Modern Programming Languages

Lecture 9-12

An Introduction to SNOBOL Programming Language

SNOBOL stands for StriNg Oriented SymBOlic Language. It is a Special purpose language for string manipulation and handling. It was developed in 1962 at the Bell Labs by Farber, Griswold, and Polensky.

It was created out of the frustrations of working with the languages of that time because they required the developers to write large programs to do a single operation. It was used to do most of the work in designing new software and interpreting different language grammars. Toward the end of the eighties, newer languages were created which could also be used for string manipulation. They include the PERL and AWK. Unlike SNOBOL, Perl and Awk use regular expressions to perform the string operations. Today, roughly forty years after its release, the language is rarely used!

SIMPLE DATA TYPES

Initial versions of SNOBOL only supported Integers and Strings as the basic data types. Real numbers were originally not allowed but were added to the language later on.

Integers

Both positive and negative integers were supported. Following are some examples of integers in SNOBOL.

14   -234   0   0012   +12832   -9395   +0

It was illegal to use a decimal point or comma in a number. Similarly, numbers greater than 32767 (that is, $2^{15} – 1$) could not be used as well. For the negative numbers, the minus sign has to be used without any spaces between the sign and the number. Following are therefore some examples of illegal numbers:

13.4   49723   -   3,076

As mentioned earlier, real numbers were allowed in the later versions of the language and hence 13.4 is a legal number in SNOBOL4.

Strings

The second basic data type supported by the language is a string. A string is represented by a sequence of characters surrounded by quotes (single of double). The maximum length allowed for a string is 5,000 characters. Examples of strings are: “This is a string”, ‘this is also a string’,
“it’s an example of a string that contains a single quote”

SIMPLE OPERATORS

SNOBOL supports the usual arithmetic operators. This includes +, -, *, and /. In addition it also supports the exponentiation operator which is ** or !. The unary – and + are also supported. = operator is used for assignment. The precedence of the operators is as follows: the unary operations are performed first, then exponentiation, then multiplication, followed by division, and finally addition and subtraction. All arithmetic operations are left associative except the exponentiation which is right associative. Therefore, 2 ** 3 ** 2 is evaluated as 2 ** (3 ** 2) which is 512 and not (2 ** 3) ** 2 which would be 64. Like most languages, parentheses can be used to change the order of evaluation of an expression.

Just like C, if both the operands of an operator are integers then the result is also an integer. Therefore, 5 / 2 will be evaluated to be 2 and not 2.5.

VARIABLE NAMES

A variable name must begin with an upper or lower case letter. Unlike C, SNOBOL is case insensitive. That is, the literals Wager and WAGER represent the same variable. If the variable name is more than one character long, the remaining characters may be any combination of letters, numbers, or the characters period '.' and underscore '_'. The name may not be longer than the maximum line length (120 characters). Following are some of the examples of variable names:

```
WAGER       P23    VerbClause   SUM.OF.SQUARES          Verb_Clause
```

SNOBOL4 STATEMENT Structure

A SNOBOL statement has three components. The structure of a statement is as follows:

```
Label  Statement body       :GOTOField
```

Label
The must begin in the first character position of a statement and must start with a letter or number.

The GOTO Field

The goto field is used to alter the flow of control of a SNOBOL program. A goto filed has the following forms:

```
:(label)
:S(label)
:F(label)
:S(label1) F(label2)
```

The first one is an command for unconditional jump to the label specified within parentheses. The second is a command for jump to the specified label if the statement in the body of the statement was executed successfully or resulted in true Boolean value. The third one is opposite to the second and the control jumps to the label if the statement is not executed successfully or resulted in false. The fourth is a combination of the second and the third and states that goto label1 in case of success and label2 in case of failure.

Statement Body

The body can be of any of the following types:

Assignment statement
Pattern matching statement
Replacement statement
End statement

Assignment Statement

Assignment statement has the following form.

```
        variable = value
```

Following are some examples of assignment statement which involve arithmetic expressions.

```
        v = 5
        w.1 = v
        w = 14 + 3  *  -2
        v = 'Dog'
```

It is important to note that the binary operators must have at least one space on both sides. For example 3+2 is not allowed.

