation is nevertheless a binary operation which expects both its arguments to be of the type STRING , and which produces a value of the type STRING .*

* It will be seen in section 10 that one or both of the arguments of the concatenation operator may be of the type PATTERN , and that the value in this case will be of the type PATTERN . However when both arguments are of the type STRING , the value of applying the concatenation operator to its arguments will be the concatenated string.

```
X = 'A' 'B'
```

This assignment statement creates a string reference block for the literal string AB and assigns a descriptor for this string reference block to X .

```
X = 'AB'
Y = 'CD'
Z = X Y
P = X X
```

The first two statements create string reference blocks for the two strings AB and CD and assign the respective descriptors to X and Y . The third and fourth statements create string reference blocks for the strings ABCD and ABAB and assign the respective descriptors to Z and P .

Literals of type INTEGER are automatically converted to literals of type STRING when they occur in a context where the argument is expected to be of the type string.

```
X = 'A' 5
Y = X 36 + 3
```

The first statement converts the integer 5 to the type STRING and assigns a descriptor for the string A5 to X . The second statement performs an integer addition, converts the result to the type STRING , and assigns a descriptor for the string A539 to Y .

When an expression contains both arithmetic operators and the concatenation operator then evaluation of arithmetic operators takes precedence over concatenation operators as illustrated above. However constituents to be concatenated must be separated by a space.

```
X36 = 'A' 5
  Y = X36 + 3
```

X36 is here treated as a single string constituent. Execution of the second statement will result in error termination since a string cannot be added to an integer.

The null string descriptor and the descriptor for the integer zero have the same representation in the value field but differ in the type field. If the null string is encountered during evaluation of an integer expression it is converted to the type INTEGER .

```
X =
X = X + 1
```

Execution of the first statement assigns the null string descriptor to X . The second statement converts this descriptor to the integer 0 , performs the addition and assigns a descriptor for the integer 1 to X .

If the integer 0 is encountered in a context where an operand of the type STRING is expected it is converted, not to the null string, but to the string constant 0 .

What happens if X is non-zero integer. Automatic conversion INTEGER → STRING?

```
X = 0
X = 'A' X
```

The first statement assigns a descriptor for the integer 0 to X . The second statement creates the string A0 and assigns a descriptor to X which points to the newly created string.

Non-significant zeros are omitted in a quantity of the type integer.

```
X = 3 + 01
Y = 'A' '01'
Z = 'A' 01
```

X is assigned the value 4 . Y is assigned the value A01 since evaluation of '01' results in the string 01 . Z is assigned the value A1 since 01 is first evaluated as an integer and results in the constant 1 on being converted to the type string.

SNOBOL4 does not permit mixed arithmetic by automatic conversion of quantities of the type INTEGER to the type REAL , or concatenation of strings and REAL quantities by automatic conversion of REAL quantities to the type STRING . However such conversion can be accomplished by the two-argument system function CONVERT which converts the quantity specified by the first argument to the type specified by the second argument.

```
X = 1.5
Y = X + 1
```

Execution of the second statement will result in error termination since X is of type REAL and 1 is of type INTEGER.

```
X = 1.5
Y = X + CONVERT(1,'REAL')
```

Execution of the second statement will assign to Y a descriptor pointing to the REAL number 2.5 .

```
X = 1.5
Y = 'A' X
```

Execution of the second statement will result in error termination, since the concatenation operator cannot handle arguments of type REAL .

```
X = 1.5
Y = 'A' CONVERT(X,'STRING')
```

Execution of the second statement creates the four character string A1.5 and assigns to Y a descriptor pointing to this string.

6. Arrays

SNOBOL4 permits the specification of arrays in which the type of each element is assigned along with its value. The types of the elements in a given array may therefore be specified independently, and any given array element may change its type whenever a new value is assigned to it.

A given variable may be assigned the type ARRAY by execution of an assignment statement.

```
VECTOR = ARRAY(3, 'ABC')
```

This assignment statement assigns a three-element array as the value of VECTOR. Each of the array elements is initialized to the string ABC .

```
VECTOR = ARRAY(3)
```

This assignment statement assigns a three element array as the value of VECTOR, and initializes each element to the null string.

An array is created by execution of a system function called ARRAY whose first argument specifies the array dimensions. The second argument is optional. If it is present then all elements of the array are initialized to the value specified by the second argument. If it is absent then all elements are initialized to the null string.

Note that an array created by the ARRAY function does not initially have a name. A name is associated with an array by execution of an assignment statement which assigns a descriptor for the array to the

descriptor field specified by its left hand side. If the left hand side is a string name this name becomes the name of the array.

Execution of the ARRAY function causes creation of an array reference block which contains a descriptor for each element of the created array together with a dope vector which specifies how descriptors corresponding to individual array elements are to be accessed. The data structures created during execution of an ARRAY function are illustrated by the following example.

VECTOR = ARRAY(3, 'ABC') Execution of this statement results in the creation of an array reference block with three descriptors and a dope vector for accessing these descriptors. Each of the descriptors is initialized to be of the type STRING. If there is already a string reference block for the string ABC the value field of each descriptor will point to this string reference block. Otherwise the string reference block will be created and each descriptor will point to the created string reference block. Finally a descriptor of type ARRAY pointing to the new array reference block is created and assigned to the variable VECTOR.

Arrays with multiple dimensions can be specified by a first argument consisting of a list of integer valued expressions.

V = ARRAY('3,2', 'ABC') Execution of this statement creates a two dimensional 3 by 2 array reference block and assigns to V a descriptor of the type ARRAY which points to the newly created array reference block. All six descriptors of the array reference block are initialized to point to the string reference block for ABC.

W = ARRAY('K,L,M') Execution of this statement assigns a three dimensional K by L by M array to W and initializes each element to zero. The values of K, L, and M must be specified prior to execution of this statement, and must be of the type INTEGER.

Access to array elements is specified by the array name followed by the array indices in triangular parentheses.

```
VECTOR< 1 > = 5
VECTOR< 2 > = 3.5
     V< 1,2 > = 'X'
VECTOR< 1 > = V< 1,2 >
```

The first two statements assign the integer 5 and the real number 3.5 to the first two elements of the array VECTOR . The third statement assigns the string constant X to the (1,2) element of V . The final statement reassigns both the value and type of the first element of VECTOR, replacing the integer constant 5 by the string constant X .

Lower and upper bounds can be specified for arrays by the use of a colon.

```
V = ARRAY('-5:5')
```

Execution of this statement creates an array reference block with eleven descriptors and assigns to V a descriptor of the type ARRAY which points to the created array reference block. The first, sixth and eleventh descriptors of the array reference block are respectively accessed by V<-5> , V< 0 > and V< 5 > .

If an array index is outside the bounds specified by the array declaration, then there is no descriptor corresponding to the accessed array elements, and accessing of the array element is said to fail.

```
     A = ARRAY(3)
A< 5 > = 2
     X = A< -5 >
```

Both the second and third statement are said to fail since the array A created by the first statement contains descriptors only for the array elements A< 1 > , A< 2 > and A< 3 > .

Failure because an array element exceeds its bounds does not cause error termination. It merely causes execution of the given statement to be terminated, and returns a failure indication for the statement which can be used in the GO TO part for conditional branching (see section 9).

The value of a variable of the type ARRAY may be assigned to another variable.

```
X = ARRAY(3)
Y = X
```

The first statement assigns to X a descriptor pointing to the array reference block of a newly created array. The second statement assigns the descriptor field of X to Y so that both variables point to the single copy of the array.

Causes equivalence of X and Y
Single copy of data object

```
X< 2 > = 5
Y< 2 > = Y< 2 > + 1
```

Since X< 2 > and Y< 2 > refer to the same object, the second statement assigns the value 6 to both X< 2 > and Y< 2 > .

A copy of the array reference block of a given array can be created by the function COPY .

```
X = ARRAY(3)
Y = COPY(X)
```

The second statement creates a copy of the array reference block associated with X and assigns to Y a descriptor pointing to the newly created array reference block.

```
X< 2 > = 5
Y< 2 > = Y< 2 > + 1
```

In this case X and Y point to different three-element arrays. The first statement assigns the value 5 to the second element of X's copy while the second statement increments the value of Y's copy by 1 . Since all elements of Y are initialized to the null string, and the null string may be automatically converted to the integer 0 , the second statement assigns the value 1 to Y< 2 > .

An array reference block is essentially anonymous, and a copied array reference block must be assigned to a variable just like a newly created array before it can be subsequently referenced.

It should be noted that the function COPY will result in an error if the descriptor which constitutes its argument is of the type STRING . String reference blocks cannot by their nature be copied. since the BCD name field of the string reference block is used for the purpose of assigning storage and accessing of the string reference block. If there were two copies of the string reference block, there would be no means for the hashing mechanism to distinguish between the two copies when the given string was referenced during program execution.*

The anonymity of array reference blocks allows multiple copies to be created and given different names. The nonanonymity of string reference blocks precludes creation of multiple copies, although a facility for pushdown of the descriptor field during fuction calls is available (see section 12).

---

*The COPY instruction copies only the array reference block and not the objects pointed to by the array reference block. The descriptors of the copied array reference block initially point to the same objects as the original, but may have different values assigned to them during execution.

The anonymity of array reference blocks creates problems in contexts where names of array elements are treated as values. The use of the name operator for obtaining names of anonymous array elements is discussed in the next section.

SNOBOL4 has a system function for determining the dimensions associated with a given array name. The function PROTOTYPE(A) delivers as its value the string which constituted the value of the first parameter of the array declaration for A .

```
A = ARRAY('3:10,15','ABC')
Y = PROTOTYPE(A)
```

The first statement creates a two dimensional array with 120(8 x 15) descriptors all initialized to ABC . The second statement creates a string reference block for the seven character string "3:10,15" and assigns to Y a descriptor which points to this string.

The system function PROTOTYPE is an example of a function which selects certain attributes of the data object which constitutes its argument. In SNOBOL4 such attribute selection functions produce the BCD string representation of the attribute as their value. The principal other attribute selection function is the function DATATYPE(X) which selects the type attribute of its argument.

```
Y = DATATYPE(X)
```

This statement assigns to Y a descriptor of type STRING which points to the string representation of the data type of X . Thus if X is of the type REAL then Y would point to the string reference block for REAL , while if X is of the type ARRAY then Y would point to the string reference block for ARRAY.

If the argument of the function DATATYPE is a programmer defined data type (see section 13) then the value returned will be the string name of the data type specified in the data structure declaration.

7. Indirect Addresses and Composite Names

The unary operator $ denotes indirect addressing. It must be placed immediately preceding its argument without an intervening space.

```
 Y = 4
 X = 'Y'
 Z = $X
$X = 5
```

The third statement assigns to Z the value of the string specified by Y, i.e., Z will be assigned the value 4. The fourth statement assigns the value 5 to Y.

In order to understand indirect addressing the process of statement evaluation will be analyzed. The evaluation process differs for variables occurring on the left and right hand side of an assignment statement.

```
X = X
```

Evaluation of the instance of X on the left hand side of this statement results in a pointer to the descriptor field of X. This pointer is referred to as the left hand value of X. Evaluation of the instance of X on the right hand side produces the contents D(X) of the descriptor field of X. This descriptor D(X) is referred to as the right hand value of X.

When $X occurs in an assignment statement, then the argument X is first evaluated yielding a pointer to the descriptor field of X (its left hand value). The effect of the $ operator is to take the corresponding right hand value (descriptor), test the type of the descriptor and, if the descriptor is of type STRING, deliver as its value a pointer to the descriptor field of the string reference block. If $X is on the left hand side of an assignment statement, this becomes the new left hand value while if $X is on the right hand side of an assignment statement the descriptor to which $X points becomes the right hand value.

```
X = 'Y'
Z = $X
```

In the second statement $X produces a pointer to the descriptor field of Y. Since $X appears on the right hand side of the assignment statement, the corresponding right hand value is assigned to Z, i.e. the content of the descriptor field of Y is placed in the descriptor field of Z.

```
X = 'Y'
$X = 5
```

The expression $X in the second statement produces a pointer to the descriptor field of Y . Execution of the statement assigns 5 to this descriptor field.

```
A< 2 > = 'X'
$A< 2 > = 'Y'
```

In the second statement the written argument A< 2 > of $ yields a pointer to a descriptor field as its value. The operator $ applied to this descriptor produces as its value a pointer to the descriptor field of the string reference block X . The value Y is then assigned to the descriptor field of X , using the normal rules for assignment statements.

Evaluation of a quoted string yields a pointer to the string reference block as its value. Since a string reference block is not a descriptor field, a quoted string cannot appear on the left hand side of an assignment statement.

```
'X' = 5
```

This statement is illegal since the value of the left hand side is the string reference block of X , and values can be assigned only to descriptor fields and not to string reference blocks.

A quoted string cannot appear as the argument of an indirect addressing operator for the same reason.

```
$'X' = 5
  Y  = $'X'
```

Both instances of $'X' are illegal since the value of 'X' is a string reference block, and the operator $ requires a descriptor as its argument.

It is convenient in a language to manipulate left hand values (pointers to descriptors) as data objects and to assign left hand values as values of variables. In SNOBOL4 there is a operator called the name operator which produces as its value the left hand value (name) of an object rather than the object itself. The name operator is denoted by a period preceding the argument.

```
X = .Y
```

This assignment statement assigns to X the name of Y rather than the value of Y .

The term "name" is a vague one, and there are several meanings of this notion that could be adopted. For example the notion "name" could be thought of as a pointer to the string reference block, so that X = .Y would be equivalent to X = 'Y' .

The technical notion of name adopted in SNOBOL4 corresponds to the notion of a left hand value as previously defined, i.e. the name of an object is a pointer to its descriptor field. The statement X = .Y assigns to X the left hand value (pointer to the descriptor field) of Y . When an assignment statement such as X = .Y is executed an object of type NAME is created. This data object can be stored either as a literal in the descriptor field or in a reference block for objects of type NAME. If names of names are to be permitted then name reference blocks are required and we shall therefore assume that data objects of type NAME are stored in name reference blocks rather than as literals in descriptor fields.

Execution of .Y creates a name reference block and initializes this reference block with a pointer that points to the descriptor field of Y . The statement X = .Y causes a name reference block for the descriptor field of Y to be created, and assigns to X a descriptor of type NAME pointing to the newly created name reference block.

The difference between execution of the two statements X = 'Y' and X = .Y is illustrated in figure 3 .

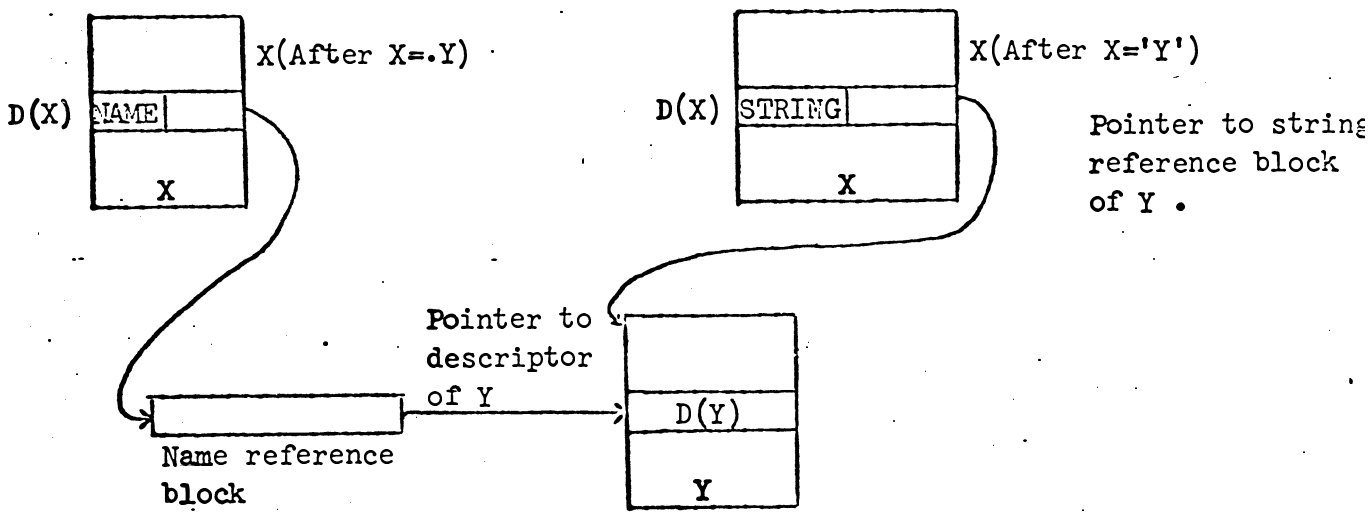


Figure 3. Information structures after execution of X = .Y (left) and after execution of X = 'Y' (right).

