SNOBOL
CS 355
©Denbigh Starkey

Background

SNOBOL (**S**tri**N**g **O**riented sym**BO**lic **L**anguage) was created in 1962 at Bell Labs by Griswold *et al*. as the first language whose only goal was to process strings. It was subsequently developed further by the SNOBOL group at the University of Arizona that included Griswold and Griswold. The language is, in today's terms, very ugly since it predates most modern program design thought. E.g., it is clearly designed for a card-based environment (as was Fortran), and it is the only high level programming language that I've used where in a typical program over 50% of the lines of code contain at least one embedded goto statement.

The SNOBOL that I will be discussing is the last major version, SNOBOL4, which was described in "the green book" *The SNOBOL4 Programming Language*, *2nd Edition*, Griswold, R.E., Poage, J.F., & Polansky, I.P., Prentice-Hall, 1970. If people use the name SNOBOL without qualification they are referring to SNOBOL4, just as Algol without qualification refers to Algol 60. The only SNOBOLs that I used were the original SNOBOL, SNOBOL3 (described in "the black book") and this SNOBOL.

Obviously a question arises on why we should even look at SNOBOL today. The primary reason in a class like CS 355 is that we want you to get a feel not only for the current programming languages, but also for the past languages, so that students can get a better understanding on where the current languages got their features and how developments occur.

A secondary reason is that despite the forbidden fruit aspects of programming in the language, I loved programming in it and used it whenever I could, even if it wasn't the most appropriate language. Until Perl came out I haven't used a language that I have enjoyed as much. Perl is also the first language that I've used since then that has had comparable string handling capabilities.

A final reason is that the old languages had some features which were, from a programming standpoint, very powerful, but which have been dropped from modern languages because nobody has been able to implement them efficiently enough. These include, for example, Algol's Call by Name parameter passing, which has only been implemented through thunks, and the ability of SNOBOL programs to modify themselves.

The Power of SNOBOL

Another obvious question is why we want to be able to process strings. A large number of problems that we deal with can be thought of as string manipulation problems.

- A compiler is a string processor. It takes an input string (the user program) and outputs a string (the executable program). So if, for example, you are developing or working with a new language, and want to hack out a (very inefficient) compiler so that you can try out new features, the easiest way to do this is through a string language. This is particularly good in a language like SNOBOL whose programs had self-modifying capabilities. If I wanted a SNOBOL program to take a program in language *X* as its input, I could read in the lines written in *X* and use pattern matching to convert them to SNOBOL, and then have the program modify itself to include that new code, and execute it.

- Say you have an old Pascal program that you want to convert to C, a SNOBOL translator would be the easiest way to go.

- Many problems in modern biology and chemistry involve sequence matching or other string-based problems. A string manipulation and pattern matching language will be most convenient for this.

- Linguistics and other language studies depend on string pattern matching.

SNOBOL had some features that have rarely been seen since its demise. As I mention above, its pattern matching capabilities were outstanding, and it was also self-modifying. Other features include dynamic binding of variables, which is much more intuitive for programmers than static binding, heterogeneous arrays, and hash tables. For many years after SNOBOL these features, which were much more convenient for the programmer but hard for the implementer, were left out of languages, although some are now reappearing in modern languages like Perl.

Structure of a SNOBOL program

A SNOBOL program was designed to be input on cards, and had one instruction per card unless a card was specified as a continuation card or unless two or more statements were separated by a semicolon. I'll start by looking at a simple SNOBOL program, and use this to describe the structure in more detail.

```
LOOP    LINE = INPUT                :F(END)
        LINE 'DENBIGH'              :S(LOOP)
        OUTPUT = LINE               :(LOOP)
END
```

This program takes an input file and outputs all lines that don't contain the substring `DENBIGH`. I'll worry about how it does this later. In this section I just am interested in the structure.

The first think to notice is that everything is upper case. The 026 and 029 keypunches did not have lower case.

The program has a couple of labels, which are any sequence of non-blank symbols beginning in column 1 whose first character is a letter or digit. `END` is a special label contained in all SNOBOL programs, and reaching it in any way (either through a goto or through normal flow of execution) causes termination.

The optional label was followed by an optional statement. Here we have three statements, two assignment statements and one simple pattern matching statement.

The last thing on the line was an optional goto component, which is designated by a leading colon. Here all three of the non-`END` lines contain a goto statement, with one which only executes on failure, one which only executes on success, and one which is unconditional.

