CS39003 Compilers Laboratory
Autumn 2025
Assignment 1
Date of posting: 04-Aug-2025

Programming Using Lex

This assignment is an extension of Assignment 0. You write an interpreter on some string operations. This time, you need to use lex in order to identify the tokens. Do not use yacc for the purpose of parsing. Your lex actions will perform this task.

Format of the input file

The input to the interpreter is stored in a text file. Lex will read the lines one by one from the input file, identify the tokens in each line, and do the necessary string operations. Each line consists of an assignment of the following form.

> *l-value* = *r-value*

Here, *l-value* is a valid name of a string variable. Follow the same naming conventions as in C.

The *r-value* is the concatenation of one or more arguments. The concatenation operator is dot (`.`). Each argument starts with a literal string (not quoted) or a reference to a string variable (starting with `$`). All strings are over the Roman alphabet (both lower- and upper-case letters are allowed, and nothing else). This is followed by an optional range selector (this was not in Assignment 0) and then by an optional exponent. The range selector is delimited by square brackets. The following options are allowed. Indexing of strings in our language is zero-based.

`[r, s]`	The substring starting at index `r` and ending at index `s`.
`[t]`	The string consisting only of the character at index `t`. This is same as `[t, t]`.
`[<l]`	The prefix of length `l`, same as `[0, l-1]`.
`[>l]`	The suffix of length `l`, same as `[n-l, n-1]`, where `n` is the length of the string.

Here, `r`, `s`, `t`, and `l` must be non-negative integers. If `r > s`, or if one (or both) of these indices is (are) out of the range of indices in the string, take the empty string as the substring. Moreover, if `l` is larger than the length `n` of the string, then take the entire string as the prefix or suffix of length `l`. Negative integers are not allowed in range specifications.

The exponent is as in Assignment 0, namely it starts with the exponentiation operator `^`, and is followed by a non-negative integer.

Some examples follow. Let the string variable `S` store the string *abcdefgh*. Then, `$S[1,4]` is *bcde*, `$S[5]` is *f*, `$S[<5]` is *abcde*, and `$S[>5]` is *defgh*. Moreover, `$S[5,0]`, `$S[5,10]`, and `$S[10]` are empty strings, whereas `$S[<10]` and `$S[>10]` are the entire string *abcdefgh*. Finally, `$S[1,2] ^ 5 . uvwxyz[>3] ^ 2` is *bcbcbcbcbcxyzxyz*. Undefined string references are to be replaced by the empty string (after printing an error message).

The tokens

An entire line (for that matter, the entire input file) can be specified by a regular expression. Lex must not use that regular expression. Doing that can only indicate whether a line (or the entire input) is syntactically valid. We instead need the indivisible parts of the input, for carrying out the string computations. These should be the tokens to be identified by lex. Here is a complete list of tokens that your lex program should identify. Allow spaces and tabs before or after any token.

l-value	Should be a valid name (as per the naming convention of C)
Alphabetic string	A literal string in the *r-value*
`$name`	Reference to a string, where `name` should follow the C naming convention
`[`	Beginning of a range selector
`]`	End of a range selector
`,`	Separator in a range selector
`<`	Prefix indicator in a range selector
`>`	Suffix indicator in a range selector
`^`	The exponentiation operator
Integer	A non-negative integer (0 allowed) that can appear in a range selector or as an exponent
`=`	Assignment operator

Note that `[5,6]`, `[<5]`, `[>6]`, and `^5` are not single tokens.

Lex states

Use Lex states to make your task easier. Note that it is valid to have the following line as a valid assignment.

```
ABC = ABC
```

Here, `ABC` on the left side is a valid name, whereas `ABC` on the right side is a valid literal string. Both the patterns for valid names and for valid strings match `ABC`. You need to distinguish them (and do other things) using lex states.

At the beginning of each line, lex is in the INITIAL state. After reading the variable name and the assignment operator, lex jumps to a state RVALUE. Encountering `[` in state RVALUE will let lex enter a state RANGESELECTOR. Encountering `]` in the state RANGESELECTOR will let lex come back to the state RVALUE. Likewise, the exponentiation operator `^` changes the state from RVALUE to EXPONENT. After reading an integer in the EXPONENT state, lex reverts back to the state RVALUE. The new-line character marks the entry of lex in the INITIAL state. Inside the RANGESELECTOR state, you may use additional states (like RANGEEND, PREFIX, SUFFIX) depending on the tokens that lex identifies in this state.

Error detection

As in Assignment 0, detect the errors "*Invalid l-value*" and "*Invalid reference*." This time, lex does pattern matching for you, so you can detect some other types of errors. Handle the following two types of errors.