Thus the name operator produces as its value a pointer to a name of (pointer to) a descriptor field whereas the quotes operator produces as its value a pointer to a string reference block. This distinction could be glossed over for data objects of type STRING for which there is a one to one correspondence between descriptor fields and data reference blocks. However array elements and certain other data objects in SNOBOL4 have associated descriptor fields but no unique data reference blocks. In this case quotes are inappropriate since there is no unique data reference block to be named. However the naming operator, may be used to name the associated descriptor field in precisely the same manner that it was used for strings above.

```
X = 'A< 2 >'
```
The right hand side of this assignment statement denotes the four character string A< 2 > .
Execution of this statement would assign to X a pointer to the string reference block of the string A< 2 > .

```
X = .A< 2 >
```
This statement creates a name reference block for the array element A< 2 > and assigns to X a pointer to this name reference block.

When the argument of an indirect addressing operator is of type NAME, then the value produced by the indirect addressing operator is the pointer in the value field of the name reference block.

```
 X = .A< 2 >
$X = 5
```
The first statement sets the descriptor field of X to point to the name reference block of A< 2 > .
The second statement assigns the value 5 to the descriptor field of A< 2 > .

```
$.A< 2 > = 5
```
This statement has the same effect as the statement A< 2 > = 5 .

The sequence of two operations $. is an identity operation. The operation $ is a left inverse of the naming operator.

An object of type NAME cannot appear on the left hand side of an assignment statement.

Why not X = .A<2>
X = .A<3>

```
.X = 5
```

This statement is illegal. The value of .X is a pointer to the created name reference block. Name reference blocks contain only a pointer to the named object and contain no information fields in which descriptors can be stored.

```
 X = 'Y'
.$X = 5
```

The second statement is illegal for the same reason as in the previous example. It is equivalent to the illegal statement .Y = 5 .

```
X = 'Y'
Z = .$X
```

In this example the second statement is legal. $X produces as its value a pointer to the descriptor field of Y , and .$X produces a name reference block pointing to the value of Y . Z is assigned a value of type NAME which points to this name reference block.

Note that .$ is not the identity even in the second case, i.e., Z = X assigns to Z a pointer to the string reference block of Y while Z = .$X assigns to Z a pointer to the name reference block for Y .

```
X = .Y
Z = .$X
```

In this case the second statement has precisely the same effect as Z = X .

When the descriptor field of X is of the type NAME the .$ applied to X is an identity operation.

When a pointer to a name reference block occurs as the value of an expression it can be used only as the argument of an indirect addressing operator or as the second argument of an assignment operator.

```
X = .Y
Z = X + 1
Z = .Y 'A'
```

Both the second and third statements are illegal, since a right hand value which is a name reference block is not a legal argument of arithmetic or concatenation operators.

8. Input and Output

Standard input and output facilities are available through use of the three initialized names INPUT , OUTPUT and PUNCH .*

---

*The names INPUT , OUTPUT and PUNCH are initialized by the system to have the effect described below, but may be redefined by the user.

Use of the name INPUT in an expression causes input of the next input record from the standard input medium.

CARD = INPUT — This statement reads an input record from the input medium, creates a string reference block for this input record, and assigns to CARD a descriptor which points to the string reference block.

DOUBLE = INPUT INPUT — This statement reads two input records, creates a string reference block for the concatenated input records, and assigns to DOUBLE a descriptor pointing to the string reference block.

Output can be accomplished by the assignment of a string to the reserved name OUTPUT .

OUTPUT = 'XY' — This statement causes output of an output record containing the two characters XY .

OUTPUT = C[illegible] — This statement causes output of the string pointed to by the descriptor field of CARD .

A punched output record may be obtained by assignment of a string valued variable to the reserved name PUNCH .

PUNCH = CARD — This statement causes punching of the string pointed to by the descriptor field of CARD .

It is sometimes convenient to think of an input or output medium as being a data stream which can be accessed in a sequential manner. The above input-output facilities may then be thought of as an association of names with standard input and output streams in such a manner that use of the name implies access to the data stream. Clearly

input streams should be accessed in a read mode while output streams should be accessed in a write mode. Thus names such as INPUT should be used in a context where access is in the read mode while names such as OUTPUT and PUNCH should be used in a context where access is in the write mode.

Use of a variable which denotes an input stream in the write mode does not make sense and will lead to error termination.

```
INPUT = 5
```

Execution of this statement would result in error termination, since we cannot assign values to an input stream.

However, use of a variable which denotes an output stream in the read mode is permitted, and produces the value most recently assigned to that output stream.

```
OUTPUT = 5
OUTPUT = OUTPUT + 1
```

The occurrence of OUTPUT on the right hand side in the second statement produces the value most recently assigned to output, i.e., 5. The second statement therefore results in output of the integer 6.

The programmer may associate his own names with input and output streams by means of READ and PRINT declarations. A READ declaration specifies an identifier name, the input stream to be associated with that identifier, and the length of the string to be read each time the identifier is invoked.

```
READ('INPUT',5,80)
```

This READ declaration specifies that the name INPUT is to be associated with data stream 5 (tape 5) and that the length of the input string is to be 80 characters. This specification is in fact the one to which the identifier INPUT is implicitly initialized by the system.*

---

*If the length L of an input record is greater than the length N specified by the READ declaration for the input identifier, then the final L-N characters of the input record will be lost.

Blanks occurring at the end of an input record can be removed by the system function TRIM .

`X = TRIM(INPUT)` Execution of this statement creates a string reference block whose BCD name field consists of the string of input characters up to and including the last nonblank character, and assigns to X a pointer to this string reference block.

The function TRIM can be applied to any object of the type STRING, but is applied most frequently to strings read from the input medium.

A PRINT declaration specifies an identifier, an output stream, and a format specification which is used to determine how characters of the output string are mapped onto the output medium. The format specification is similar in style to that used in FORTRAN. In particular the specification nX where n is an integer specifies n spaces and the specification nA1 where n is an integer specifies an n-character sequence.

`PRINT('OUTPUT',6,'(1X,131A1)')` This print declaration associates the identifier OUTPUT with data stream 6 (tape 6), and specifies a format of one space followed by 131 characters. This specification is the one to which the identifier OUTPUT is implicitly initialized by the system.

`PRINT('PUNCH',7,'(80A1)')` This print declaration associates the identifier PUNCH with data stream 7 (tape 7) and specifies a format of 80 characters. This specification is the one to which the identifier PUNCH is implicitly initialized by the system.

Page ejects and vertical spacing on the printer are controlled by the character in the first column. Access to the first column of the standard output medium (data stream 6) can be obtained by introducing the nonstandard output identifier CONTROL as follows:

```
PRINT('CONTROL',6,'(132A1)')
```

This print declaration associates the identifier CONTROL with data stream 6 and specifies a format of 132 characters.

```
CONTROL = 1
```

This statement results in output of a line image with a 1 in the first column which will result in a page eject on being printed.

The format for a title can be specified as follows:

```
PRINT('TITLE',6,'(1H1,131A1/(1X,131A1))')
```

This print declaration associates the identifier TITLE with the standard output stream. This first line has a 1 in column 1 (specified by 1H1) followed by 131 characters of message. The format specification following the slash specifies an arbitrary number of lines of 131 characters preceded by a blank. When printing a given title only the number of lines needed will be used.

```
TITLE = 'SNOBOL4 TEST PROGRAM'
```

This assignment statement will cause the title 'SNOBOL4 TEST PROGRAM' to be printed in columns 2 through 21 of line 1. Since the total number of characters in the title is less than 132, only a single line will be printed.

Only character strings in an A1 format can be printed in the output stream. An attempt to output a data object which is not a string results in printout of the data type.

```
OUTPUT = 3.5
```

Since 3.5 is of the type REAL it cannot be printed on the output stream. Execution of this statement would result in printout of the word REAL in columns 2 through 5.

```
OUTPUT = CONVERT(3.5,'STRING')
```

This statement would result in conversion of the number into a string and printout of the three character string 3.5.

SNOBOL4 contains a number of commands for backing up over data streams.

REWIND(N) — This statement causes backup to the beginning of data stream N .

BACKSPACE(N) — This statement backspaces one record on data stream N .

ENDFILE(N) — This statement writes an end of file on data stream N .

9. Labels and Conditional Branching

Statements in SNOBOL4 are normally executed in sequence. However, statements may be labelled, and transfer to a labelled statement by unconditional or conditional branching is permitted. Branching after execution of a statement is specified in the GO TO field, and is separated from the statement by a colon.

```
L X = 5 : (L1)
```

This statement has a label L , and a GO TO field which specifies unconditional transfer to the label L1 when execution of this statement has been completed.

Labels are strings and have a string reference block which defines them just as other strings. However the program point to which control is to be transferred on branching to the label is specified not in the descriptor field of the string reference block but in the attribute list pointed to by the attribute pointer. This allows assignment of values to strings used as labels in precisely the same manner as assignment of values to strings not used as labels.

```
L1 = 5 : (L1)
```

This statement specifies assignment of the value 5 to L1 followed by transfer to the statement labelled by the string name L1 . There is no conflict between the two uses of L1 , since the value is assigned to the descriptor field of the string L1 , while the label to which control is transferred is specified by the label attribute of the attribute list.

Unconditional branching to a label is specified by enclosing the label in the GO TO field in parentheses. Conditional branching in SNOBOL4 is specified in a nonstandard manner. Conditional branching statements of the form "if X < 0 then go to L1 else go to L2" are not permitted in SNOBOL4 . Instead of branching on the basis of an explicit logical condition, SNOBOL4 automatically associates the predicate success or failure with a certain class of statements. Such statements automatically return one of the two values success or failure when their execution has been completed. This information may in turn be used in the go to field for conditional branching purposes.

Statement : S(L1) F(L2) When execution of this statement is completed, branching to L1 occurs if the value "success" is returned and branching to L2 occurs if the value "failure" is returned.

The system functions S and F can be used only in the GO TO field, and respectively denote branching on success and branching on failure to the label which constitutes their arguments.

The assignment statements considered in the previous section normally return success, or an error return if there is a discrepancy of type during expression evaluation or assignment.*

The principal class of statements which return success or failure on their completion is the pattern matching assignment statement.. Pattern matching assignment statements are discussed in the next section.

10. Simple Pattern Matching

SNOBOL4 has facilities for the construction of patterns, for the assignment of patterns as the values of variables, and for pattern

*However exceeding the bounds of an array returns failure in the above sense and permits conditonal branching in the GO TO field. Failure during execution of certain system functions and programmer defined functions (see section 12) also permit conditional branching.

matching to determine whether a given string contains a substring which is an instance of a given pattern.

Pattern matching in SNOBOL4 is accomplished by a special kind of assignment statement referred to as a pattern matching assignment statement. The left hand side of a pattern matching assignment statement consists of a reference string S followed by a pattern specification P . The right hand side of a pattern matching assignment statement consists of an expression of the type STRING , whose value is used to replace the matched substring of S if pattern matching succeeds.

```
S P = X
```

This pattern matching assignment statement has a reference string S , a pattern specification part P and a right hand side X . Its execution results in an attempt to find a substring of S which is an instance of the pattern specified by P . If the pattern match succeeds, then the matched substring is replaced by the string specified by X . If the pattern match fails then execution of the statement is said to fail, and no replacement takes place.

The simplest example of a pattern is a string. Pattern matching when the pattern is a string is illustrated by the following three statements:

```
S = 'ABCDE'
P = 'BCD'
S P = 'X'
```

The third statement attempts to find an instance of the string BCD in the string S . The pattern P wil match the 2nd through 4th characters of the string S . Replacement of the matched string by X will result in assignment to S of the string AXE .

It is important to specify the order in which instances of P are matched against substrings of the reference string since different orders of matching may in some cases lead to matching of different substrings. The pattern matching procedure first attempts to find an instance of the pattern P starting at position 1 of the reference string. If it fails to find an instance of P at position 1 , it attempts to

find a pattern match at successive positions k = 2,3,... until it either finds a match or finds that there are insufficient characters in S to match the minimum length instance of the pattern P .

In the above example the pattern matching procedure first attempts to match P at position 1 of the string S and fails. It then attempts to match the pattern at position 2 of the string S and succeeds.

```
  S = 'ABCDE'
  P = 'XYZ'
S P = 'X'
```

In this case the pattern matching procedure would try to match the pattern XYZ at positions 1 , 2 and 3 of the string S and fail on each occasion. It would then recognize that the number of remaining characters starting at position 4 of the string S are insufficient to match P and would register failure without attempting to match P at positions 4 and 5 . Because of failure, the value of the string S would remain unchanged.

```
  S = 'ABBC'
  P = 'B'
S P = 'X'
```

Execution of the third statement would result in matching of position 2 of the string S , and assignment to S of the string AXBC . Note that P could also have matched position 3 of the string S , but that this match will not be attempted.

Patterns which consist of a finite number of alternatives can be specified by explicitly listing the alternatives. The vertical stroke (referred to as the alternation operator) is used instead of the comma to separate the alternatives.

```
'X' | 'Y' | 'Z'
```

This expression specifies a pattern consisting of the three alternative strings X or Y or Z .

A pattern is an example of a data object just like a string or an array. It is represented by an information structure referred to as a pattern reference block. SNOBOL4 permits patterns to be assigned as values of variables. When a pattern is assigned as the value of a variable

the type field of the assigned descriptor is of the type PATTERN and the value field of the descriptor points to the pattern reference block.

```
P = 'X' | 'Y' | 'Z'
```

This assignment statement creates a pattern reference block for the pattern 'X' | 'Y' | 'Z' and assigns to P a descriptor of type PATTERN which points to the pattern reference block.

```
  S = 'YXZ'
  P = 'X' | 'Y' | 'Z'
S P = 'A'
```

The third statement will result in a match of P against the first character of S , and assignment to S of the value AXZ .

When a pattern P consists of a sequence of alternatives, the order of matching at a given string position is the left to right order in which the alternatives are listed. However all instances of a pattern P are matched at a given string position k before any attempt is made to match the pattern P at the string position k + 1 .

```
  S = 'XYZ'
  P = 'XY' | 'X'
S P = 'A'
```

The pattern matching procedure will match the first two characters XY of the string S and the value AZ will be assigned to S . If the order of listing of alternatives had been reversed, the pattern matching procedure would have matched the pattern X against the first character of S , and the value AYZ would have been assigned to S .

```
  S = 'XXYZ'
  P = 'XY' | 'X'
S P = 'A'
```

In this case the pattern matching procedure will fail to match XY at position 1 and will match the second alternative X at position 1 . The value AXYZ will be assigned to S . The substring X at position 1 is matched before the substring XY at position 2 , since all instances of P are matched at position 1 before a match of any instance of P at position 2 is attempted.

The first successful pattern match automatically terminates the pattern matching procedure, and returns an indicator of success, together with a specification of the matched substring so that substitution can be performed. Failure of all attempts to match causes substitution to be

abandoned and returns a failure indicator.

A pattern matching assignment statement may be thought of as a predicate or conditional expression which returns one of the two values success or failure. The success or failure indicator may be used in the GO TO part of the statement for the purpose of conditional branching.

S P = X : S(L1)F(L2) — This statement transfers to L1 on successful pattern matching and to L2 on failure to match.