It may also be noted that, unlike C, variable are not declared in SNOBOL. The assignment statement creates a variable if it is not already created and assigns a type to it which is the type of the expression in the right hand side of the assignment operator (r-value). So the type of v and w is integer and w.1 is of type real. If a new value is assigned to a variable, it type changes accordingly. For example the type of the variable v is integer and it is now assigned a value of 'Dogs' then its type will change to string. This is known as dynamic typing. That is, the type is determined at run time and changes during execution.

If there is nothing on the right hand side of the assignment statement, then a value of NULL string is assigned to the variable. This is demonstrated in the following example.

```
        This.is.null.string =
```

Another very interesting aspect of SNOBOL is mixing numeric and string values in an expression. If a string can be interpreted as a number then its numeric value will be used in the expression as follows:

```
z = ‘10’
x = 5 * -z + ‘10.6’
```

Now a value of -39.4 will be assigned to x.

Strings Concatenation

SNOBOL was designed mainly for manipulating and handling strings. It has therefore rich string manipulation mechanism. The first one is string concatenation. A *Space* is used as an operator for concatenation as shown below:

```
TYPE = ‘Semi’
OBJECT = TYPE ‘Group’
```

The above set of statements will assign ‘SemiGroup’ to the variable OBJECT. Numbers can also be concatenated with strings, producing interesting results. For example, consider the following code segment:

```
ROW  = ‘K’
NO.    = 22
SEAT = ROW NO.
```

In this case NO. is concatenated with ROW and a value of ‘*K22’* is assigned to SEAT.

The Space operator has a lower precedence than the arithmetic operators and hence we can have arithmetic expressions in string concatenation. For example,

```
SEAT = ROW NO. + 6 / 2
```

will assign the value ‘*K25’* to SEAT if ROW and No. have the values ‘K’ and 22 respectively.

Pattern Matching

Pattern matching and manipulation is another very important feature of SNOBOL. The first statement in this regards is the Pattern Matching Statement. Once again *Space* is used as the pattern matching operator and the statement has the following form:

subject pattern

Note that there is no assignment operator in this case. Both the *subject* and the *pattern* are strings and this statement tries to match the *pattern* in the *subject*. It will be successful if the match is found and will result in failure otherwise. This is demonstrated with the help of the following example:

```
TRADE = ‘PROGRAMMER’
PART    = ‘GRAM’
TRADE PART
```

Now ‘GRAM’ will be searched in ‘PROGRAMMER’. Since it is present in it (‘PROGRAMMER’), it will result in success.

We can use Space for string concatenation as well as for pattern matching in the same statement. In this case, the first space is used as pattern matching and the second one is used as concatenation with concatenation taking precedence over pattern matching. This is shown with the help of the following example:

```
ROW  = ‘K’
NO.    = 2
‘LUCK22’  ROW NO.
```

In this case, NO. is first concatenated with ROW resulting in ‘K2’ which will then be matched in ‘LUCK22’. Once again the match will be successful.

Replacement

Replacement statement is used in conjunction with the pattern matching statement in the following manner.

subject pattern = object

In this case the pattern is searched in the subject and if found it is replaced by the object as demonstrated by the following example:

```
SENTENCE = ‘THIS IS YOUR PEN’
SENTENCE ‘YOUR’ = ‘MY’
```

Since ‘YOUR’ is present in the subject, it will be replaced by ‘MY’, resulting in changing the value of SENTENCE to ‘THIS IS MY PEN’.

If we now have the following statement

```
SENTENCE ‘MY’ =
```

then SENTENCE will be further modified to ‘THIS IS PEN’ as we are now replacing ‘MY’ with a NULL string, effectively deleting ‘MY’ from the subject.

Pattern

There are two type of statements for pattern building. These are Alternation and Concatenation.

Alternation

Vertical bar is used to specify pattern alternation as shown in the example below.

```
P1 | P2
```

This is example of a pattern that will match either P1 or P2.

Here are some more examples:

```
KEYWORD = ‘INT’ | ‘CHAR’
```

This statement assigns the pattern ‘INT’ | ‘CHAR’ to the variable KEYWORD.

```
KEYWORD = KEYWORD | ‘FLOAT’
```

KEYWORD will now get a new value which is ‘INT’ | ‘CHAR’ | ‘FLOAT’. So we can create new pattern by using the assignment statement.

