Programming Languages Homework 1

# LEXICAL ANALYZER FOR THE STRAWMAN PROGRAMMING LANGUAGE

## Due date: 4.8.2013 (April 8th)

## *Objectives*

1. Write a lexical analyzer for the STRAWMAN language which is defined below.
2. Write the BNF grammar for the lexemes used in the STRAWMAN language.

## *The STRAWMAN Programming Language*

STRAWMAN (STring Reading And Writing and MANipulation language) is aprogramming language whose basic data structure is string. A Strawman script is composed of just a sequence of lines. There is no block structure. The whole code is also stored in one file that has a ***.stw*** extension. Files with this extension are regarded as executable.

**Line of code:**

One line of code is either:

An assignment

A read command

A write command

Strawman lines end with a semicolon. The textual end-of line has no significance. One *Strawman line* may be split into multiple text lines; likewise, one text line may contain multiple *lines of code*. Strawman lines may contain blank characters between the lexemes. It is also possible to write comments between `/*` and `*/` marks.

**Assignment:**

An assignment has the form: `<identifier> := <String_expression>;`

Names (identifiers) are associated with the strings in order to provide a way of referring to particular strings. The name of a string may contain any sequence of alphanumeric characters but must start with an alphabetical character. Identifiers must be at most 20 chars long. Identifiers are case sensitive. (so are keywords)

**Ex**: `STR1 := "To be or not to be!";`

The assignment above creates a variable called STR1 and initializes it with the value "To be or not to be!".

Variables do not require a declaration and they are implicitly declared the first time they are used.

**String Expression:**

"To be or not to be!" is a string literal (string constant). String literals are written between double quotations and are not allowed to contain any double quotations. Although in this example a basic string literal (constant) has been used as the assignment value, it is possible to write a complex expression for this purpose. Complex expressions are written by combining the basic string operations.

The basic operations of this language **are *concatenation*, *trimming, replacement*** *and* ***truncation***.

**String Concatenation:** Strings can also be formed by concatenation. Thus the statement:

```
STR1 := "To be" + " or " + "not to
                    be!";
```

produces the same result as the preceding example.

Strings which have been named previously can be used to form new strings. For example, the statement STRX = STR1; forms a string named STRX with the same contents as the string named STR1.

Both literals and named strings can be used in formation. The sequence of statements

```
STR1 := "To be or not to be!";

STR2 := "This is the problem!"; TEXT= STR1+ " "+

        STR2+ "isn't it?";
```

will form a composite string.

**String Trimming:** String trimming removes all leading and trailing occurrences of a set of specified characters. The first operator defines the string to be trimmed. The second string contains a list of characters that will be trimmed from the two sides of the string. Thus the following statement:

```
STR1 := "*** To be or not to be!**  **" / "*";

/* produces the same result as the line below */

STR1 := "To be or not to be!**";
```

Trimming is particularly useful for removing unnecessary blanks.

**String Replacement:** Another operation permitting alteration of the contents of a string is replacement. Suppose in the string STR2 we wished to replace "problem" by "question". The following statement will accomplish this

```
STRX := STR2:"problem" < "question";
```

This statement scans STR2 for an occurence of "problem". If this scan is successful, "problem" is then replaced by "question". Thus STRX would become "This is the question!". If the scan fails, STRX will be equal to STR2.

Using replacement, deletion may also be done. Writing an empty string ("") to the right of the less-than sign (<) causes the substring found by the scan to be deleted. Thus

```
STRX := STR2:"problem" < "";
```

would delete "problem" from STR2 and assign the result to STRX.

**Ex:**

```
BaseString:="Strawman is a new language";

Message:=BaseString:"a new"<"the"

      +" of future!";

/* Now the variable Message contains:

 "Strawman is the language of  future!" */
```

**String Truncation:** String truncation operator, searches for a substring inside the base string and if it finds an occurrence, it truncates the base string from the start of that occurrence of the substring. The syntax of the operation is: `<BaseString>:/<substr>`

**Ex:**

```
STR1:="Smile,this is the first day of the last";

STR2:=STR1:/"the";
/* Now STR2 has the value: "Smile,this is " */
```

**Parenthesis and Precedence:** Strawman operations are performed from left to right without and operator precedence rules. However the programmer may manipulate the operation order by using parenthesis.

```
STR1:= "Smile,this is the first day of the last";
myString:=((STR1:"is"<"can be"):/"day")+"or the last";

/* This script results in the variable myString being assigned the value

   "Smile,this can be the first or the last" */
```

**Read Command:**

Read command is used to read a string value from a file or from the keyboard. It has the format:

```
 READ <Identifier> [FROM <filename>];
```

**Ex:**

```
  READ myString FROM "input.txt";

    /* note keywords are written with capitals*/
  READ  myString;
```

**Write Command:**

Write command is used to write a string value to a file or to the screen.

```
WRITE <String_expression> [TO <filename>];
```

**Ex:**

```
myDataString:="Hello world.";

WRITE myDataString + " Oh What a cliché!" TO "output.txt";

   /*Written to file*/

WRITE "Message successfully written!" /*written to screen*/ ;
```

**Output Example for the Lexical Analyzer:**

**Code line:** `Message:=BaseString:"a new"<"the"+" of future!";`

**Output:** identifier("Message"), assignmentOperator, identifier("BaseString"), replacementOperatorColon, stringConstant("a new"),replacementOperatorLessThanSign, stringConstant("the"), concatOperator, stringConstant(" of the future!"), endOfLine

**Error Detection:**

Catch all the necessary syntax errors and issue suitable error messages. This means your lexical analyzer should have the ability to:

- detect all and only errors,
- detect multiple errors at once,
- give accurate detection and indication of the error type,
- give accurate detection and indication of the error position.

**Error Examples:**

```
WRITE "myString ; (non-terminated constant)

aString:="MyString"; /*This is an assignment expression     (non-terminated
comment)
```

## Implementation and Evaluation

The program should be implemented in C by teams of at most 3 people from the same lab group. Late submissions will be subject to a penalty of 10% decrease in grade for each day.

The teams are expected to hand

- a hardcopy of the source code and lexeme grammar to your lab instructor, and
- an electronic submission of source and executable files of the lexical analyzer together with the soft copy of the lexeme grammar on moodle.

Good luck.