If either one or both of the above branching specifications are omitted the next statement in sequence is taken for the omitted conditions. A pattern matching statement with no GO TO part will always be followed by the next statement in sequence, and have as its principal effect the substitution for a resulting pattern match.

A blank right hand side results in substitution of the null string for the matched string. However omission of the = sign as well as the right hand side will result in omission of substitution even when the pattern match is successful.

S P : S(L1)F(L2) — Execution of this statement will leave the string S unaltered even when the pattern match is successful. Its principal effect is that of a conditional branching instruction where the condition is success or failure of the pattern match.

S P — Execution of this statement will cause pattern matching followed by transfer to the next instruction in sequence. At first glance it appears that this statement will have a null effect. However SNOBOL 4 permits assignment during pattern matching (see section 17) so that a given pattern matching statement may in general result in an arbitrarily large number of assignments of values even when it does not explicitly specify replacement of the match substring.

The expression S P above represents the core of any pattern matching statement. It specifies the fundamental operation of testing

whether any one of the finite set of substrings of S belongs to the set P . This is a fundamental question in set theory, automata theory and other formal theories related to computation. It is clear that the range of such an operation is truth or falsity (success or failure). Adding on the assignment part of the statement merely adds a simple transformation onto the end of the pattern matching process if it succeeds. The presence of an assignment operation should not obscure the fact that the basic portion of the statement is the pattern matching predicate.

There are two pattern matching constituents (FAIL and ABORT) which explicitly signal failure when they are encountered in a pattern. The constituent FAIL denotes a pattern which always fails to match. However, FAIL does not automatically cause failure of the complete pattern match but merely requests the pattern matching algorithm to seek an alternative. One of the principal uses of FAIL is to initialize a pattern which is built up during execution by the accumulation of a sequence of alternatives.

fixed field (col 1)

```
    P = FAIL
L   P = P | TRIM(INPUT) : S(L)
```

P is initialized to FAIL prior to accumulating a sequence of alternatives for P in much the same way that a variable used for summation is initialized to zero.

The pattern constituent FAIL may be thought of as the identity under alternation, since the pattern FAIL | X is equivalent to the pattern X for all patterns X . We may also think of FAIL as a pattern which has a null set of instances. Since alternation is essentially the operation of set union, it is natural that the identity element under alternation is the null set.

The pattern constituent ABORT causes failure of the complete pattern match when it is encountered during pattern matching. ABORT would not normally occur as the first constituent of an alternative. However when it occurs as the last constituent of an alternative, it specifies

that none of the patterns which match the earlier constituents are to be included in the set of strings determined by this pattern specification.

P = '*' ABORT | Q The pattern P denotes all strings of the form Q other than those beginning with a * .

The pattern ABORT essentially allows a form of set difference to be introduced into pattern matching. i.e. If B is the set consisting of all patterns with an initial part of the form X , then the ordered set A - B can be specified as "X ABORT | A". However the sets B which can be "subtracted" by the use of the function ABORT are restricted to those which can be characterized as having a common initial part.

The patterns FAIL and ABORT above are examples of primitive pattern functions defined by the programming system. Additional primitive pattern functions will be defined in sections 15 and 16 .

11. The Order of Pattern Matching

Patterns are constructed from a set of primitive initial patterns by a set of primitive pattern building operations. The primitive initial patterns include the set of all strings over the SNOBOL4 character set, the primitive pattern functions FAIL and ABORT above, and primitive pattern functions introduced in later sections. The primitive pattern building operations include alternation and concatenation.

VOWEL = 'A' | 'E' | 'I' | 'O' | 'U' Execution of this statement creates a representation for the pattern " A or E or I or O or U ," referred to as a pattern reference block, and assigns to the variable vowel a descriptor of the type PATTERN pointing to the given pattern reference block.

Patterns created by alternation can be concatenated with strings or with other patterns to build more complex patterns.

VC = VOWEL ',' — Execution of this statement assigns to VC a pattern which consists of a vowel followed by a comma.

PVT = 'P' VOWEL 'T' — Execution of this statement assigns to PVT the pattern consisting of the five alternatives PAT , PET , PIT , POT , PUT .

Concatenation of two pattern valued variables specifies the pattern which consists of any instance of the first pattern followed by any instance of the second pattern.

VPAIR = VOWEL VOWEL — Execution of this assignment statement creates a pattern reference block for a pattern which matches any one of the 25 sequences of two vowels in succession, and assigns to VPAIR a descriptor of the type PATTERN which points to the newly created pattern reference block.

Note that both the alternation and concatenation operator are binary operators whose arguments may be either of type string or of type pattern. The alternation operator always converts string arguments to the type pattern and creates a new pattern which is the ordered union of its two arguments. The value of the concatenation operator is of type STRING if both its arguments are of type STRING and of type PATTERN if one or both of its arguments are of type PATTERN . When one or both arguments of the concatenation operator are of the type pattern, the pattern which results from its application is any instance of the pattern determined by its first argument followed by any instance of the pattern determined by its second argument.

Patterns which, when considered by themselves, contain redundant alternatives, may become nonredundant when concatenated with other patterns.

```
  X = 'AB' | 'A' | 'ABC'
S X = 'Y'
```

When the pattern X is used in matching a given string S , successive string positions will be tested first for an occurrence of AB , then for an occurrence of A and then for an occurrence of ABC .

For example if S = 'ABCD' then AB will be matched at position 1 , while if S = 'BACD' then A will be matched at position 2 . The second alternative will match whenever the first occurrence of A is not followed by a B . The third alternative will never match since any occurrence of ABC in a string will result in a match of AB by the first alternative.

Although the alternative 'ABC' is redundant in the above example, it becomes nonredundant when X is used as a constituent of a composite pattern.

```
X = 'AB' | 'A' | 'ABC'
Y = 'D' | 'E'
Z = X Y
S = 'ABCEF'
S Z = 'Q'
```

The fifth statement specifies matching of S for an instance of the pattern X followed by an instance of the pattern Y . This results in a match of the first four characters of S and assignment to S of the value QF .

The order of pattern matching for the above pattern Z at a given string position will now be described. The pattern matching procedure first attempts to match an instance of X at the given string position, in the order specified by the pattern for X . If matching for X succeeds then an attempt is made to find an instance of the constituent Y immediately following the matched constituent. If an instance of Y is found the pattern match is successful. If no instance of Y can be found following the matched instance of X then backup occurs and an attempt to match the next untried alternative of X is attempted. If all alternatives of X are exhausted without a successful pattern match then the pattern match is said to fail for that string position and pattern matching at the next string position is attempted.

On this particular instance X will first match the pattern AB at string position 1 . Y will then fail because AB is not followed by a D or E . The next alternative of X will then be tried and will match A . Y will again fail becuase A is not followed by

a D or E . The final alternative of X will then be tried and will match ABC . The second alternative of Y will then succeed in matching E , resulting in a successful match by Z of ABCE .

When a pattern P consists of K components P1,P2,...,PK the pattern matching procedure tries to match all instances of P for successive string positions POS = 1,2,...,LASTPOS, where LASTPOS is the last position of the string S at which sufficient characters remain to match the minimum length instance of P .

At each string position an attempt is made for successive constituents P1,P2...etc using the internal ordering for each P_j determined by the rules of pattern construction. When a given instance of P_j is successfully matched then if $j = K$ the whole pattern has been successfully matched. If $j \neq K$ the order index and length of the matched P_j is pushed down in case backtracking is required, and matching of P_{j+1} is attempted starting at the point following the matched component. If matching of a component P_j fails, then if $j = 1$ the complete pattern match has failed and a pattern match is attempted at the next string position. Otherwise backtracking to the previous component P_{j-1} is tried until it either succeeds or exhausts all of its alternatives and causes further backtracking.

A flow diagram for the above pattern matching procedure is as follows:

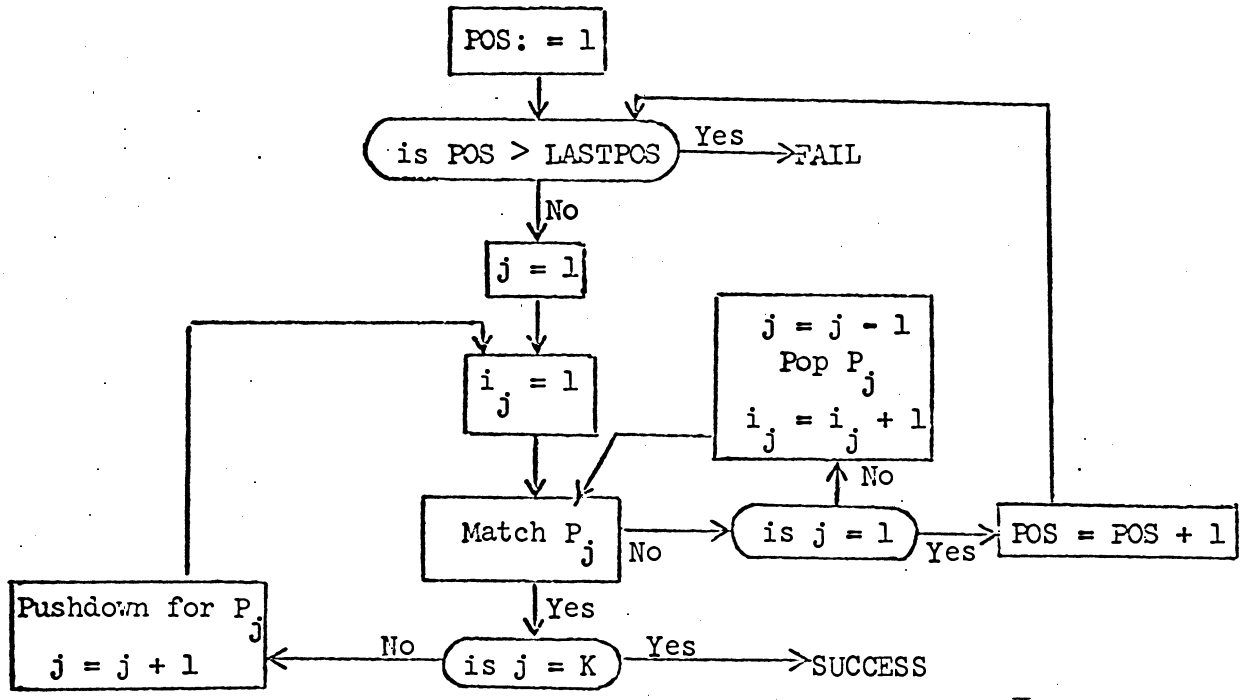


Figure 4. The order of matching a sequence of concatenated components $P_1, P_2, \cdots, P_K$.

The "Match P_j" box shown in the above flow diagram can be thought of as a subroutine which requires the string position, order index i_j and pattern reference block of P_j as parameters, sequences through successive alternatives and terminates either when all alternatives are exhausted or when it has ascertained that all remaining patterns are longer than the number of characters in the string being matched. The subroutine for matching P_j is clearly recursive since P_j may in general consist of alternatives which themselves consist of sequences of concatenated constituents. The order in which nested pattern matching is performed is further discussed in section 19.

12. Programmer-Defined Functions

SNOBOL4 subprogram definition facilities serve the same purpose as procedure definition facilities in procedure-oriented languages,

but are organized in a different way. Programmer defined functions are introduced by the DEFINE declaration, which may have either one or two parameters.* The first parameter specifies the function name, the formal parameters in parentheses, and the set of local variables. The second parameter, which is optional, specifies the label of the entry point. If the second parameter is absent, the entry label is the same as the name of the function.

```
DEFINE('F(X,Y)N,M','ENTRY')
DEFINE('SIZE(X)')
```

The first declaration defines a function F with two parameter X , Y , two local variables N , M , and an entry point ENTRY . The second declaration defines a function called SIZE with one parameter X , no local variables, and an entry point SIZE .

A function call is specified by the function name followed by actual parameters enclosed in parentheses.

```
F(P,Q)
```

If a two parameter function F has been previously defined by a DEFINE statement, this function call will call the function F with actual parameters P , Q . If an entry point was specified in the DEFINE statement control will be transferred to the statement having the entry point name as its label. Otherwise control will be transferred to the statement having the function name as its label.

Every function expects its argument to be of a certain type and returns a value of a certain type. This applies both to functions defined by the programmer and to system functions.

```
SIZE(S)
```

There is a one parameter system function SIZE which expects its argument to be of the type STRING and produces a value of type INTEGER specifying the number of characters in its string argument.

---

*The define declaration is implemented by the system function DEFINE. A function definition has the form of a function call to the system function DEFINE. A function definition is here called a declaration because it associates a function attribute with a variable in much the same way that a procedure declaration associates a procedure attribute with variables of a procedure oriented language.

Function calls may occur in expressions in any context where the value delivered by the function is appropriate.

```
S = 'ABC'
X = SIZE(S) + 1
```

The second statement evaluates the function SIZE(S) producing the value 3 and then assigns the value 4 to X .

When a function is called, the current values of the function name, formal parameters and local variables are pushed down.* The descriptor fields of the function name and local variables are initialized to the null string, and the descriptor fields of formal parameters are initialized to evaluated actual parameters.

```
X = F(P,Q)
```

If the function F has been defined by the declarations DEFINE('F(X,Y)N,M') , then the function call F(P,Q) causes pushdown of the descriptor fields of F , X , Y , N , M . The descriptor fields of F , N and M are initialized to the null string. The parameters P and Q are evaluated, and their values are assigned to X and Y . The function F is then entered and its value is assigned to X .

Function definition by the programmer will be illustrated by the following definition of the function SIZE .**

```
    DEFINE('SIZE(X)')
SIZE X LEN(1) =          :F(RETURN)
       SIZE = SIZE + 1 :(SIZE)
```

Since no entry point is specified in the DEFINE statement, the entry point for this function is at the statement having the same label as the function name. The second line of the above definition is a pattern matching statement which fails if X is a null string and branches to the label RETURN which causes control to be returned to the calling program. If X is non null, its first

---

* i.e. The descriptor fields of the string reference blocks of the function name, formal parameter strings and local variable strings are pushed down.

** The function SIZE is already defined by the system. However any system function may be redefined by the programmer by executing a DEFINE statement for the function. Function definitions in SNOBOL4 are global so that redefinition of a function destroys the previous definition.

character is matched by LEN(1)* and replaced by the null string. The value of the variable SIZE , which is initially null, is then increased by 1 and control is returned unconditionally to the second line. If X is initially a string of length N , the second line will fail at the N+1th execution, with SIZE having the value N , and control will be returned to the calling program.

On exit from a function control is normally transferred to the label RETURN . This label may be thought of as an entry point to a system function which moves the function value to an anonymous register, restores the values of all descriptors pushed down by the function call, and then returns control to the point of call with a pointer to the function value.

The set of all information items which are pushed down on function call will be referred to as an activation record. The activation record contains pushed down descriptor fields, a return pointer to the point of call, and other logical information. Since return from called functions is always in a last-in-first out order, activation records may be stored in a stack whose entries are activation records.

Expression evaluation may also require a stack. It is convenient to combine the stack for expression evaluation with the stack for function evaluation as described in (2) for ALGOL, and to arrange for the value delivered by a given function to be left at the top of the stack rather than in an anonymous register.

The pattern activation records described in section 21 may also be stored in the same stack, resulting in a single stacking mechanism for expressions, function calls and patterns.

---

*The function LEN(N) will match any string of N characters, and is an example of a primitive pattern function defined by the SNOBOL4 system. It is defined in section 15 .