Let us now use the KEYWORD pattern in pattern matching and replacement.

```
TEXT = ‘THIS IS AN INTEGER’
TEXT KEYWORD =
```

Since KEYWORD had the value ‘INT’ | ‘CHAR’ | ‘FLOAT’, that means any one of these strings, that is ‘INT’, ‘CHAR’, or ‘FLOAT’, will match. This matches with INT in ‘THIS IS AN INTEGER’ and will be replaced by the NULL string. So the new value of TEXT is ‘THIS IS AN EGER’.

Concatenation

Two or more patterns can be concatenated to create new patterns. For example we define P1 and P2 as follows:

```
P1 = ‘A’ | ‘B’
P2 = ‘C’ | ‘D’
```

Now if we concatenate these and assign the result to P3 as shown below:

```
P3 = P1 P2
```

This will result in assigning ‘AC’ | ‘AD’ | ‘BC’ | ‘BD’ to P3 which is concatenation of different alternatives of P1 and P2.

Therefore

        “ABCDEFGH” P3

will match at “ABCDEFGH”

Conditional Assignment

At times we want to ensure that assignment occurs ONLY if the entire pattern match is successful. This is achieved by using the '.' (dot) operator as shown below.

pattern . variable

In this case upon successful completion of pattern matching, the substring matched by the pattern is assigned to the variable as the value.

Following is a more elaborative example of this concept.

```
KEYWORD = ('INT' | 'CHAR') . K
TEXT = "INT INT CHAR CHAR INT"
TEXT KEYWORD =
```

In this case, the variable K which was associated with the pattern will get the value INT.

Here is another example.

Let us define the following pattern where BRVAL has been associated to this pattern by the dot operator.

```
BR = ( 'B' | 'R' ) ( 'E' | 'EA' ) ( 'D' | 'DS' ) . BRVAL
```

Associates variable BRVAL with the pattern BR

It may be noted that the pattern BR is created by concatenating three different alternations as shown below.

```
BR = ( 'B' | 'R' ) ( 'E' | 'EA' ) ( 'D' | 'DS' )
```

Note that it is equivalent to

```
          BR = 'BED' | 'BEDS' | 'BEAD' | 'BEADS' | 'RED' | 'REDS' | 'READ' |
+                'READS'
```

This '+' in the first column of a line states that this is continuation of the last line and not a new statement. Anyway, let us come back to our example.

On successful completion of matching, the entire substring matched will be assigned as value of the BRVAL

So

```
	'BREADS' BR
```

will assign 'READS' to BRVAL.

Let us now define the BR pattern as shown below and also associate FIRST, SECOND, and THIRD to the respective constituent sub-patterns of this larger pattern.

```
	 BR = ( ( 'B' | 'R' ) . FIRST ( 'E' | 'EA' ) . SECOND
+			( 'D' | 'DS' ) . THIRD) . BRVAL
```

On successful completion of matching, the entire substring matched will be assigned as value of the BRVAL. In addition B or R will become the value of FIRST, E or EA will be assigned to SECOND, and THIRD will get either D or DS.

So

```
	'BREADS' BR
```

will result in the following assignments.

FIRST – 'R' , SECOND – 'EA', THIRD – 'DS'
BRVAL – 'READS'

Immediate Value Assignment

The ‘.’ (dot) operator is used for conditional assignment only when the entire pattern is matched.

The $ is used for immediate value assignment even if the entire pattern does not match. It is used as follows:

```
                Pattern        $        Variable
```

In this case whenever pattern matches a substring, the substring immediately becomes the new value of the Variable. This is demonstrated with the help of the following example.

```
         BR = ( ( ‘B’ | ‘R’ ) $ FIRST ( ‘E’ | ‘EA’ ) $ SECOND
+                    ( ‘D’ | ‘DS’ ) $ THIRD) . BRVAL
```

Value assignment is done for those parts of the pattern that match, even if the overall match failed.

So the following statement

```
        ‘BREAD’ BR
```

will result in the following matches.

FIRST – B, FIRST – R, SECOND – E, SECOND – EA, THIRD – D,
BRVAL - READ

Note that FIRST and SECOND get a value as soon as a match is made and hence are assigned values more than once. Of course only the last value will be the value of these variables at the end of entire match.

Let us now try make the following match.

```
        ‘BEATS’ BR
```

In this case the statement fails as the whole pattern is not matched but parts of it are matched and hence get the values as shown below.