Execution is sequential unless it is changed by a goto, or a function call. So in this case we'll first execute `LINE = INPUT`. If this fails (don't worry about how yet) we'll go to the `END` label and terminate. If the first line succeeds we'll move on to the pattern match, `LINE 'DENBIGH'`. If this succeeds we return to the first line, but if it fails we'll move on to the third line, `OUTPUT = LINE`. We'll then unconditionally return to the first line.

In the program I've lined up instructions and labels, but that is just for readability.

If an instruction cannot fit on one line then the continuation line(s) were designated by a + in column 1.

Apart from restrictions that the first column is reserved for the continuation symbol or for a label, the lines were free format.

Normally we'll just have one instruction per line. If we want more they are separated by semicolons. E.g., later in these notes I give a program that includes the line:

```
&FULLSCAN = 1;  &TRIM = 1
```

SNOBOL Statements

The four basic SNOBOL statements were:

- Assignment statement
- Pattern matching statement
- Replacement statement
- `END` statement

I've already discussed the `END` statement, and won't discuss it further, except to mention that function definitions and code created during execution appear after the `END` so that the program can't fall into either accidentally but must be accessed through an explicit call.

In the next three sections I'll describe the other three statement types.

Assignment Statement

The simple assignment statement has the common form

variable = *value*

There are few restrictions on these assignments, and so, for example, variables aren't restricted by type, and the consecutive statements

```
BOOK = 22
BOOK = 'CATCH' BOOK
```

where `BOOK` first contains the integer `22` and then, through concatenation, the string value '`CATCH22`', are completely legal. Variables aren't declared, and their types just depend on their latest value. There are three basic types that they might have at any time, string, integer, and real. Real numbers aren't common in SNOBOL-type applications, and so I'll ignore them.

Note the use of a space as a concatenation operator. SNOBOL, amazingly, uses a space for three different operators, depending on context, and so later we'll get to statements like `A B C = D E`, where the space between `A` and `B` is a pattern match, the space between `B` and `C` is a pattern concatenation, and the space between `D` and `E` is a string concatenation. This doesn't turn out to be as confusing as you might expect.

The variable name can be any non-null string, and so if you want a variable named, say, `X = Y`, then you can create it. However a variable with a name like this or, say, three spaces, could cause problems, and so can only be accessed through indirect references. E.g., the statement `X = Y = Z` can not be used to assign `Z` to the variable named `X = Y`, but `$'X = Y' = Z` can be used to do this. Variable names which can be accessed without indirection consist of any number of letters, digits, periods, and underscores, starting with a letter.

Since there are no variable declarations, at the beginning of execution all variable names can be thought to exist. They will be initialized to either 0 or the null string, depending on how they are first evaluated. If they are first used in an integer context they will be given the value 0, if they are first used

in string context (e.g., in a concatenation or pattern match) they will be given the default value of the null string.

There are three special I/O variables, INPUT, OUTPUT, and PUNCH, which have special meanings. Any time INPUT is evaluated the value is the next line from the standard input stream (typically line 5, the card reader). Any time a value is assigned to OUTPUT it is sent to the standard output stream (typically line 6, the printer). Similarly if a value is assigned to PUNCH then it is sent to the punch stream (typically line 7, the card punch). So now I can go back to the example program that I had earlier:

```
LOOP    LINE = INPUT                    :F(END)
        LINE 'DENBIGH'                  :S(LOOP)
        OUTPUT = LINE                   :(LOOP)
END
```

The first line takes the value of INPUT, which is the next input line/card, and assigns it to the variable LINE. If there is no more input available this will fail, which will be detected by the conditional goto, we'll go to END, and terminate. Otherwise we move on to the second line. This, as we'll see later, is a simple pattern match, which checks to see whether the string DENBIGH is a substring of the string in LINE. If it is we go back and attempt to read another line of input and repeat. If not we print out the line by assigning it to OUTPUT, and unconditionally go back to the first line. I.e., the result of this program will be to print out all input lines that don't contain the substring DENBIGH.

Before moving on I'll look at another simple program:

```
LOOP    PUNCH = INPUT                   :F(END)
        OUTPUT = PUNCH                  :(LOOP)
END
```

This will take an input deck of cards, print it out, and also punch out a copy of the card deck. Note that although PUNCH has the special property that any time it is assigned a value the result is punched, it is still a variable and so its value can be assigned to OUTPUT without problems.