Return to the point of call can be accomplished either by transfer to the label RETURN as described above, or by a transfer to the labels FRETURN or NRETURN . FRETURN restores the descriptor fields pushed down on function call, and returns a null value together with a failure indicator which can be used for conditional branching in the GO TO field. NRETURN indicates that the value returned is of the type NAME , and is used when the function occurs on the left hand side of an assignment statement and specifies the name of a descriptor to which a value is to be assigned. The difference between RETURN and NRETURN is illustrated by the following examples.

```
  DEFINE('F')
F F = A< 2 >  :(RETURN)
```

This parameterless function makes a copy of the descriptor A< 2 > and returns a pointer to this copy. It can be called on the right hand side of an assignment statement.

```
A< 2 > = 'Y'
     X = F( ) 'Z'
```

A call of a parameterless function must be followed by parentheses. The function call will deliver a pointer to a descriptor which points to Y . Further evaluation creates a string reference block for YZ , and assigns to X a pointer to this string reference block.

```
  DEFINE('F')
F F = .A< 2 > :(RETURN)
```

In this case F returns a pointer to a descriptor of type NAME which points to a name reference block which names A < 2 > .

```
   X = F( )
   X = $F( )
$F( )= 'Y'
```

Each of these three statements is legal. The first assigns to X a descriptor of type NAME which points to the name reference block of A < 2 > . The second assigns to X the descriptor A< 2 > , and the third assigns Y as the value of A< 2 > . The statement F( ) = X would be illegal since values cannot be assigned to a name reference block.'

```
  DEFINE('F')
F F = .A< 2 >  :(NRETURN)
```

In this case F returns a pointer to the descriptor A< 2 > , i.e. it returns the name of the descriptor A< 2 > .

```
F( ) = 'Y'
```

This statement stores the value Y in A< 2 > . The NRETURN operator allows the function call to appear on the left hand side of an assignment statement, and returns a true left hand value which may be used in assigning a value to the specified descriptor.

```
Y = F( )
```

This statement assigns the descriptor A< 2 > to Y , since occurrence of the left hand value results in further evaluation.

Note that the first and third function definitions above are equivalent if the function call occurs on the right hand side of an assignment statement, since they both name the same right hand value. However the left hand value of the first function call is a pointer to an anonymous descriptor to which no value can be assigned, while the left hand value of the third function call is a pointer to the original copy of the descriptor. The NRETURN facility allows the name of a specific descriptor to be returned while the RETURN facility preserves the value but destroys the identity of the name.

In the above definition of the function SIZE , the string SIZE was used for three different purposes:

1. As the name of a local variable to which values are assigned.
2. As a function name.
3. As a label and an entry point.

The three uses of the string SIZE correspond to three different attributes of the string reference block. The value attribute is stored in the descriptor field of the string reference block while the function and label attributes are specified in the attribute list associated with the string reference block.

The value attribute of a string which is also a function name is pushed down and initialized to null on function call. The function and label attributes cannot be pushed down and are therefore global, as is the BCD name. Since the value of the function attribute is not affected by a call of the function, recursive calls are permitted without execution of an inner DEFINE statement.

The use of recursive subroutines in SNOBOL4 will be illustrated by an application to the "tower of Hanoi" problem. In this problem it is assumed that we have a pole (pole A) on which a set of discs of successively decreasing size is stacked with the smallest at the top. It is assumed that there are two additional poles (pole B and pole C) which are initially empty. The objective of the game is to move the set of discs from pole A to pole C using pole B for intermediate storage such that only one disc is moved at a time, and a larger disc is never allowed to lie upon a smaller disc.

The problem with N discs can be defined recursively in terms of the problem with N - 1 discs as follows.

Move N - 1 discs from pole A to pole B
Move one disc from pole A to pole C
Move N - 1 discs from pole B to pole C

This recursive formulation of the problem leads to the following recursive SNOBOL4 function definition.

```
      DEFINE('HANOI(N,A,C,B)')
HANOI EQ(N,0)                                  :S(RETURN)
      HANOI(N-1,A,B,C)
      OUTPUT = 'MOVE DISC' N 'FROM' A 'TO' C
      HANOI(N-1,B,C,A)                         :(RETURN)
```

The subroutine call "HANOI(N,A,C,B)" can be interpreted as a call for moving N discs from pole A to pole C with B as the intermediate pole. When N = 0 then no action need be performed. When N is nonzero the problem is specified in terms of moving N-1 discs from A to B with C as the intermediate pole, execution of the output statement for moving the Nth disc from A to C , and then moving N discs from B to C with A as the intermediate pole. Note that the only tangible result of a given call of HANOI is the output generated by the OUTPUT statement. Any given call of HANOI will result in execution of output statements of each of the nested calls in precisely the order

in which the discs are to be moved.

The sequence of instructions actually executed for N = 2 is as follows:

```
    HANOI(2,A,C,B)
[   EQ(2,0)
    HANOI(1,A,B,C)
[   EQ(1,0)
    HANOI(0,A,C,B)
[   EQ(0,0) : S(RETURN)
    OUTPUT = 'MOVE DISC' 1 'FROM' A 'TO' B
    HANOI(0,C,B,A)
[   EQ(0,0) : S(RETURN) : RETURN
    OUTPUT = 'MOVE DISC' 2 'FROM' A 'TO' C
    HANOI(1,B,C,A)
[   EQ(1,0)
    HANOI(0,B,A,C)
[   EQ(0,0) : S(RETURN)
    OUTPUT = 'MOVE DISC' 1 'FROM' B 'TO' C
    HANOI(0,A,C,B)
[   EQ(0,0) : S(RETURN) : RETURN : RETURN
```

The square braces on the left indicate the instructions associated with each of the levels. The set of instructions generated by HANOI(3,A,C,B) consists of HANOI(2,A,B,C) followed by an output for disc 3 from A to C , followed by HANOI(2,B,C,A) . Each of the calls of HANOI with a parameter 2 will generate the same instruction sequence as that above (apart from differences in parameter names) including three output statements for discs 1 and 2 . The call to HANOI with parameter 3 will generate 7 output statements. By induction it follows that a call to HANOI with parameter N will result in $2^N - 1$ output statements, i.e. the tower of Hanoi problem with N discs requires $2^N - 1$ discs to be moved altogether in order to solve it.

Each resursive call of HANOI pushes down the five descriptor fields of the strings HANOI,N,A,C,B, initializes the parameters A,C,B to some permutation of A,B,C and initializes N to the appropriate integer value. The value field of HANOI is not used by this function

and remains null at each recursive level. However the function attribute of HANOI is not pushed down, and each recursive call examines the same physical copy of the attribute list to determine the function attribute of HANOI .

13. Programmer defined data types

SNOBOL4 permits the programmer to define new data types by means of the system function DATA .

DATA('TYPE(FIELD1,FIELD2,...,FIELDK)') This command specifies creation of a new data type of the type specified in the TYPE field, having K components whose names are specified in the same fields.

DATA('ELEMENT(LEFT,RIGHT)') This definition creates a new data type called ELEMENT , having two subfields LEFT and RIGHT .

Instances of a defined data type can be created by use of the data name. Individual fields of the newly created instance can be initialized. In the absence of initialization newly created instances are initialized by default to the null string.

L = ELEMENT('X','Y') This statement creates an instance of the data type ELEMENT , assigns a descriptor to L pointing to the newly created structure, and initializes the two fields of the newly created structure to 'X' and 'Y' .

L = ELEMENT('X')
L = ELEMENT( ) These two statements each create an instance of the structure and assign a descriptor to L which points to the structure. The first initializes the first field to 'X' and the second field to the null string, while the second statement initializes both fields to the null string.

The descriptor which is assigned to L is of type ELEMENT . If the type field of descriptors is sufficiently large (say 10 bits or

more) a distinct type code can be assigned to each programmer-defined data type and the assumption can be made that the sum of the number of primitive data types and programmer-defined data types will never exceed 2^n where n is the number of bits in the type field. Alternatively codes for programmer-defined data types could be represented by a single code in the type field denoting "programmer defined", and the type code itself could be given in the attribute list.

Execution of a DATA command results in the creation of a special kind of programmer-defined function having the name specified by the type field. When called this function creates an instance of the data structure. Execution of the DATA command also creates additional programmer defined selector functions whose names correspond to the field names specified in the data command. These selector functions take a single argument which must be of the given data type, and return as their value, the value of the field named by the selector function.

```
X = LEFT(L)
```

If the argument L is of the type ELEMENT then the function LEFT automatically selects the descriptor in the LEFT field and assigns this descriptor as the value of X .

The field names of two different data types may be the same.

```
DATA('MESH(LEFT,RIGHT,UP,DOWN)')
```

This declaration of the data type MESH has two of its field names (LEFT and RIGHT) the same as those of the data type ELEMENT .

When a given field name is used as a selector, then a test must be made to determine the type of the object that is pointed to. The selector function used is determined by the result of this data type test.

```
X = LEFT(L)
```

If L is of the type ELEMENT then the selection rule for data objects of the type ELEMENT will be used, while if L is of the type MESH then the selection rule for data objects with the type MESH will be used.

Selectors can be implemented by associating a function attribute with the string reference block of the selector. The function attribute specifies execution of a one parameter system function which first tests the type of its parameter and then uses the selector mechanism for that data type to perfom selection.

Programmer defined data structures may be thought of as one dimensional arrays whose components are explicitly named rather than being referred to by indices. This similarity is reflected in the representation of programmer defined data structures.

Creation of an instance of a programmer defined data structure results in the creation of a data reference block which characterizes both the form of the data structure and the value associated with each field of the data structure. The value associated with each field of the data structure is represented by a descriptor, and the form of the data structure is represented by a block heading which is analogous to an array dope vector in that it contains the information required to select components of the data structure.

`L = ELEMENT('X','Y')` This statement creates a data reference block with two descriptors of type STRING which respectively point to the strings X and Y . The block heading specifies how the selector functions LEFT and RIGHT are used in accessing the descriptors in the data reference block.

There are great similarities between array reference blocks and data reference blocks. Both kinds of reference blocks contain a descriptor which specifies the value of each component, and a block heading which specifies how components are to be accessed. The principal difference is that array components are accessed by indexing, while components of programmer defined data types are accessed by explicitly naming them.

The indirect naming problem described for arrays in section 7 , arises in precisely the same form for names of components of programmer-defined data structures.

```
Z = 'LEFT(X)'
$Z = 5
```

The first statement assigns to Z the quoted seven character string LEFT(X) , and creates a string reference block for this string. The second statement assigns the value 5 to this newly created string.

The name of the component denoted by LEFT(X) can be obtained by the naming operator . which was used in similar circumstances to obtain array names.

```
Z = .LEFT(X)
$Z = 5
```

The first statement assigns to Z a descriptor of the type NAME which points to the LEFT component of the defined data type X . The second statement assigns the value 5 to the LEFT component of X .

The type of a programmer defined data structure may be determined by use of the function DATATYPE .

```
X = DATATYPE(L)
```

If L is of the type ELEMENT , then this assignment statement will assign to X a descriptor of the type STRING which points to the string reference block for the string whose BCD name is ELEMENT .

14. Compilation During Execution

A SNOBOL4 source program is normally compiled into an intermediate language prior to execution. During execution this intermediate language is executed interpretively by a SNOBOL4 interpreter. Compilation includes the creation of string reference blocks for all string names which explicitly occur in the program. Occurrences of string names in the program are replaced by pointers to the created string reference blocks.

SNOBOL4 can handle code segments of the intermediate language as data objects, and assign such data objects as the values of variables. Code segments of the intermediate language are said to be of the data type 'CODE' . Source language segments can be converted into data objects

of the type CODE by use of the CONVERT function.

CONVERT(' X = 1;','CODE') The CONVERT function converts the source language statement X=1 into the corresponding intermediate language code segment. Use of the CONVERT function to accomplish compilation during execution emphasizes the fact that compilation is essentially a process of converting a static program from one initial representation to another. Conversion during execution is sometimes referred to as incremental compilation.

CODE = CONVERT(' X = 1;','CODE') This statement creates a code segment for X=1 and assigns to CODE a descriptor of type CODE which points to the created code segment.

Note that in the above example the string name CODE is used both as a type name and as a string name to which a value is assigned. Use of CODE as a type name causes reference to the attribute list of CODE .

In order to execute a created code segment a mechanism is required for transferring to its first instruction. When a code segment has been assigned as the value of a string, it can be accessed through the descriptor field of the string. A mechanism is available in SNOBOL4 for accessing the object pointed to by the descriptor field in the execute mode. Access in the execute mode is accomplished by enclosing the name of the data object in triangular parentheses.

Z = CONVERT(' X = 1 : (L);','CODE') : < Z > This statement converts to code the source language statement X=1 : (L); and transfers control to the code pointed to by the descriptor field of the identifier enclosed in triangular parentheses. The GO TO field of the code segment specifies unconditional transfer to the label L when execution is completed. The label L would normally be a label in the source program.

Execute mode accessing of Z succeeds only if the descriptor field of Z is of the type CODE , and results in a tranfer of control to the code segment. The code segment must contain within itself a provision for transfer of control to a source program or to some other created segment

on its completion.

A variable Z which occurs enclosed in triangular parentheses in a GO TO field has many of the properties of a label. A different notation is used for specifying transfer of control so that the processor can recognize that access is through the value field of the descriptor rather than through the label attribute of the attribute list.

Transfer to a created code segment can be accomplished without the use of triangular parentheses by labelling one or more of the statements of the code segment.

```
S = 'L X = 1;M Y = 2 : (Q);'
C = CONVERT(S,'CODE')
S P = Y : S(L)F(M)
```

The first statement assigns to S a source language representation of two statements, the first having label L and the second having label M. The second statement converts the source language string S to code. The third statement transfers to the label L in the created code on success and to label M in the created code on failure. When execution of the code has been completed, control is transferred to Q, which is assumed to be a label of the source program or of a created code segment.

The above example indicates that triangular parentheses are redundant. They are made available in SNOBOL4 to avoid explicit specification of the entry point when the entry point is at the beginning of the segment. Triangular parentheses have been discussed here at some length because they illuminate the accessing structure of string reference blocks.

When the CONVERT statement converts a source language string it calls the compiler and converts the string to code in precisely the same manner that it would have done during translation prior to execution. Source language names are not destroyed by compilation, since the string reference blocks to which they correspond are accessible by the hashing

mechanism. The CONVERT function creates any new string names of the program string it is creating, and translates both old and new string names to pointers to the corresponding string reference blocks.

All string names in SNOBOL4 are universally known, and a given string name inside an incrementally compiled program segment refers to precisely the same string reference block as an occurrence of that string name in the static source program, a created occurrence of the string name during execution, or an occurrence of the string name in some other incrementally compiled program.

The string reference block remains unique even on function call, although a function call pushes down the descriptor field of the string reference block. Every descriptor field in SNOBOL4 may be thought of as representing the top of a stack which may be pushed on function call and popped on return. Access to a descriptor field in the read or write mode is always interpreted as the currently top stack element. Access to any other attribute of a string is always interpreted as access to the unique copy.

Occurrence of a label in a label field results in association of the label attribute with the point in the program at which the label occurs. The association of a label and a program point is established at compile time. If, during an incremental compilation, a labelled statement is compiled whose label corresponds to a previously existing label, then the previous definition of the label is erased and the new definition of the label is substituted.

The use of compilation during execution will be illustrated by defining a one argument function CALL which takes the expression which constitutes its argument and evaluates it.

```
     DEFINE('CALL(FORM)S')
CALL S = ' CALL =' FORM ' : S(RETURN)F(FRETURN);'
     S = CONVERT(S,'CODE') : < S >
```

The first statement specifies a function definition for CALL(FORM) with local

variable S . The second statement assigns to S a string which constitutes a source language statement for evaluating FORM , assigning the value to CALL and returning. The third statement converts to code the statement assigned to S , and then executes it.

```
Q = CALL('F(X,Y)')
```

This function call of the function CALL creates the statement " CALL=F(X,Y) : S(RETURN)F(FRETURN converts this statement to code. executes it and either returns the value of F(X,Y) if F(X,Y) succeeds or fails if F(X,Y) fails.

15. Primitive Pattern Functions

The patterns considered in previous sections were built up from SNOBOL4 character strings by alternation and concatenation. SNOBOL4 also contains a number of primitive pattern functions that allow the programmer to directly specify certain useful classes of complex patterns.

The primitive pattern functions include the function ARB which specifies an arbitrary sequence of characters, including the null string.

```
ITEM = ARB '*'
```

This statement creates a pattern reference block for an arbitrary number of characters followed by a star, and assigns a descriptor for this pattern reference block to ITEM .