- Literal string containing any non-alphabetic character (including spaces)
- Any symbol that is illegal in a state (like negative integers in ranges or exponents, `.` or `^` in INITIAL state, and so on)

If an error is detected by lex, it jumps to a state ERROR. In this state, the rest of the line is read but ignored, and processing the entire line is skipped.

Symbol table

As in Assignment 0, maintain a symbol table of (*name*, *value*) pairs. A reference to a variable in *r-value* will initiate a symbol-table lookup. As assignment to *name* supplied as an *l-value* replaces the *value* if *name* is already present in the symbol table. Otherwise, a new entry is to be created in the symbol table.

Reading from a file

Allow lex to read from a file. That is, both the following should be supported.

```
$ ./a.out < input.txt
$ ./a.out input.txt
```

Multiple input files are not to be handled in this assignment.

Submit a single lex file. The user-code section should have the `main()` function (and `yywrap()`, of course). The `main()` function should set `yyin` (if needed), and call `yylex()`. It will do nothing else. The actions against the patterns will do all the job of the interpreter (and will not return token types). Lex generates a C file `lex.yy.c`. Any C++ compiler can compile any C program (because C is a subset of C++). Inserting C++ constructs in the lex actions should not be a problem, because lex copies the actions verbatim to the function `yylex()`. If you face a problem, use C syntax only.

Sample Output

Input (without errors)

```
S1 = abcdefghijkl . mnopqrst . uvwxyz
S2 = $S1[10] ^ 20
S2 = $S1[<6].$S1[8,10].$S1[>6]^3
      S3      =       $S1 [ < 6 ] . $S1 [ 8 , 10 ] . $S1 [ > 6 ] ^ 3
S4 = abcdefghijklmnopqrstuvwxyz [ < 6 ] . abcdefghijklmnopqrstuvwxyz [ 8 , 10] . abcdefghijklmnopqrstuvwxyz [ > 6 ] ^ 3
```

Output

```
+++ Line 1 processed: S1 is set to "abcdefghijklmnopqrstuvwxyz"
+++ Line 2 processed: S2 is set to "kkkkkkkkkkkkkkkkkkkk"
+++ Line 3 processed: S2 is set to "abcdefijkuvwxyzuvwxyzuvwxyz"
+++ Line 4 processed: S3 is set to "abcdefijkuvwxyzuvwxyzuvwxyz"
+++ Line 5 processed: S4 is set to "abcdefijkuvwxyzuvwxyzuvwxyz"
```

Input (with errors)

Here is an input file that contains various types of errors in the input. Try to detect as many of these types of errors as possible. Note that a reference to an undefined string is recovered by substituting the empty string for that reference, whereas a prefix or suffix length larger than the length of the string is handled by substituting the entire string. All other errors are handled by stopping processing of that line.

```
E .= abc
E = F = abc ^ 2
123E = uvw . xyz
K L = ijklmn
E = abc123 ^ 2
= abc
L
L =
E = $abc.def^2
E = abc def^2
E = abc[>]
E = abc ^ -2
E = [9]
E = abc[<4] ^ 2 xyz[>2] ^ 4
E = abc[<4] ^ 2 . xyz[>2] ^ 4
F = $E[2,3][4,5]
F = $E ^ 4 ^ 5
F = $E ^ 4 [2,3]
```

Output (error handling)

```
    *** Invalid character '.' found. Line 1 cannot be processed
    *** Invalid character '=' found. Line 2 cannot be processed
    *** Invalid character '1' found. Line 3 cannot be processed
    *** Invalid lvalue: Line 4 cannot be processed
    *** Invalid character '1' found. Line 5 cannot be processed
    *** Invalid lvalue: Line 6 cannot be processed
    *** Assignment operator missing: Line 7 cannot be processed
    *** No r-value: Line 8 cannot be processed
    *** Reference to undefined variable "abc"
+++ Line 9 processed: E is set to "defdef"
    *** Invalid string "def" in rvalue: Line 10 cannot be processed
    *** Invalid character ']' found. Line 11 cannot be processed
    *** Invalid character '-' found. Line 12 cannot be processed
    *** No string argument: Line 13 cannot be processed
    *** Invalid string "xyz" in rvalue: Line 14 cannot be processed
+++ Line 15 processed: E is set to "abcabcyzyzyzyz"
    *** Invalid range selector: Line 16 cannot be processed
    *** Invalid exponent specifier: Line 17 cannot be processed
    *** Invalid range selector: Line 18 cannot be processed
```