Getting back to the basic assignment statement, expressions can be built using arithmetic and/or string operators. The basic binary arithmetic and string operators, with their precedence, are:

Symbol	Meaning	Associativity
! or **	exponentiation	right
*	multiplication	left
/	division	left
+, -	addition, subtraction	left
space	concatenation	left

Note that as usual exponentiation is evaluated right to left because (*a* ** *b*) ** *c* can be better written as *a* ** (*b* * *c*).

There are, of course, many other operators, and so this is a very simplified precedence table that is just looking at the basic arithmetic and string operators.

One unusual (unique?) precedence rule in SNOBOL is that multiplication has higher precedence than division. The average person, who hasn't been intimidated by a number of programming classes, will interpret *a* / *b* * *c* as *a* / (*b* * *c*), not as (*a* / *b*) * *c*. SNOBOL accepts this and so, for example, 12 / 2 * 3 is 2, not 18 as it would be in most other languages.[1]

Using these rules, the assignment:

```
VAR = (22 123) + 9 * 3 'DE'
```

would assign the string `'22150DE'` to `VAR`. Note that SNOBOL has to keep converting between strings and numbers as it interprets this expression.

[1] APL and FORTH would also interpret this as 2, but for a different reason. In those languages precedence is just right to left, and all operators have the same precedence.

Pattern Matching

This is where SNOBOL got its strength, and where it had a power that wasn't matched until about 30 years later with Perl. The basic pattern matching statement has the form:

subject pattern

We saw a simple version of this earlier with `LINE 'DENBIGH'`, and so obviously a pattern can be just a string, and we are testing to see whether the pattern string is a substring of the subject. We can do dramatically more than this, however. E.g., the pattern match

```
MYSTRING (('A' ('B' | 'CD')) | 'E')
```

would succeed if `MYSTRING` contained one or more of `'AB'`, `'ACD'`, or `'E'` as a substring. It is using the alternation operator, |, twice to give multiple possibilities. Because alternation has a lower precedence that pattern concatenation this would usually be written as

```
MYSTRING 'A' ('B' | 'CD') | 'E'
```

There are also a number of functions that return patterns, which I'll discuss after first looking at pattern matching with replacement.

Pattern Matching with Replacement

The basic form of this, which is also called the replacement statement, is

subject pattern = object

where *object* is a string. This will perform a pattern match using the left hand side as usual, and if this pattern match fails nothing further will happen. If, however, the pattern match succeeds, then the matched substring in subject will be replaced with the object string. Perl programmers will be very familiar with this kind of statement. E.g., the very simple code

```
        FRUIT = 'BANANA'
        FRUIT 'A' = 'CC'
        OUTPUT = FRUIT
```

will find the first `A` in `BANANA` and replace it with `CC`, and so `BCCNANA` will be output. Now let's go one step further

```
        FRUIT = 'BANANA'
ATOCC   FRUIT 'A' = 'CC'          :S(ATOCC)
        OUTPUT = FRUIT
```

will replace every `A` with `CC` and will output `BCCNCCNCC`.

Consider another example:

```
    FACULTY = 'LLOYD' | 'STARKEY' | 'ROSS'
    COMMENT = INPUT
    COMMENT FACULTY = 'HARKIN'
    OUTPUT = COMMENT
END
```

Note that a pattern can be saved in a variable. If the input is, say, `MR. LLOYD LOVES GOLF`, then the output will be `MR. HARKIN LOVES GOLF`.

Capturing Matched Pattern Components

When a pattern match succeeds it is often important to know which part of a string was matched by specific parts of a pattern. To do this we use conditional value assignment, which associates parts (or all) of a pattern with a variable using the form

pattern . *variable*

and if the whole pattern match succeeds then the part matched by this piece of the pattern is assigned to the variable. There is also an immediate value assignment operator, $, which has the form

pattern $ *variable*

and assigns any substring matched by this pattern to the variable even if the complete pattern fails.