ARB will always match the shortest possible string that satisfies the later constituents of the pattern.

```
  P = ARB '*'
  S = 'X**'
S P = 'A'
```

The third statement will cause ARB to match X since this is the shortest string that is followed by a star.

A pattern consisting only of the constituent ARB will always match the null string since this is the shortest string matching the pattern.

```
  S = 'XY'
  P = ARB
S P = 'A'
```

This causes P to match the null string prior to the beginning of the string S , and assigns to S the string AXY .

ARB specifies an ordered class of strings just like patterns constructed by the user. However the ordered class is determined implicitly by the system rather than explicitly by the user. ARB matches successively longer strings until the number of characters beyond the string matched by ARB is insufficient to match the minimum number of characters of later pattern constituents.

```
  P = ARB '*'
  S = 'XYZ'
S P = 'A'
```

Execution of the third statement would result in three attempted matches at position 1 with ARB respectively matching the null string, X and XY and failing each time because none of these substrings is followed by a * . Matching of ARB with XYZ would not be attempted because insufficient characters would remain in S to match * . Two matches (null and Y) would be attempted at position 2 , one match (null) would be attempted at position 3 , and the pattern matching statement would then fail.

Note that ARB specifies an infinite class of patterns. Any algorithm for handling infinite classes of patterns clearly cannot afford to match all instances of an infinite class at a given position before attempting to match instances of the class at the next position. The present algorithm uses the fact that ARB matches patterns strictly in increasing order of length to terminate searches for members of this infinite class.

The number of patterns to be matched in order to establish failure to match a given string may easily become combinatorially explosive. The SNOBOL 4 pattern algorithm uses the length termination criterion very effectively in order to keep down this combinatorial growth.

The function BAL is another example of a primitive pattern function. BAL will match any nonnull substring with no parentheses

or with an equal number of opening and matching closing parentheses. At any given string position, BAL always tries to match the shortest possible nonnull string, and increases the number of characters to be matched by one until there are insufficient remaining characters to match the minimum number of characters required by the pattern which follows BAL .

```
      S = 'A*(B+C)'
    S BAL = 'X'
S BAL ',' = 'X'
```

The second statement matches A , and assigns X*(B+C) to S . The third statement fails since S does not contain an example of a balanced expression followed by a comma.

When executing the third statement above BAL will match a succession of substrings of S and each time fail because the substring is not followed by a comma. At position 1 BAL wil match A and A* . At position 2 BAL will match * . At position 3 BAL will match nothing at all. At position 4 BAL will match B , B+ and B+C . At position 5 BAL will match + and +C . At position 6 BAL will match C . Matching of the 7th position will not be attempted because not enough characters remain to match the comma..

In this example BAL will only attempt to match strings which do not include the last character. Although A*(B+C) , *(B+C) and (B+C) are valid instances of the pattern BAL , they would not be tried because the pattern matching procedure tests to ensure that the remaining unmatched characters of S are sufficient to match the one character string which follows BAL .

```
  S = 'A*(B+C)'
  P = '(' BAL ')'
S P = 'X'
```

The pattern P specified in the second statement will match any balanced string enclosed in parentheses. The third statement will result in a match of (B+C) and replacement of S by A*X .

Since the first component of the pattern P above is a literal, pattern matching will immediately fail at positions 1 and 2 of the

string S . At position 3 , pattern matching of the first component of P will succeed. BAL will then succeed in matching B and B+ and result in failure because these strings are not followed by a closing parenthesis. Finally BAL will match B+C and be followed by matching of the third component of the pattern P . A success indicator together with pointers to the matched statement will then be made available so that substitution and possibly conditional branching may be performed.

The functions ARB and BAL above are parameterless pattern functions denoting a fixed class of patterns. SNOBOL4 also contains a number of primitive pattern functions which define different classes of patterns depending on a parameter.

The function LEN is a one parameter pattern function which, when supplied with a parameter N of the type INTEGER will match any character string of length N .

```
CARDLENGTH = LEN(72)
```

Execution of this statement creates a pattern reference block for matching any 72-character pattern and assigns to CARDLENGTH a descriptor which points to this pattern reference block.

```
P = 'X' LEN(3) 'Y'
```

Execution of this statement creates a pattern which will match any instance of X followed by any 3 characters followed by Y .

```
  P = 'X' LEN(2) 'Y'
  S = 'XXYXYX'
S P = 'A'
```

The third statement will fail at string position 1 , and succeed at string position 2 in matching characters 2 through 5 . S will be assigned the value XAX .

```
  P = 'X' LEN(3) 'Y'
  S = 'XXXXXX'
S P = 'A'
```

The third statement will fail at string positions 1 and 2 , and then return failure of the complete pattern match since the four remaining characters starting at string position 3 are insufficient to match the 5-character pattern P .

Note that the pattern component LEN always results in an immediate pattern match and therefore takes very many fewer operations

on the average than ARB or BAL .

A further interesting one-parameter primitive pattern function is the function ARBNO(P) which requires its argument P to be of the type STRING or PATTERN , and specifies an arbitrary number of occurrences, including no occurrences of P .

```
X = ARBNO('A') 'B'
```
This statement specifies X to be a pattern consisting of an arbitrary number of A's followed by a B .

```
Y = ARBNO('A' | 'B') 'C'
```
This statement specifies Y to be a pattern consisting of an arbitrary number of A's or B's followed by a C .

Like ARB , ARBNO(P) will try to find a match at the leftmost possible position. For a given position it will first try to match the null string, then a simple instance of the pattern P , and then a sequence of k instances of the pattern P for k=2,3,... .

ARBNO(P) is in principle equivalent to the following infinite set of alternatives.

ARBNO(P) ≡ NULL | P | P P | P P P | ...

The above expression specifies the order in which alternatives will be matched at a given string position.

```
X = ARBNO('A' | 'B')
S = 'ABDBBC'
S X 'C' = 'Y'
```
The pattern X 'C' will match an arbitrary number of A's and B's followed by a C . X will first match the null string and result in failure since the first character of S is not a C . X will then successively match A and AB and result in failure because there is no following C . Then, since ABD is not an instance of X , matching of S at position 1 will fail. Matching of S at positions 2 and 3 will fail for similar reasons. Matching of S at position 4 will first fail twice with X matching the null string and B , and finally succeed with X matching BB . HBDY

Note that in matching ARBNO(P) , the order of matching for elements of P is determined by the pattern specification for P ,

using the previously specified ordering rules.

```
  P = 'AB' | 'A' | 'ABC'
  X = ARBNO(P) ('B' | 'C')
  S = 'ABC'
  T = 'ABABC'
S X = 'Y'
T X = 'Y'
```

Execution of the fifth statement would cause matching of ABC by X rather than of the shorter pattern AB , since the alternative AB occurs before the alternative A in P . Execution of the sixth statement will cause matching of the pattern AB rather than of the pattern ABABC , since all alternatives of P are explored before considering alternatives of P .

```
Y = ARBNO('A' | 'B') 'B'
```

This pattern will match the shortest string of A's and B's followed by a B . The shortest such string will always consist of a string of A's followed by a B .

The above pattern matches precisely the same set of characters as the pattern ARBNO('A') 'B' . However the classes of strings matched by these patterns are different when they appear as one of a sequence of concatenated constituents.

```
  Y = ARBNO('A' | 'B') 'B'
  P = Y 'C'
  S = 'ABABC'
S P = 'X'
```

Execution of the fourth statement will result in matching of the complete string. Note that if Y had been the pattern ARBNO('A') 'B' , then the fourth statement would have matched the string ABC .

The argument of ARBNO may be a pattern specification that includes primitive pattern functions.

```
ARBNO(LEN(5))
```

This pattern specification specifies the set of all character strings that are multiples of five characters.

```
ARBNO(ARB)
```

This pattern specification specifies an arbitrary number of occurrences of an arbitrary number of characters. This class of patterns is precisely the same as the class defined by ARB . Occurrence of this constituent in a pattern specification would however affect the order in which patterns were matched, and would in general result in a larger number of attempted matches during execution than ARB , particularly if the end result were failure.

```
ARBNO(LEN(1) ',')
```

This pattern specifies an arbitrary number of occurrences of strings of length 1 followed by a comma, such as "X,Y,Z," .

```
X = '(' (ARBNO(LEN(1) ',') LEN(1) | NULL) ')'
```

This statement assigns to X a pattern specification for a parenthesiezed list which contains either no elements or an arbitrary number of characters separated by commas.

The parameterless pattern function ARB is equivalent to the function ARBNO(P) for the fixed parameter P=LEN(1) . ARBNO(LEN(1)) will match patterns consisting of arbitrary character strings in precisely the same order as the pattern ARB .

The pattern ARBNO(P) occurring by itself will always match the null string regardless of the argument P .

```
  X = ARBNO(P)
S X = 'A'
```

The second statment will always succeed and will cause S to be augmented by an initial A independently of the pattern P .

The pattern functions available in SNOBOL4 include the function SPAN(CS) whose argument CS specifies a set of characters. SPAN(CS) matches the longest non-null run of characters of the set CS starting at the point of matching, and fails if the first character does not belong to CS .

```
SPAN('ABC')
```

This pattern specifies the longest (non-null) run of A's , B's and C's at the current point of matching, and fails if the first character is not A , B or C .

```
S = 'ABBC'
S SPAN('AB') = 'X'
```

The second statement will match ABB and assign the string XC to S .

The set of strings matched by SPAN(P) | NULL are identical to the set of strings matched by ARBNO(P) . However, the order of string matching is quite different.

```
  P = SPAN('A')
  Q = ARBNO('A')
  S = 'AAB'
S P = 'X'
S Q = 'X'
```

The fourth statement matches AA and assigns XB to S . The fifth statement matches the initial null string and assigns XAAB to S .

```
  P = SPAN('A') 'B'
  Q = ARBNO('A') 'B'
  S = 'AAB'
S P = 'X'
S Q = 'X'
```

In this case both the fourth and fifth statement will match the complete string S . However, SPAN('A') will immediately match the longest string of A's whereas ARBNO will match the null string and a single A before succeeding.

There are certain instances in which SPAN(P) will fail to match and ARBNO(P) will succeed.

```
  P = SPAN('AB') 'B'
  Q = ARBNO('A' | 'B') 'B'
  S = 'AABC'
S P = 'Y'
S Q = 'Y'
```

The fourth statement will fail since the longest string of A's and B's is not followed by a B . indeed, the pattern P of this example must always fail. The fifth statement will match AAB and assign YC to S .

A slightly more complex example of the difference between the SPAN and ARBNO functions is the following.

```
    P = SPAN('AB') ('B' | 'C')
    Q = ARBNO('A' | 'B') ('B' | 'C')
    S = AABABCD
S P 'A'= 'X'
S Q 'A'= 'X'
```

The fourth statement will match P against AABABC and will then fail since this string is not followed by an A . Similar failure would result in positions 2, 3, 4, 5 of the string S , and the fourth statement would then fail. The fifth statement would match AABA and assign to S the string XBCD .

Matching of a given longest constituent by SPAN never permits backtracking. Although in the above example the string AA is an instance of a pattern matched by SPAN which would result in success, this string will never be matched because backtracking is not permitted.

Backtracking at an arbitrary point during pattern matching can be eleiminated by means of the primitive function FENCE , which causes the

complete pattern match to fail whenever an alternative for it is requested in backup.

```
  X = 'AB' | 'ABC'
  P = X FENCE 'D'
  S = 'ABCD'
S P = 'Y'
```

The fourth statement will fail because X will match 'AB' and FENCE will prevent backtracking to match an alternative of the pattern X .

FENCE may be thought of as a primitive pattern which always matches the null string when it is encountered. It has the side effect of replacing the pattern components up to and including FENCE by the matched string, eliminating all alternatives other than continuations of the pattern in which FENCE occurs, and anchoring the pattern match so that only continuations of the partial pattern matched prior to the occurrence of FENCE are considered.

The primitive pattern function ANY has an argument which consists of a string of characters in quotes, and specifies the set of single character patterns given in the string argument.

```
VOWEL = ANY('AEIOU')
    X = 'P' VOWEL 'T'
```

The first statement specifies that VOWEL is a pattern consisting of any one of the five vowels. The second statement specifies the set of five patterns PAT, PET, PIT, POT, PUT .

Patterns defined by ANY can be defined by explicitly listing alternatives. However patterns specified by ANY are both represented more compactly and recognized more rapidly than if the corresponding patterns were specified by alternation.

The primitive pattern function NOTANY has the same form of argument as ANY , but specifies the set of single character patterns which is the complement of the set of characters specified by the argument.

```
NOTA = NOTANY('A')
```

This statement specifies that NOTA is a pattern which matches any character other than an A .

The primitive pattern function BREAK(P) has the same form of argument as the functions, SPAN , ANY and NOTANY and will match the longest string of characters not in the set specified by P (including the null string).

X = BREAK(',.') This statement specifies a pattern which consists of all characters starting at the present string position and terminatingat the next comma or period. If there is no comma or period beyond the string position being matched pattern matching fails.

16. Matching of String Positions

SNOBOL4 has a number of primitive pattern functions which match positions in a reference string rather than elements of the string itself. It is assumed in this section that an N-character string has N+1 explicit positon names 0,1,...,N , such that string position 0 precedes the first character of the string, and string position K follows the Kth character of the string.

The pattern function POS(N) matches the string position following the Nth character of the string.

X = 'A' POS(2) 'B' This pattern matches an A followed by a B only if A is in the second position and B is in the third position, i.e., POS(2) matches the fixed pattern which consists of a null string at position 2 of the reference string.

Y = POS(0) P This pattern will match the pattern P only if it can be matched starting at string position 1 . The occurrence of POS(N) as the first constituent of a pattern effectively anchors the pattern match so that only pattern matches of remaining constituents starting at character position N + 1 will be considered.

`Y = 'A' POS(0) 'P'` This pattern match can never succeed since the constituent POS(0) can only match the position prior to the first string character.

Since the function POS(N) matches a null pattern following the Nth string character, it may be used for inserting characters into a string at a given string position.

`S POS(2) = 'ABC'` This pattern matching statement will succeed for all strings S with two or more characters and will cause ABC to be inserted following the second position of the string S .

The pattern function RPOS(N) counts character positions backwards from the end of the string, and matches the string position prior to the Nth character, counting backwards.

`X = 'A' RPOS(0)` This pattern will match the last character of any string ending in an A .

`Y = RPOS(1) LEN(1)` This pattern will match the last character of any nonnull string.

`S RPOS(0) = 'ABC'` This pattern matching statement will always succeed and concatenate ABC onto the end of the string S .

```
S RPOS(1) LEN(1) = 'ABC'
      S RPOS(1) = 'ABC'
```

The first statement will replace the last character by ABC while the second statement will insert the string ABC prior to the last character.

`X = POS(0) P RPOS(0)` This pattern specifies that P must match the complete reference string, rather than a substring of the reference string. Since set recognition problems are usually formulated as problems of determining whether a string S belongs to a set P , rather than as problems of determining whether a substring of S belongs to P , this form of pattern specification is prevalent when dealing with theoretical recognition problems.

```
  P = POS(0) SPAN('0123456789') RPOS(0)
S P = 'INTEGER'
```

The second statement replaces the string S by the string INTEGER if and only if the complete string is a string of digits.

The pattern function TAB(N) will match the sequence of characters starting at the current string position up to and including the Nth string character.

```
X = POS(2) TAB(5)
```

This pattern will match the third, fourth and fifth character of the string being matched. It is equivalent to POS(2) LEN(3) . Both POS(2) TAB(5) and POS(2) LEN(3) will fail if the string being matched has less than 5 characters.

The pattern TAB(N) will fail whenever the string being matched has less than N characters, and whenever the current string position is beyond that specified by TAB(N) .

```
X = POS(N) TAB(N)
Y = TAB(N) POS(N)
```

The first pattern will always result in TAB(N) matching the null string. The second statement will cause TAB(N) to match the first N characters.

```
X = POS(N + 1) TAB(N)
Y = TAB(N + 1) POS(N)
```

Each of these statements will always fail since the first component positions the string pointer beyond the point at which the second component can succeed.