FIRST – ‘B’ , SECOND – ‘E’, SECOND – ‘EA’

Input and Output

The I/O is accomplished through INPUT and OUTPUT statements. We will look at examples of I/O in the next sections.

Control Flow

Control flow is achieved through the ‘go to field’ in the statement body as shown in the following examples.

The first example simply keeps on reading the entire line from the INPUT and stops when there is end of input.

```
LOOP        PUNCH = INPUT    :S(LOOP)
END
```

The next example reads a line and prints it on the output and then goes back to read the next lines again.

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

Here is a more interesting example that uses control flow. In this example it continues to delete a matched pattern in the subject and stops when no match is found.

```
            TEXT = “REDPURPLEYELLOWBLUE”

            COLOR = ‘RED’ | ‘BLUE’ | ‘GREEN’
BASIC       TEXT   COLOR =   :S(BASIC) F(OTHER)
OTHER
```

In the following example all the numbers for 1 to 50 are added and the result is stored in SUM and is printed on the output.

```
            SUM = 0
            N     = 0
ADD         N = LT(N, 50) N + 1         :F(DONE)
            SUM = SUM + N               :(ADD)
DONE        OUTPUT = SUM
```

Indirect Reference

Indirect reference is also an interesting feature of SNOBOL. It is similar to a pointer in concept but there are certain differences as well. These differences will be highlighted in the following code segments. Let us first look at the syntax:

The unary operator ‘$’ is used for indirect reference. Note that ‘$’ has been overloaded in this case. This says that now instead of using the operand as the variable, use its value as the variable. The rest will remain the same. Here is an example that elaborates this concept:

```
MONTH = ‘APRIL’
$MONTH = ‘FOOL’
```

MONTH is given the value ‘APRIL’ 1. In statement 2, indirect reference is used. Here, instead of MONTH, its value will be used as the variable. So it is equivalent to saying

```
APRIL = ‘FOOL’
```

Here is another example that uses indirect reference in a more interesting manner:

```
RUN = 10
WORD = ‘RUN’
$(WORD) = $(WORD) + 1
```

Because of indirect reference, it increments the value of RUN and not WORD.

The following example of indirect reference uses it in the go to field.

```
N = 3
N = N + 1      :$( ‘LABEL’ N)
```

This results in making the control jump to LABEL4

Functions
SNOBOL 4 supports two types of functions: (a) built-in function which are known as primitive functions and (b) user defined functions.

Primitive Functions

There are a number of primitive functions but we shall look at only a few. These include SIZE and REPLACE functions. The SIZE function returns the size of a string and the REPLACE function is used to replace one character with another in the entire string. Here are some examples of these functions:

```
SIZE(TEXT)

PART1 = “NUT”
OUTPUT = SIZE(PART1)
```

Since the string stored in PART1 is ‘NUT’, it has three characters and hence the result is 3.

```
PART2 = “BOLT”
N = 64
OUTPUT = SIZE(‘PART’ N + 36)
```

In this case, PART will be concatenated with 100 and will generate the string PART100. The output will thus display 7.

REPLACE function has the following syntax:

```
REPLACE(STRING, ST1, ST2)
```

In this case all occurrences of ST1 in STRING will be replaced by ST2. ST1 and ST2 must have the same number of characters. The corresponding characters of ST1 will be replaced by characters corresponding to them in ST2. This is shown in the following example.

```
TEXT = “A(I,J) = A(I,J) + 3”
OUTPUT = REPLACE(TEXT , ’()’ , ‘<>’)
```

The output will thus display:

```
A<I,J> = A<I,J> + 3
```

Arrays
SNOBOL4 has array data type just like other languages. However, unlike most languages, data stored in an array is not necessarily of the same type. That is, each element can be of different data type.

The following statement creates and assigns to V a one-dimensional array of 10 elements, each assigned to the real value of 1.0.

```
        V = ARRAY(10, 1.0)
```

The next statement creates a one-dimensional array X of 8 elements with the lower index being 2 and the upper index being 9. As there is no initial value given, each cell is initialized to NULL string.

```
        X = ARRAY('2:9')
```

The following statement creates a two dimensional array of 3 x 5 and each cell is initialized to NULL string.

```
        N = ARRAY('3,5')
```

The size of the array can be determined at run time by using input from the system of using a variable. Following example shows this concept:

```
        A = ARRAY(INPUT)
```

In this case size and initial value of the array is determined by the input.