As an example, say that the input contains an assignment statement in some language like C, with a semicolon terminator, and that we want to print out the variable on the left hand side and the expression on the right hand side.[2] E.g., if the input is `fred = 3 * x – fabs(33.5);` then I want to output two lines containing `fred` and `3 * x – fabs(33.5)`. This can be done with the following code:

```
        INPUT ARB . VAR '=' ARB . EXPR ';'  :F(END)
        OUTPUT = 'Variable is:  ' VAR
        OUTPUT = 'Expression is:  ' EXPR
END
```

where `ARB` is a special pattern which matches anything. So this is looking for anything followed by an =, followed by anything followed by a semicolon, and if all this is found then the first anything is assigned to `VAR` and the second anything to `EXPR`.

For another example, say that you want to replace any references to a dollar and cents figure in a string with just the total number of cents. E.g., if the string is `I HAVE $3.51 AND $13.27 SO A TOTAL OF $16.78`

[2] To avoid having to introduce too many new features I'll assume that the expression cannot include a semicolon.

then I want to output `I HAVE 351 CENTS AND 1327 CENTS SO A TOTAL OF 1678 CENTS`. The program below will do this:

```
        DIGITS = '0123456789'
        STR = INPUT
LOOP    STR '$' SPAN(DIGITS) . DOLLAR '.'
+           (ANY(DIGITS) ANY(DIGITS)) . CENTS
+           = DOLLAR CENTS ' CENTS'         :S(LOOP)
        OUTPUT = STR
END
```

The new functions here are `SPAN` which matches any number of characters from it argument and `ANY` which matches exactly one string from its argument. I've also used a continuation character for the first time with the + in column 1 to extend one statement over three lines.

Other Pattern Matching Goodies

SNOBOL has many operators and built-in functions and predicates that relate to pattern matching. I'll briefly describe some that we haven't seen yet below, and will use many of them in my final example.

@X assigns the current pattern cursor position to the variable X, where 0 is the position before the first letter in the subject string. So, for example

'DENBIGH STARKEY' @X ('S' | 'BIG') @Y

will assign 3 to X and 6 to Y. Technically @X returns the null string so that it doesn't mess up the pattern match.

LEN(N) matches any substring of length N. It is particularly useful for extracting fixed-length fields from a table.

SPAN(STRING) matches a sequence of characters from a string. E.g., SPAN('0123456789') matches any unsigned integer.

BREAK(STRING) matches everything up to the first occurrence of a character from the string.

ANY(STRING) matches any single character from the string.

NOTANY(STRING) matches any single character that is not in the string.

TAB(N) matches everything from the current cursor position up to cursor position N.

RTAB(N) is similar except that it counts the cursor position from the right end of the string. So, for example, if the subject string has length 10 then TAB(8) and RTAB(2) are identical. RTAB(0) matches everything from the current position to the end of the subject string. A variable REM is initialized to RTAB(0).

POS(N) is a predicate which succeeds and returns the null string only if the pattern cursor is currently in position N. RPOS(N) is similar except that it counts from the right end.

Arrays and Tables

SNOBOL has static heterogeneous arrays and dynamic heterogeneous hash tables. Examples of array definitions are:

```
X = ARRAY(7)
Y = ARRAY(7, 3)
Z = ARRAY('7, 3')
W = ARRAY('-2:5, 0:3, 2', 'X')
```

The first creates an array of seven elements, `X<1>, X<2>, …, X<7>`, using default initialization. The second also initializes the seven elements to 3. The third creates a two-dimensional array with 21 elements with ranges from 1 to 7 and from 1 to 3. The fourth creates a three-dimensional array with 64 elements, all initialized to 'X'.

The arrays are, as discussed above, heterogeneous, and so with the first definition we could, for example, say

```
X<1> = 33
X<2> = 2.67
X<3> = ARRAY(3)
X<4> = 'DENBIGH STARKEY'
```

Tables provide the capability to generate associative arrays/hash tables. This is an incredibly powerful feature that was abandoned by most language designers before Perl made them popular again. The definition has the form

```
T = TABLE()
```

Which creates a table T. Values in T are subscripted through any SNOBOL object. E.g., if HEIGHT is a table then one could say

```
HEIGHT<'DENBIGH'> = "6' 3.5" '"'
```
[3]

The function TABLE can also be given up to two arguments as in

```
HEIGHT = TABLE(20, 5)
```

[3] Note the use of a string delimited by double quotes to contain a single, concatenated to a string delimited by singles which contains a double, to get the string `6' 3.5"`

This sets up an initial table size of 20 elements, and when it overflows tells SNOBOL to always increment by 5. 10 is the default for both arguments. The only reason for this is that if you know a lot of information about the potential table size and the hashing algorithm you can help SNOBOL to implement the table as efficiently as possible.