The function RTAB(N) specifies the position prior to the Nth string component counting backwards; i.e. it specifies the same string position as RPOS(N) . RTAB(N) matches all characters between the current string position and the position specified by RTAB(N) .

```
X = POS(0) RTAB(1)
```

The pattern RTAB(1) will match all characters up to but not including the last character of the reference string.

X = RPOS(3) RTAB(0)	The pattern RTAB(0) will match the last three characters of any string with at least three characters.

The parameterless pattern matching function REM will match the remaining characters of any string. It is equivalent to the function RTAB(0) .

X = RPOS(3) REM	The function REM will match the last three characters of the reference string.

17. Assignment During Pattern Matching

Pattern matching is a complex process involving the matching of a sequence of subpatterns and backup if a given instance of a subpattern is inconsistent with what follows. It is sometimes convenient to associate actions with subpatterns, to be performed when the subpattern is successfully matched. Because of the possibility of backup, it is necessary to associate "delayed" actions with subpatterns which are specified when the subpattern is successfully matched but which are performed only when the complete pattern has been successfully matched. However, it is convenient also to allow a second class of actions that are performed immediately on successfully matching a subpattern irrespective of whether backup occurs or of whether the pattern match is completed. Subpattern actions which are performed only when the complete pattern has been successfully matched are referred to as static actions. Subpattern actions which are performed immediately on matching the subpattern are referred to as dynamic actions.

SNOBOL4 permits only a restricted class of actions to be associated with subpatterns. It permits assignment of the instance of the matched subpattern as the value of a variable. Both static and dynamic assignment of subpatterns is permitted. Static assignment is delayed until the complete pattern has been matched and is not performed at all if the pattern match is unsuccessful. Dynamic assignment is performed immediately if a

given subpattern is matched, and is performed repeatedly if backup and successful rematching of that subpattern occur.

Static assignment is specified by the binary "." operator, whose first argument is of the type STRING or PATTERN and whose second argument is a variable.

P = P1 . V1 P2 . V2 ... PK . VK — If P1, P2,...,PK denote data objects of the type STRING or PATTERN, then the pattern P denotes the sequence of K subpatterns P1,P2,...,PK. When P is successfully matched, the matched instances of successive subpatterns will be assigned to the variables V1, V2,...,VK .

The static assignment operator has a higher precedence than concatenation so that P1 . V1 P2 . V2 automatically associates V1 with P1 and V2 with P2 .

Dynamic assignment is specified by the binary operator $ which like the "." operator, expects its first argument to be of the type STRING or PATTERN and expects its second argument to be a variable. The dynamic assignment operator has the same precedence as the static assignment operator.

P = P1 $ V1 P2 $ V2 ... PK $ VK — If P1, P2,...,PK denote data objects of the type STRING or PATTERN , then the pattern P denotes the sequence of K subpatterns P1,P2,...,PK . Whenever one of the K subpatterns is successfully matched its value is immediately assigned to the corresponding variable.

Examples of the use of static and dynamic pattern matching are given below.

```
  P = 'BC'
  S = 'ABC'
  X = P . V
S X = 'Y'
```

— The fourth statement will match the second and third characters of S , assign BC as the value of V and assign AY as the value of S .

```
    P = 'B' | 'C'
    S = 'ABC'
S P . V = 'Y'
```

The third statement will match the second character of S , assign B to V and assign AYC to S .

```
  P = '(' BAL . B ')'
  S = 'A*(B+C)'
S P = 'Y'
```

The third statement will match "(B+C)" , assign B+C as the value of B and assign A*Y to S .

```
  P ='B' | 'C'
  Q = P . V 'D'
  S = 'ABCA'
S Q = 'Y'
```

The pattern P matches both at the second position and at the third position, but the complete pattern Q fails on each occasion. Since the value of V is assigned only upon successfully matching of the complete pattern of which P . V is a constituent, V is not assigned a value by failure of this pattern matching statement.

Multiple naming can be accomplished by repeated use of the naming operator.

```
X = P . V . OUTPUT
```

This associates both the names V and OUTPUT with the pattern P , and will result in output of the matched substring if the pattern in which X occurs is successfully matched.

Different alternatives in a pattern may be associated with different names.

```
  VOWEL = ANY('AEIOU')
LETTER  = (VOWEL . V|LEN(1) . C) . L
```

The pattern LETTER matches precisely the same set of strings as LEN(1) . However when LETTER is used for pattern matching, then additional name assignments occur as side effects, i.e. if the matched constituent is a vowel it is assigned as the value of V , if it is not a vowel it is assigned as the value of C , and in any event it is assigned as the value of L .

Some examples of dynamic assignment during pattern matching will now be given.

```
  X = P $ V
S P = 'Y'
```

When a component P is the only constituent in a pattern, the P $ V has the same effect as P . V .

```
      P = 'A' | 'B'
      S = 'ABC'
S P $ V 'C' = 'Y'
```

The pattern P will first match A , resulting in immediate assignment of A to V , and then fail because the next character is not C . The pattern P will next succeed in matching B , resulting in assignment of B to V and then succeed. When pattern matching is completed V will have its most recently assigned value B as its value.

The sequence of successful components matched by a given pattern component during a pattern matching operation can be printed by associating the name OUTPUT with that component.

```
    P = ('A' | 'B') $ OUTPUT
    S = 'ABC'
S P 'C'= 'Y'
```

The pattern matching operation will give rise to two lines of output, the first containing the character A and the second containing the character B .

```
  P = BAL $ OUTPUT RPOS(0)
  S = 'A*(B+C)'
S P = 'Y'
```

The pattern BAL will match A , A* and P will fail because RPOS(0) does not match. BAL will finally match A*(B+C) and P will succeed. The third statement thus causes the three lines of printout A , A* and A*(B+C) .

A given pattern component may be assigned both a static and a dynamic name.

```
X = P . V $ W
```

The pattern component P has been assigned a static name V and a dynamic name W . Whenever a pattern match containing X is successful, the value assigned to V will be the same as the final value of W . If the pattern match is unsuccessful but the component P is successfully matched, then W will have been assigned a value, but V will remain unchanged. If W is replaced by the string OUTPUT , then the dynamic name associated with P will print out the sequence of successfully matched substrings, and V will be assigned a value only if the complete pattern match succeeds.

18. Deferred Pattern Evaluation

When an assignment to a pattern-valued variable is executed, then all elements on the right hand side are evaluated at the time that the assignment statement is executed.

```
  P = 'X' | 'YZ'
S P 'A' | P = 'B'
```

Execution of the second statement first creates an anonymous pattern identical to the pattern ('X' | 'YZ') 'A' | ('X' | 'YZ'). This pattern is then used to match the reference string S .

The above evaluation rules require that all pattern-valued variables are evaluated prior to the beginning of pattern matching. Pattern valued variables and the anonymous patterns created prior to pattern matching always have as their value a structure of pattern constants built up from primitive pattern elements by pattern building operations.

It is sometimes convenient to allow patterns to contain pattern-valued variables which are assigned values during pattern matching. This is accomplished in SNOBOL4 by the deferred pattern evaluation operator * . (unary)

```
X = 'A' *P 'B'
```

The second component of this pattern is not evaluated when this assignment statement is executed. Execution of this statement creates a pattern reference block with a pattern-valued variable whose value is assigned only during pattern matching.

The * operator is needed for reasons similar to the unary naming operator in section 7 . A quoted instance of a string denotes the pattern component consisting of the literal string, while an unquoted instance denotes immediate replacement of the string by its value. A constituent *P denotes a pattern matching constituent which, when it is encountered by the pattern matching procedure during execution, causes the pattern matching procedure to execute itself reentrantly using as its parameters the value of P and some additional information regarding the minimum length of the pattern which follows P and the current position in the reference string S .

A value must be assigned to P prior to the time when it is encountered in the pattern matching operation. This can be accomplished either by an explicit assignment statement or by dynamic assignment during a pattern matching operation.

```
P = LEN(1) $ X *X
```

This pattern specifies dynamic assignment of a value to X immediately prior to its occurrence as a pattern valued variable. This pattern will therefore match any sequence of two identical characters.

```
  P = LEN(1) $ X *X
  S = 'ABBC'
S P = 'Y'
```

The first constituent LEN(1) of the pattern will first succeed in matching A and will assign A as the value of the string (pattern) X . The second constituent *X whose value is now A , will fail to match the second character of the string. LEN(1) will then match the second constituent B of S , and assign B as the value of X . The constituent *X whose value is now B wil match the third string constituent, pattern matching will succeed, and S will be assigned the value AYC .

The facility introduced above of matching only on occurrence of two identical instances of pattern is a powerful one that cannot easily be simulated by previously defined pattern matching facilities.

A second important use of deferred pattern evaluation arises in recursive pattern specification.

```
P = 'C' | P 'A'
```

This pattern specification defines the new value of the pattern P to be C or the old value of P followed by an A .

```
P = 'C' | 'A' *P
```

This pattern specification defines the pattern P to be either a C or an instance of A followed by the value of the pattern P at pattern evaluation time. In this definition *P denotes the pattern as a whole, rather than the previous value of P .

```
  P = 'C' | 'A' *P
  S = 'AAC'
S P = 'X'
```

When matching S at string position 1 , the first alternative fails, the first component A of the second component succeeds, and *P is then evaluated as 'C' | 'A' *P. C fails to match the second character, the first component A of 'A' *P matches the second character, and *P is again evaluated as 'C' | 'A' *P. This time the first component C matches the third string character, resulting in a successful match of the third definition of *P . Success is transmitted back to the second level that *P has been successfully matched. Since *P is the last pattern component of the second level success is transmitted back to the first level, resulting in a successful pattern match and S is assigned the value X .

The explicit pattern which is obtained by nested evaluation of *P down to level 3 is as follows:

P = 'C' | 'A' ('C' | 'A' ('C' | 'A' *P)) When this pattern is evaluated by ordinary rules of pattern matching it matches AAC without encountering the pattern consituent *P .

A pattern specification in which the right hand side contains a deferred instance of the variable on the left hand side is said to be a recursive pattern specification. Recursive pattern specifications allow the specification of infinite classes of patterns without the use of primitive infinite classes such as ARBNO(P) . The recursive pattern specification in the previous example is clearly equivalent to the specification ARBNO('A') 'C' . However it follows from the theory of context free grammars[4] that there are certain recursively specifiable patterns which cannot be constructed from previously specified primitives.*

A given pattern specification is said to be left recursive if one of its alternatives contains a deferred instance of the variable on the left hand side as its first component.

P = *P 'A' | 'C' This pattern specification is left recursive because *P occurs as the first component of the first alternative.

Left recursive alternatives could lead to infinite evaluation loops if the pattern matching algorithm blindly attempted matching of the left recursive component at successively deeper levels of nesting.

---

*This statement is strictly true only in the absence of dynamic naming. In the absence of dynamic naming and deferred pattern recognition the class of sets which can be built up from primitive patterns by pattern building operations is essentially the regular sets. It is not clear to what extent dynamic naming without deferred pattern recognition increases the class of definable sets.

However, use of the length criterion for termination ensures that left recursive alternatives such as *P 'A' which have a pattern of length at least one following the left recursive constituent, will always fail at a finite level of nesting.

```
  P = *P 'A' | 'C'
  S = 'CAA'
S P = 'X'
```

The pattern matching process in the third statement would result in nested evaluation of *P until, at the second level, it was found that there were insufficient characters in S to match *P followed by three characters. Failure of the first alternative at the second level would occur, and the second alternative C at level 2 would successfully match string position 1 . The A at level 1 would successfully match string position 2 , resulting in a value of *P = 'CA' being passed to level 0 . The A at level 0 would then match resulting in an overall successful pattern match.

The explicit pattern which is obtained by nested evaluation of P to level 3 is as follows:

```
P = ((*P 'A' | 'C') 'A' | 'C') 'A' | 'C'
```

The first inner component of this pattern will automatically fail since if *P has minimum length 1 , then matching using the first inner component would require at least 4 characters. Use of the second inner component leads to matching of the string CAA by ordinary pattern matching rules.

When a deferred pattern component occurs in a pattern the minimum length of the pattern it denotes cannot in general be predicted. In SNOBOL4 it is assumed that the minimum length of a deferred pattern constituent which has not yet been matched is precisely one.

```
P = *Q *R *S
```

When matching *Q only those instances of Q will succeed which leave at least two characters of the reference string to be matched by *R and *S .

The above rule can lead the pattern matching rule to fail in certain instances when there is a matching pattern.

A → B
B → C
C → 'X'

```
  P = *Q *R
  Q = 'ABC'
  R =
  S = 'ABC'
S P = 'Y'
```

The fifth statement will fail because the length criterion only allows matching of *Q to succeed if one character is left for matching *R . In this instance *R has zero length and the assumption that *R is of minimum length 1 is erroneous.

The assumption that deferred patterns have a minimum length of 1 can prevent infinite loops when a left recursive alternative is followed by a deferred pattern specification.

```
  P = *P *Q | 'A'
  Q = 'B'
  S = 'ABB'
S P = 'X'
```

The fourth statement will treat *Q as a constituent of length 1 . The first alternative will fail at level 2 just as though *Q had been a literal of length 1 , and the pattern match will eventually succeed.

Since it is not in general possible to predict the minimum length of a deferred pattern constituent prior to its evaluation, some assumption as to its minimum length must inevitably be made. The most rigorous assumptions would be that all deferred pattern constituents had a minimum length zero. This assumption would ensure that the length criterion would never cause failure erroneously. However the assumption that deferred pattern constituents have minimum length zero would result in infinite loops for certain other well defined patterns, and would cause nontermination rather than failure in many other instances. It was felt that the minimum length 1 assumption would fail only in relatively esoteric cases, and would result in a more practical pattern matching algorithm than the length zero assumption.

19. Pattern Charts and Syntactic Recognition.

A pattern specification may be represented by a two-dimensional tree-like structure referred to as a pattern chart.

P = ('A' | 'B') ('C' | 'D') | 'E' This pattern specification can be represented by the following pattern chart.

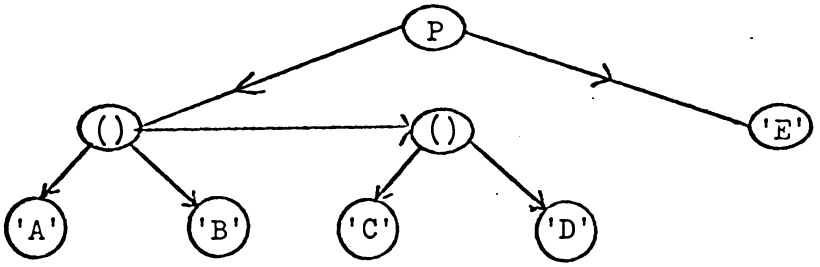


Figure 5. The pattern chart for P = ('A' | 'B') ('C' | 'D') | 'E'

In a pattern chart, horizontal arrows specify concatenation, while arrows with a vertical component specify that the sequence of constituents at the lower level is an alternative of the higher level constituent. The number of downward arrows emanating from a given constituent specify the number of alternatives associated with that constituent.

In the pattern chart of Figure 5, P has two alternatives. The first alternative of P consists of a sequence of two anonymous constituents represented by () , each having two alternatives. The second alternative of P has just a single constituent.

The number of vertical levels required to represent a syntactic chart is related to the maximum depth of nesting of expressions which contain alternatives.

P = 'A' ('B' | 'C' ('D' | 'E')) This pattern P has the following syntactic chart with four vertical levels.