Following example creates an array whose size is determined at run time. It then stores values in it from the INPUT. It is important to note here that when the array index 'I' gets a value, as the result of increment in the second last statement, greater than the value of the last index of the array, the statement with Label MORE fails and the control goes to the Label GO.

Arrays
```
                &TRIM = 1
                I = 1
                ST = ARRAY(INPUT)
MORE            ST<I> = INPUT           :F(GO)
                I = I + 1               :(MORE)
GO
```

The following example uses arrays in an interesting manner:

```
        &TRIM = 1
        WORDPAT = BREAK(&LCASE &UCASE) SPAN(&LCASE &UCASE "'-") . WORD
                COUNT = ARRAY('3:9',0)
READ            LINE = INPUT                                    :F(DONE)
NEXTW           LINE WORDPAT =                                  :F(READ)
                COUNT<SIZE(WORD)> = COUNT<SIZE(WORD)>+ 1        :(NEXTW)
DONE            OUTPUT = "WORD LENGTH NUMBER OF OCCURRENCES"
                 I = 2
PRINT           I = I + 1
                OUTPUT = LPAD(I,5) LPAD(COUNT<I>,20)            :S(PRINT)
END
```

The first statement simply deletes leading spaces from the input.

In the second statement, a pattern WORDPAT is defined. It uses tow primitive functions BREAK and SPAN. BREAK takes the cursor to the first lowercase or uppercase character in the subject. The SPAN provides a mechanism to keep moving as long as we get uppercase, lowercase, or a hyphen in the subject string. So this pattern scans the subject string for complete words. Since we have associated variable WORD with this pattern, the matched value will be stored in it.

In the third statement an array COUNT is created whose first index is 3 and last index is 9. All elements are initialized to 0.

Fourth statement reads a line from the input. At the end of the input, the control the statement fails and the control goes to DONE.

In the fifth statement WORDPAT is matched in the LINE and match is replaced by the NULL string. The matched value is also stored in WORD. When there is no match found, the statement fails and the control goes to READ to read the next line from the input.

In the sixth statement, the size of the word is used as the array index and the value is incremented at that index. If it is less than 3 or greater than 9 then nothing happen. In fact the statement fails in this case but since there is unconditional jump, it goes back to read the next word from the input.

The rest of the statements are used to display the values at each array index. That is, this program simply counts the occurrences of words of lengths between 3 and 9 in the input and displays that number.

Tables
A table data structure maps pairs of associated data objects. Tables have varying lengths and are dynamically extended if necessary.

```
T = TABLE()
```

creates a table of unspecified length.

```
T = TABLE(N)
```

creates a table that initially has room for N elements

Once created, table can be used in a manner similar to arrays. The only difference is that they are indexed by the key value and not the index number. For example:

```
T<‘A’> = 1
```

Associates key ‘A’ with value 1.

SNOBOL Features

- Dynamic typing
- Mixing arithmetic and string operations
- String operations including concatenation
- GOTO control structure
- Overloaded operators
- Space as an operator
- Run-time compilation
- code can be embedded in data, allowing easy run-time extension of programs
- Variable length string
- Array tables and record type objects
- Absence of declaration
- Operator overloading.

Issues

Developers of SNOBOL had no or limited experience in software development. Griswold did not have any prior computing experience. Polonsky had also limited software development skills. This led to a number of issues:

- Crucial aspects of development process, such as documentation, were overlooked.
- The lack of formal documentation led the initial version of SNOBOL to minimal use.
- Another design problem was that developers treated language semantics casually. Such a poor practice hindered other programmers from understanding the meaning of the program.
- Choice of operators
- Poor readability and writability.
- Dynamic typing
- Mixing arithmetic and string operations
- String operations including concatenation
- GOTO control structure
- Overloaded operators. For example, the use of a blank as a concatenation operator caused confusions in coding and maintenance. Users had to be bewared of the proper use of the blank space operator, and not to confuse it with other operators.

To top it all the compiler was not sophisticated. Any string of characters was a legal SNOBOL program, and the compiler would accept everything as long as it was a string. The legend says that Griswold accidentally gave a non-SNOBOL program to the SNOBOL compiler, and the program was successfully compiled!