Since any object can be used as the reference variable, some care has to be taken when using integers. E.g., `T<3 - 2 '' >` and `T<3 - (2 '')>` are different entries because the first becomes `T<'1'>` and the second `T<1>`.

A Complex Pattern Match

Griswold *et al*. have an exercise to determine which lines in the input are palindromes. The obvious solution to this is

```
READ S = TRIM(INPUT)                          :F(END)
     IDENT(S, REVERSE(S))                     :F(NO)
     OUTPUT = S ' IS A PALINDROME'            :(READ)
NO   OUTPUT = S ' IS NOT A PALINDROME'        :(READ)
END
```

Where TRIM removes the trailing blanks from the input, IDENT determines whether or not two strings are equal, and REVERSE reverses a string.[4] Instead of this the solution in the book was:

```
     &FULLSCAN = 1;  &TRIM = 1
     PALIN = LEN(1) $ H @X (*GT(2 * X, SIZE(S))
+         | *RTAB(X) @Y @Z
+         (*H @Z | *EQ(Y, Z) ABORT) FAIL)
READ S = INPUT                                :F(END)
     S PALIN                                  :F(NO)
     OUTPUT = S ' IS A PALINDROME'            :(READ)
NO   OUTPUT = S ' IS NOT A PALINDROME'        :(READ)
END
```

When I first saw this awful solution, my reaction was to (a) panic, and (b) type it in to see whether or nor it worked. It did, and it was also faster than the obvious IDENT/REVERSE solution. To explain what is going on here there are a few new things. Setting &FULLSCAN to 1 means that it will keep attempting all pattern matching paths, even when they appear to be leading nowhere. Setting &TRIM automatically trims all input strings, so S will not have trailing blanks.

Take the pattern PALIN slowly, and assume that S contains a palindrome like ABLE WAS I ERE I SAW ELBA. LEN(1) matches the first character (A) and it is assigned to H with immediate value assignment. The cursor position, 1, is assigned to X. If we aren't past half way through S the GT fails[5] and so we switch to trying the second half of the alternation on lines 2 and 3 of the pattern. The RTAB forces the match to begin X

[4] TRIM, IDENT, and REVERSE are built-in functions

[5] We need the * in front of the GT to force it to use the latest value of X. We have to use the * quite often in this pattern match.

characters from the right hand end (in this case `X` is 1) and we set both `Y` and `Z` to this cursor position (in this case 24). Now we check to see whether the next character matches the first character, which since it is an `A` it does, so we then increment `Z`, in this case to `25`. If it doesn't match then `Y` and `Z` are still equal, so we hit `ABORT`, which fails the whole pattern match. In this case they match, and so we hit `FAIL`, which fails this attempt at the pattern match and tries again starting at the second character. We'll then repeat all this, saving `B` in `H`, going to `RTAB(2)`, finding the matching `B` on the right, and so on through the string until either we reach a non-match and fail with the `ABORT` or we get more than half way through the string, which means that everything has matched, and so we succeed.

Frankly, if you understand all this completely then you have a very good understanding of the details of pattern matching in SNOBOL. I much prefer the `IDENT/REVERSE` solution, but the pattern makes a detailed example of pattern matching in action.

Indirect Reference

There are many features of SNOBOL that I haven't covered in these notes. I'll do one last feature, indirect reference, which uses the unary $ operator. It is easiest to show this with a couple of examples.

Example 1: The statements

```
STATE = 'MONTANA'
$STATE = 'LARGE'
```

Assigns `MONTANA = 'LARGE'`.

Example 2: The input contains the office numbers of a number of faculty members with the format Name:Office number. E.g., my entry might be `DENBIGH:EPS 358`. We want to assign the information to separate variables for each faculty member where the variable name for me would be `DENBIGHOFFICE`. To do this we could use:

```
IN   TRIM(INPUT) BREAK(':') . FAC ':' REM . OFF
+                                          :F(END)
     $(FAC 'OFFICE') = OFF                 :(IN)
END
```

Here I'm trimming off trailing blanks from the input lines, using `BREAK` to get everything up to the colon, matching the colon, and then using `REM` to get the rest of the line for the office.