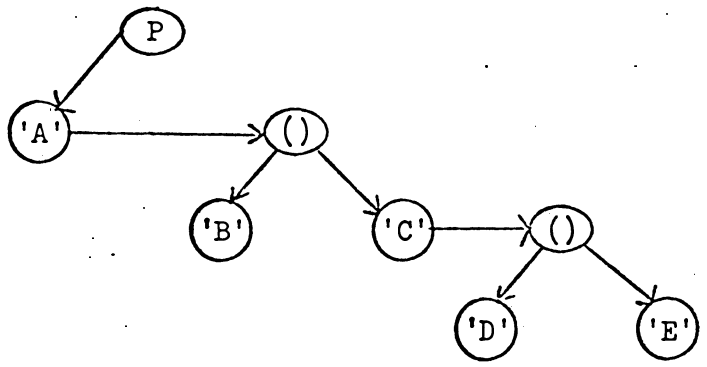


Figure 6. The pattern chart for P = 'A' ('B' | 'C' ('D' | 'E'))

Constituents in a pattern chart which call for deferred pattern evaluation result in dangling non-terminals which, during pattern evaluation, will constitute the root vertex of a "subpattern chart" corresponding to the current value of the pattern P .

Pattern matching at a given string position can be thought of as a process of sequencing over the vertices of the pattern chart, taking the leftmost untried alternative of successive constituents and attempting to match the sequence of concatenated constituents.

The term "terminal vertex" will be used to denote any vertex of a pattern chart which has no downward arrows emanating from it. The parent vertex of a given vertex of a pattern chart is the higher level vertex which points to the beginning of the chain of concatenated constituents in which the given vertex occurs. All vertices of a pattern chart other than the root vertex have a parent vertex.

Primitive pattern functions may be represented in a pattern chart by terminal vertices.

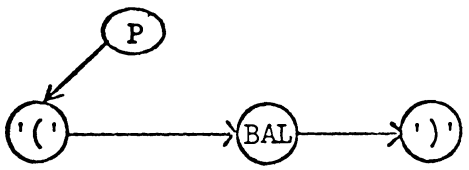


Figure 7. Representation of the pattern P = '(' BAL ')'

Every terminal vertex of a pattern chart may be thought of as a rule for determining whether the pattern constituent associated with the vertex occurs at the current point of the reference string. Terminal vertices representing literals have very simple matching rules. Terminals representing primitive pattern functions may have very complex matching rules.

Deferred pattern variables may also be represented by terminals in a pattern chart.

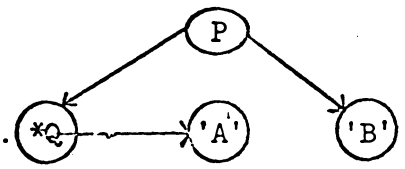


Figure 8. Pattern chart for P = *Q 'A' | 'B'

A deferred pattern variable is replaced by its current value during pattern matching. In pattern chart terms it may be thought of as a vertex to which a subtree will be attached during execution, where the form of the subtree is not in general known prior to execution. The value of a deferred pattern variable occurring in a given pattern may be changed by dynamic naming, so that the value of a given deferred pattern variable may take on an unpredictable sequence of different values during matching of a given pattern.

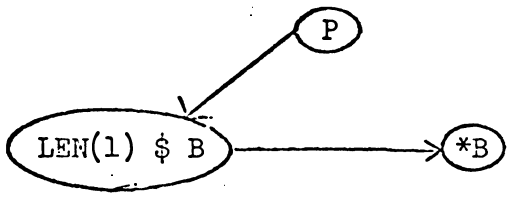


Figure 9. Pattern chart for P = LEN(1) $ B *B

In the above pattern chart the deferred pattern variable *B will have different subtrees associated at different points of execution.

Recursive patterns allow patterns with an arbitrarily large implicit nesting depth to be defined.

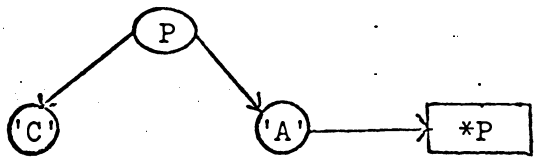


Figure 10. Pattern Chart for P = 'C' | 'A' *P

In the above pattern chart the pattern *P is a fixed pattern. However, this fixed pattern contains a deferred pattern specification of itself, allowing an arbitrarily large number of recursive replacements of *P by a copy of itself. Recursively defined instances of a given pattern constituent will be enclosed in a square box. The occurrence of a square box in a pattern chart is analagous to the occurrence of a procedure call in a program. When a square box is encountered during pattern matching, it is replaced by an instance of the pattern definition and an attempt is made to match the constituents of the pattern definition. If all square boxes were replaced by their definitions prior to pattern matching, then an infinite pattern chart would be obtained. Replacement of square boxes

by their definitions as they are required during pattern matching results either in successful matching after a finite number of steps or in failure and backtracking after a finite number of steps.

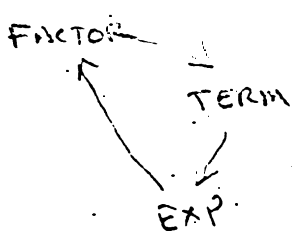
Pattern matching at a given string position can be thought of as a process of sequencing over the vertices of the pattern chart, taking the leftmost untried alternative of non-terminal constituents, replacing deferred pattern variables by their values as they are encountered and attempting to match other terminal vertices directly against the reference string.

Deferred pattern variables allow the specification of context-free grammars in a manner that is very similar to Backus-Naur form (see Ref. 3). The following set of interconnected pattern specifications create a pattern P which matches a simple class of arithmetic expressions.

```
VARIABLE = ANY('XYZ')
  ADDOP = ANY('+-')
  MULOP = ANY('*/')
 FACTOR = VARIABLE | '(' *EXP ')'
   TERM = *FACTOR | *TERM MULOP *FACTOR
    EXP = *TERM | *EXP ADDOP *TERM
      P = POS(0) *EXP RPOS(0)
```

<VARIABLE> ::= X | Y | Z
<ADDOP> ::= + | -
<MULOP> ::= * | /
<FACTOR> ::= <VARIABLE> | (<EXP>)
<TERM> ::= <FACTOR> | <TERM><MULOP><FACTOR>
<EXP> ::= <TERM> | <EXP> ADDOP TERM

The above pattern specification can be represented by the following pattern chart.



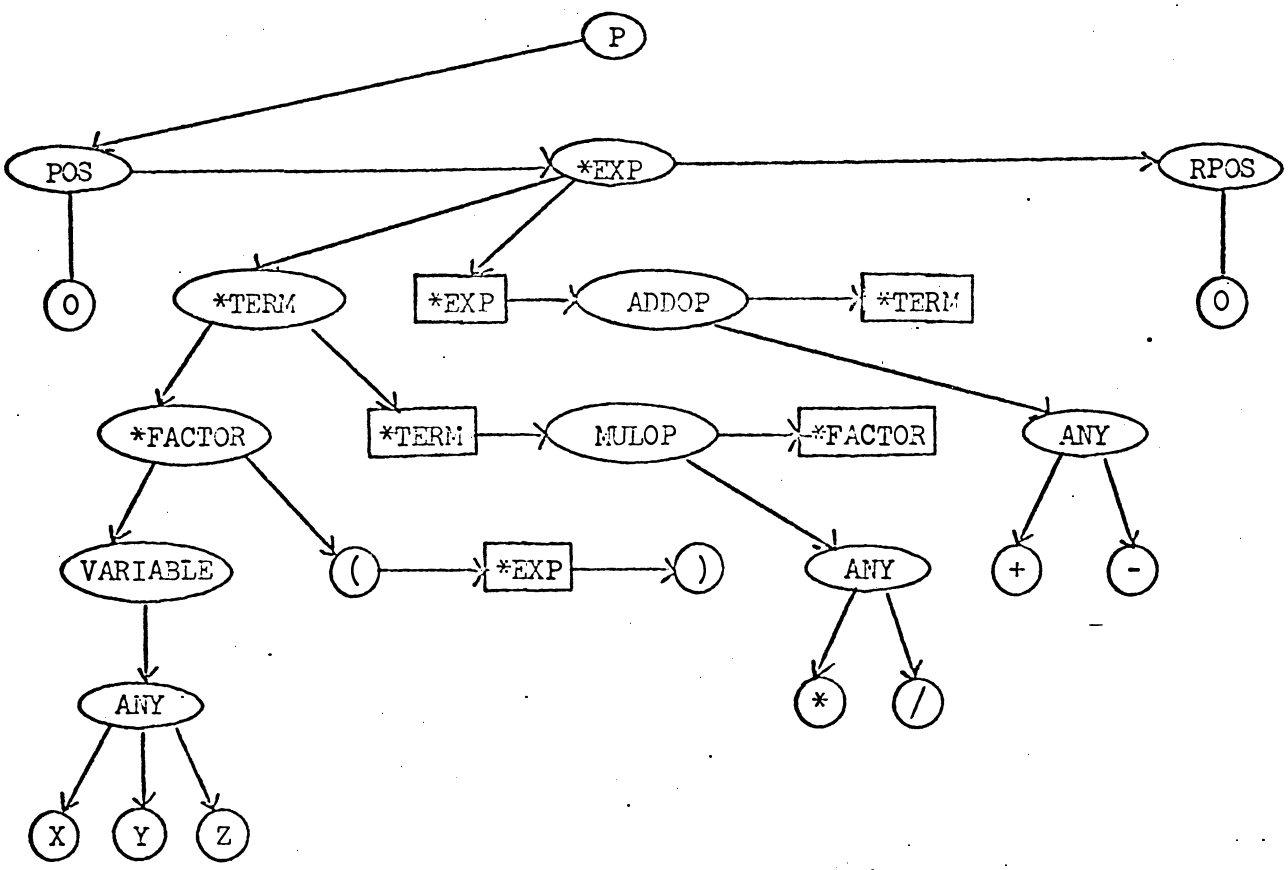


Figure 11. Pattern chart for an arithmetic expression pattern

A pattern is said to impose a structure on the set of strings which it recognizes. If a string S is an instance of a pattern P and is matched by an alternative which consists of K pattern constituents P1 ... PK , then P partitions S into K substrings S1 S2 ... SK where successive substrings S1 S2 ... match successive pattern constituents P1 P2 ... .

```
  S = 'X+Y'
  P = ARB '+' LEN(1)
S P = 'Y'
```

When S is matched by P , then P imposes a structure on S which partitions S into three constituents such that the first is an instance of ARB , the second is an instance of + , and the third is an instance of LEN(1) .

The structure associated with a given string is not an inherent property of the string but is determined by the pattern which matches it. Since pattern matching may result in actions depending on matched constituents, the structure associated with a string S by a pattern P may be thought of as a decomposition of the string into components which may trigger actions. The notion of structure is taken even further in syntactic analysis and is used as a basis for specifying the semantics of a given string in terms of the semantics of its substrings.

When a pattern is defined in terms of a hierarchy of deferred pattern constituents, it is convenient to specify the structure imposed on a string S by a pattern P as a tree whose root vertex represents the complete pattern P, whose terminal vertices represent the string constituents and whose non-terminals represent pattern constituents which have been successfully matched.

If S = 'X+Y' is matched by the pattern P defined above for arithmetic expressions, then the structure imposed on S may be represented by the following tree.

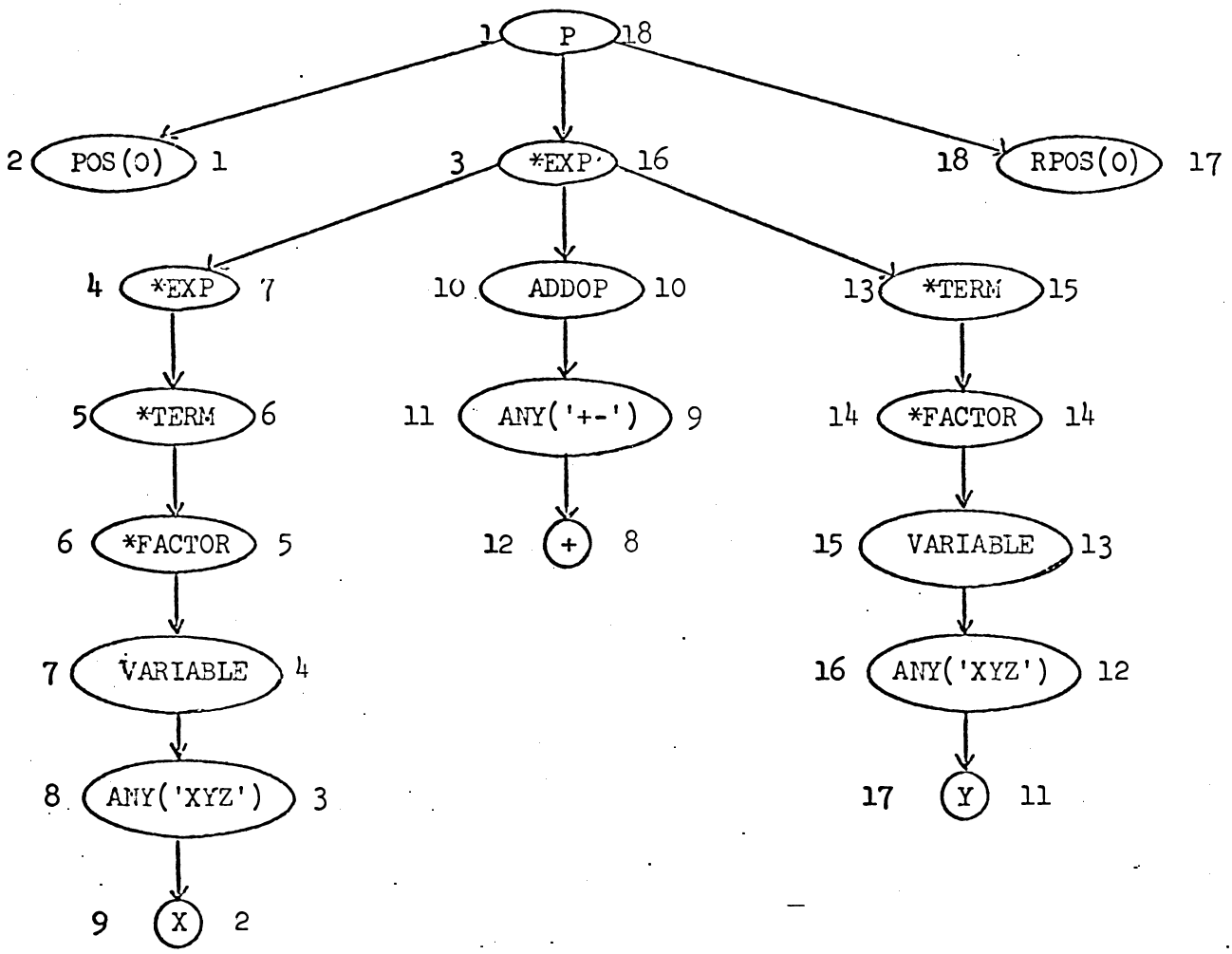


Figure 12. Tree Structure imposed by P on S = 'X+Y'

Matching of the string S by the pattern P involves successful matching of pattern elements corresponding to each of the vertices of the above tree. The vertices have been numbered on the left in the order in which the pattern matching process attempts to match constituents, and have been numbered on the right in the order in which successful matching of a constituent is registered. The numbering on the left represents the order in which pattern elements are first encountered during the pattern matching process. The numbering on the right represents the order in which dynamic names are assigned during pattern matching and also the order in which static

names are assigned on successful pattern matching. Name-value correspondences for static names are stored in a "static name stack" which is built up during pattern matching and used in a first-in-first-out order on successful pattern matching.

In matching of the string S = 'X+Y' by the pattern P, matching succeeds only after many partial failures. The sequence of pattern matching steps which lead to matching of P by the above tree are illustrated in the following example.

```
  S = 'X+Y'
S P = 'EXPRESSION'
```

The second statement will first match POS(0) and then try successive first alternatives of EXP, TERM, FACTOR, VARIABLE and ANY. ANY succeeds in matching X, success is passed all the way up to the second constituent *EXP of P, and matching of the third constituent RPOS(0) then fails. Backtracking then occurs, the pattern ANY fails, matching of the second alternative of FACTOR is attempted and fails immediately because the opening parenthesis does not match, and matching of the second alternative of TERM is attempted. This generates FACTOR, VARIABLE and ANY, results in a successful match of X, but fails when MULOP cannot be matched. Backtracking then results in matching of the second alternative of EXP which matches X using successive first alternatives of EXP, and then matches + and Y, associating the structure X + Y with S. This structure is destroyed as soon as the pattern is successfully matched, and the value EXPRESSION is assigned to S.

20. The Representation of Patterns

The information structures which arise during pattern matching include a static representation of the pattern which is fixed at the time of pattern specification, and some dynamically changing structures which represent the point reached by the pattern matching procedure in generating the ordered set of strings determined by the pattern specification.

The static representation of a pattern in SNOBOL 4 consists of a sequence of linked pattern elements. A pattern element consists of a group of information items which includes a type specification, the next concatenated constituent (if any), and the first terminal vertex of the next alternative (if any). Information specifying the minimum number of characters of pattern constituents which follow the given pattern element is also included in the pattern element. The information fields of a pattern element are illustrated in Figure 13.

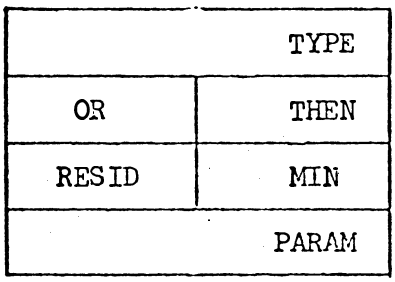


Figure 13. Format of a pattern element

The type field specifies the name of the procedure which is to be used for matching the given pattern constituent. The OR field specifies the alternative pattern element to be tried when the given pattern constituent fails. The THEN field specifies the next pattern element to be tried when the pattern constituent succeeds. The RESID field specifies the minimum number of characters of the pattern constituents which follow while the MIN field contains the sum of RESID and the minimum number of characters in untried instances of the given pattern element. The field PARAM contains a descriptor of the parameter when the type field specifies a one parameter function, and is otherwise empty.

A pattern specification in SNOBOL4 is represented by a sequence of pattern elements with interconnected OR and THEN fields.

P = LEN(2) 'AB' | BAL *P

This pattern is represented by the following pattern chart.

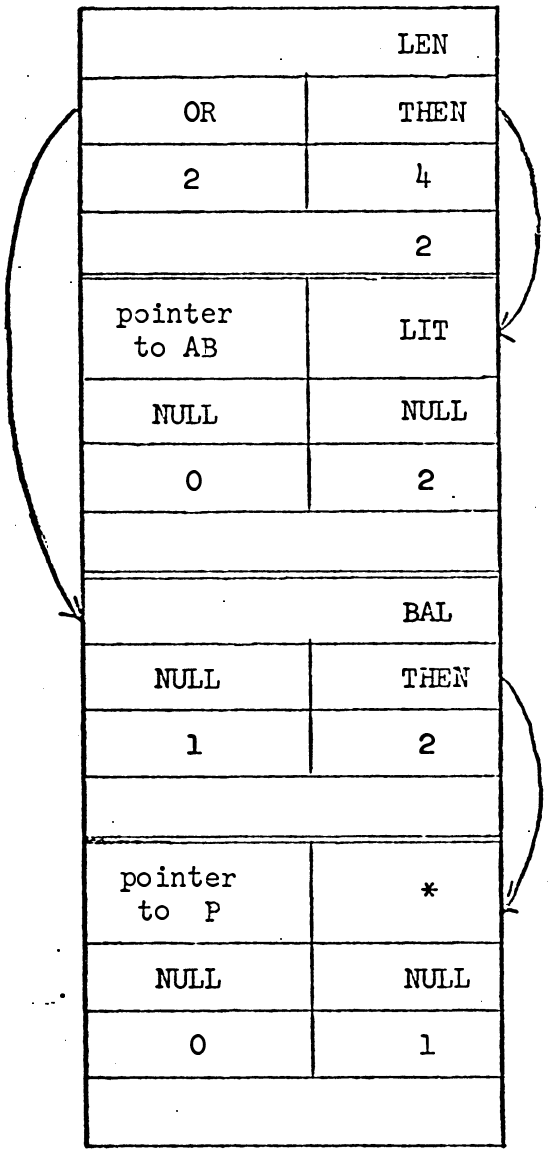


Figure 14. Representation of P = LEN(2) 'AB' | BAL *P

The information in the type field of a pattern element is a procedure call to a procedure for generating successive instances of the ordered set specified by the pattern constituent and matching the generated

instances at the current point of the reference string. The procedure has as its parameter the remaining information fields of the pattern element. During execution it generates some temporary information which specifies the point reached in generating instances of the pattern element. The procedure may call another procedure through its OR and THEN links, or through its deferred pattern pointer if the constituent is a deferred pattern variable.

During execution a stack is required to keep track of successive pattern constituents whose alternatives are being explored. The set of information items stored in the stack to keep track of alternatives for a given pattern element will be called the activation record of that pattern element. The activation record of a given pattern element consists both of the parameters specified in the static representation and of the temporary information generated by the pattern matching procedure. When a procedure has been entered, its activation record will remain on the stack until all its alternatives have failed. When this happens, the OR branch is taken, so that the OR branch corresponds essentially to the return link of a regular procedure.

When a pattern constituent succeeds, the THEN link is tried. If the given pattern element has no successor, the THEN link is null and success is passed back to the previous constituent without deleting the successful pattern elements. Success is passed upwards through the stack until a pattern element with a THEN link is encountered. If matching of all THEN links reached in this way is successful, the complete pattern has been matched successfully. If failure occurs, backtracking is required to try alternatives of the activation record currently at the top of the

stack (corresponding to the most recent successfully matched element). If an element runs out of alternatives or causes length failure, further backtracking is performed. If, during backtracking, the first procedure in the stack runs out of alternatives, then pattern matching fails for each string position.

Dynamic naming during pattern matching is performed immediately on successful matching of a constituent and no provision need be made for remembering to assign a value for the dynamic name. Static naming, however, requires remembering of the association between names and values since assignment is performed only when pattern matching has been completed successfully. The association between values and static names can be remembered either in the activation record stack for pattern elements, or in a separate stack.

In the example for S = 'X+Y' in the previous section a total of eighteen activation records of pattern elements will be in the stack in the order indicated by the numbering on the left of vertices in Figure 12. Each of the vertices of the tree structure essentially corresponds to an activation record. Conversely, the sequence of activation records on the stack at any given intermediate point of the pattern matching process corresponds to a partial tree of a partially matched string.

Entry to and exit from functions and matching of pattern constituents are performed mutually in a last-in-first-out order; i.e. if a function is called during pattern matching then exit from the function must occur before pattern matching is resumed, while if a pattern is matched during a function call the pattern match must be completed before execution of

the function is resumed. Activation records of pattern elements and activation records of function calls can, therefore, be stored in a single run-time stack.

21. Relational Functions and Deferred Expression Evaluation

SNOBOL4 contains a number of relational functions for testing the relative magnitude of integers or the equality of strings. These functions return the value SUCCESS if the relation holds and the value FAIL if the relation does not hold.

The six relational functions EQ , NE , GT , GE , LT , LE correspond respectively to the six relations $=$ $\neq$ $>$ $\geq$ $<$ $\leq$. Each function takes two arguments.

EQ(N,M) NE(N,M) GT(N,M) GE(N,M) LT(N,M) LE(N,M)	Each function returns the value SUCCESS if both N and M have integer values and the specified relation between N and M is satisfied. Otherwise, the function returns the value FAIL .

The values SUCCESS and FAIL associated with relational functions cannot be assigned as values of variables in SNOBOL4 , but can be used only to trigger the functions F and S in the GO TO part of a statement. When a relational function returns the value FAIL , execution of the statement is immediately terminated*, and control is transferred to the GO TO part of the statement. When a relational statement returns

---

*When a relational function occurs as a pattern matching constituent, its failure is equivalent to occurrence of the system constituent FAIL and results in matching of the next alternative. However, the occurrence of unevaluated predicates as pattern constituents can be accomplished only by use of the system function DEFPEX which defers evaluation of expressions occurring in pattern specifications. Deferred expression evaluation is further discussed below.

the value SUCCESS , the next constituent of the statement is executed just as though it were a constituent of a pattern.

Relational functions may appear in any portion of a statement other than in the GO TO part.

```
      X = EQ(N,M) 5 : S(L1)F(L2)
      Y = EQ(N,M)   : S(L1)F(L2)
Z EQ(N,M) = 5       : S(L1)F(L2)
```

If N = M , the first statement assigns 5 to X and goes to L1 , the second statement assigns NULL to Y and goes to L1 , and the third assigns 5 to Z and goes to L1 . If $N \neq M$ each of the statements FAIL before assignment can take place and control is returned to L2 .

In SNOBOL4 there is a predefined lexicographic ordering of characters which imposes a lexicographic ordering on all strings of the language. The relational function LGT(X,Y) , (lexicographically greater than) is a two-argument function which returns the value SUCCESS if the value of X is lexicographically greater than the value of Y , and the value FAIL otherwise.

```
X = 'ABC'
Y = 'DEF'
X = LGT(X,Y) : S(L1)F(L2)
```

The function LGT(X,Y) in the third statement would return the value FAIL since ABC precedes DEF in the lexicographic ordering. Control would then be transferred to L2 .

The relational functions IDENT and DIFFER may be used to test the identity and non-identity of strings.

IDENT(X,Y) — This function returns the value SUCCESS if the values of X and Y are both of the type STRING and the strings are identical. Otherwise, the function returns the value FAIL .

DIFFER(X,Y) — This function succeeds if X and Y have different strings as their values, and fails otherwise.

Predicates which occur in a pattern specification are evaluated at pattern specification time rather than at pattern matching time.

P = Q EQ(X,Y)	If Q has a value of the type PATTERN then this statement will assign Q to P if the predicate EQ(X,Y) succeeds, and will leave P unchanged if EQ(X,Y) fails.

It is sometimes convenient to evaluate predicates and other expressions during pattern matching rather than during pattern specification. This can be done by the system function DEFPEX * (deferred pattern expression).

DEFPEX(E)	If this expression occurs as a pattern constituent, it causes evaluation of E whenever the pattern matching procedure calls for evaluation of this constituent.
Q = P DEFPEX(EQ(X,Y))	If P is of the type PATTERN then the pattern Q will match a given reference string if and only if EQ(X,Y) succeeds after P has been matched.

The function DEFPEX is essentially a generalization of the deferred pattern variable operator. Thus, if P is of the type PATTERN then DEFPEX(P) is equivalent to *P . However, DEFPEX can be used to defer the evaluation of arbitrary expressions such as predicates.

The following pattern specification allows matching of any string consisting of a sequence of A's followed by an equal number of B's.

---

* DEFPEX is not available in the preliminary distribution of SNOBOL4 dated January 1968, but will be available in later distributions. Note that the function DEFPEX is treated differently from other functions in that argument evaluation is inhibited whenever it is called. This requires the SNOBOL4 interpreter to make a test on every function call to determine whether the called function is DEFPEX . DEFPEX could alternatively have been implemented as a generalization of deferred pattern variables by allowing a * followed by parentheses to denote deferred evaluation.

```
P = SPAN('A') $ X SPAN('B') $ Y DEFPEX(EQ(SIZE(X),SIZE(Y)))
```

The pattern P will match the longest string of A's and assign it to X , then match the longest string of B's and assign it to Y and then test for equality of the length of these two strings. Note that the use of DEFPEX to defer evaluation of the expression is essential.

```
Q = POS(0) P RPOS(0)
```

Whereas P will match a substring consisting of an equal number of A's and B's , Q will only match if the complete string consists of precisely N A's followed by N B's for some integer $N \geq 1$.

```
  S = 'AAABB'
S P = 'X'
S Q = 'X'
```

If P and Q are as above, the second statement would match positions 2 through 5 of the string S , while the third statement would fail.

There are two unary operations which may be used to test the success or failure resulting from the evaluation of expressions. These operations suppress the value associated with the expression and may manipulate the indication of success or failure.

The operation ¬ suppresses the value of the expression which constitutes its argument, and reverses the success-failure indicator.

```
X = ¬A<N> N
```

A<N> fails if N is outside the range of the array A . ¬A<N> suppresses the value of A<N> and succeeds only if A<N> is outside the range of A . Thus X is assigned the value N if N is outside the range of A and is left unchanged otherwise.

```
A<0> = ¬A<N> ¬A<-N> 0
```

This statement assigns 0 to A<0> only if A<N> and A<-N> are both outside the range of A and if A<0> is in the range of A .

The operator ? merely suppresses the value of its argument and leaves the success-failure indicator unchanged.

A<0> = ?A<N> ¬A<-N> 1 — This statement assigns 1 to A<0> only if A<N> is in the range of A , A<-N> is outside the range of A and A<0> is in the range of A .

22. Keywords

It is convenient in a programming system to allow the user access to certain internal system constants and system variables. System quantities to which the user has access are denoted by a special class of names, whose first character is always the ampersand symbol & . These names are referred to as keywords. Keywords provide an interface between a SNOBOL4 program and certain internal symbols of the SNOBOL4 system.

Keywords may be either protected or unprotected. Protected keywords may be accessed only in a read only mode, and may not be written into. Unprotected keywords may be both read and written into.

The protected keywords can be classified into system variables and system constants.

There are two protected system variables.

The protected system variable &STCOUNT counts the number of statements whose execution has been started (but not necessarily completed) for the given program.

The protected system variable &STFCOUNT counts the number of statements which have failed in the given program.

There are seven protected system constants. These include the system constant &ALPHABET whose value is a string containing precisely one copy of each of the characters of the SNOBOL4 alphabet, and the six primitive pattern constants &ARB , &ABORT , &BAL , &FAIL , &FENCE , &REM .

The six protected pattern constants initially have the same value as the corresponding pattern functions without the ampersand. However, the pattern functions without the ampersand are unprotected, and may have their values reassigned during program execution, while the protected pattern functions cannot be modified. The protected pattern functions may be used to reinitialize modified unprotected pattern functions to their original values whenever desired.

Only parameterless primitive pattern functions have protected system constants associated with them. Primitive pattern functions with parameters cannot be modified by value assignment, since they are functions. However, their functional attribute can be redefined by occurrence of the name as a function name in a DEFINE statement. Once a function like ARBNO or LEN has been redefined by a DEFINE statement, it cannot be reconstructed as a primitive.

The unprotected keywords may be classified into switches which may be either ON or OFF , and modifiable system variables. OFF is represented internally by the null string descriptor, while any other value of a switch variable is interpreted as ON.

There are currently two system switches in SNOBOL4.

The system switch &ANCHOR specifies normal pattern matching when it is OFF and requires patterns to match starting at string position 1 when it is ON . When &ANCHOR is ON , every pattern P is matched as though it were preceded by an implicit POS(0) .

The system switch &DUMP will cause dumping of variable storage when it is on, and will inhibit dumping when it is off.

There are currently two modifiable system variables in SNOBOL4.

The system variable &MAXLENGTH specifies the maximum length of BCD strings that may be created during execution of a program. This system variable initially has the default value 5000 but may be set to some other integer value by the programmer.

The modifiable system variable &STLIMIT is the limit on the number of statements that may be executed in a given program.

If either &MAXLENGTH or &STLIMIT is exceeded then error termination results. Resetting of &MAXLENGTH to a lower value results in error termination only if a longer string is created after &MAXLENGTH has been reset, and does not lead to an error if already existing strings are above the limit. Error termination on &STLIMIT occurs as soon as the number of initiated statements exceeds the limit.

REFERENCES

1. R. E. Griswold, J. F. Poage and I. P. Polonsky, Preliminary Report on the SNOBOL4 Programming Language, Bell Telephone Laboratories, Holmdel, New Jersey, November 22, 1967.

2. D. J. Farber, R. E. Griswold and I. P. Polonsky, The SNOBOL3 Programming Language, Bell System Technical Journal, July-August, 1966.

3. P. Wegner, Programming Languages, Information Structures and Machine Organization, McGraw Hill, 1968.

4. S. Ginsburg, The Mathematical Theory of Context Free Languages, McGraw Hill, 1966.