CS39003 Compilers Laboratory
Autumn 2025
Assignment 4
Date of posting: 08-Sep-2025

Using yacc

Let us consider strings again. As in earlier assignments, we consider only alphabetic strings (both lower- and upper-case letters are allowed, but nothing else). Literal strings will not be quoted. Named strings are to be accessed by the `$` symbol. All the string operations that we plan to support are listed below.

`+` String concatenation operator. Example: `Computer + Science` is `ComputerScience`.

`-` This is a new operation introduced in this assignment. For two strings *s* and *t*, the string *s* – *t* is obtained by removing from *s* the longest possible prefix of *t* appearing in *s* as a subsequence (not as a substring). Each character of *s* in the matched subsequence is chosen as left as possible (but in the same order as in *t*). Here are some examples.

$\underline{\texttt{co}}\texttt{m}\underline{\texttt{p}}\texttt{u}\underline{\texttt{ter}}\texttt{ - }\underline{\texttt{copter}}\texttt{ = mu}$
$\underline{\texttt{co}}\texttt{pter - }\underline{\texttt{co}}\texttt{mputer = pter}$
$\texttt{de}\underline{\texttt{b}}\texttt{itc}\underline{\texttt{a}}\texttt{r}\underline{\texttt{d}}\texttt{ - }\underline{\texttt{bad}}\texttt{credit = deitcr}$
$\texttt{Computer - computer = Computer}$
$\underline{\texttt{compute}}\texttt{ - }\underline{\texttt{computer}}\texttt{s = }\varepsilon$
$\texttt{a}\underline{\texttt{b}}\texttt{r}\underline{\texttt{a}}\texttt{cadab}\underline{\texttt{r}}\texttt{a - }\underline{\texttt{bar}}\texttt{foo = arcadaba}$

This operator is not associative. For example, `abcd - (abd - b) = bc`, whereas `(abcd - abd) - b = c`. Let us impose the convention of left-to-right associativity for the operator, that is, *r* – *s* – *t* will be interpreted as (*r* – *s*) – *t*.

`^` This is the exponentiation operator. Example: `abc^5 = abcabcabcabcabc`.

`[m]` The *m*-th character of a string. Example: `abcdef[3]` is the string `d`, and `abcdef[7]` is the empty string. We use zero-based indexing.

`[m,n]` The substring from index *m* to index *n*. Example: `abcdef[2,4] = cde`, `abcdef[2,8] = cdef`, `abcdef[4,2] = ε`.

`[<m]` Prefix of length *m*. Example: `abcdef[<3] = abc`, `abcdef[<8] = abcdef`, `abcdef[<0] = ε`.

`[>m]` Suffix of length *m*. Example: `abcdef[>3] = def`, `abcdef[>8] = abcdef`, `abcdef[>0] = ε`.

`( )` Parentheses are used for grouping expressions.

The operators `+` and `-` are at the same precedence level which is lower than the precedence of the exponentiation and the range-selection operators. All the operations at the same precedence level should be carried out in a left-to-right associative manner. For example, the expression `$s-$t[>10]^3[4,12]^2+abc` should be interpreted as `($s-(((($t[>10])^3)[4,12])^2))+abc`. Here are some other examples of string expressions. Also see the Sample Run section.

```
(abc^3 + d^5)[3,10] - (c + d^3)^2 = ababc
(abc + d)^3[3,10] - (c + d^3)^2 = dababc
(abc + d)^3[3,10]^2 - (c + d^3)^2 = dababcabcab
ab^4^3 - b^2^6 = aaaaaaaaaaaa
abcdefgh[>6][<2] = cd
```

The input file consists of several lines. Each line is an assignment of the form *var* = *expr*, where *var* is a C-type valid variable name, and *expr* is a string expression. The grammar for an input file follows. The terminal symbols (tokens) are shown in red.

PROG	→	LINE \n | LINE \n PROG
LINE	→	id = EXPR
EXPR	→	EXPR + TERM | EXPR – TERM | TERM
TERM	→	TERM [RANGE] | TERM ^ XPNT | BASE
BASE	→	str | $id | (EXPR)
RANGE	→	num | num , num | < num | > num
XPNT	→	num

In this assignment, you implement this grammar using a lex file and a yacc file. You may write the functions for string operations in the yacc file itself or in a separate source file.

Lex file

Write a lex file to identify all the tokens from the input file. The tokens for which the values are relevant are id (variable name), $id (variable reference), str (literal string), and num (a non-negative integer). Other tokens to be recognized are the assignment operator (=), string concatenation (+), subtraction (–), and exponentiation (^) operators, the punctuation symbols used for range selection ([,], comma , < , and >), parentheses (and) used for grouping, and the end-of-input marker \n. Use the state RVALUE.

Yacc file

The start symbol is PROG. The grammar already takes care of the precedence and the associativity of the string operators, so you do not need to specify these in your yacc file. Note however that yacc's stack will consist of elements of multiple types. XPNT and num are non-negative integers, whereas the tokens str and $id and most other non-terminals are strings. You also need a custom-designed structure to store a RANGE (do **not** store it as a string). This implies that you need a `%union` directive for yacc. Moreover, `%token` and `%type` directives are also necessary to set the types of the terminal and the non-terminal symbols.

In the second section, write the productions along with short actions that will call the relevant string functions. For example, the production EXPR → EXPR + TERM will call a string concatenation function on `$1` and `$3`. The function should return a string that will set the value of `$$`. Yacc supports only basic C data types (like `int`, `char *`, `struct range *`), not even `typedef`-ed names. Use only these data types in the union. In particular, C++ types cannot be used.

Do **not** implement the string operations inside the body of the yacc actions. Implement the string functions in the third section (or in a separate source file). Yacc actions will only call these functions.

Symbol table

Maintain a symbol table of defined strings. An assignment statement in the input creates a new entry in the table (if that name is not present in the table) or overwrites the old value (if that name already resides in the table).

Error Handling

- Lex detects unrecognized symbol (print a warning message, and ignore)
- An undefined string is referenced (use the empty string after printing a warning message)
- Parse error (write `yyerror()` that will print an error message, and exit from the program)

What to submit

Write a `makefile` with compile, run (on `input.txt` without `stdin` redirection), and clean targets. Pack your lex file, your yacc file, and your `makefile` (and any other source file that you may write) in a single zip/tar/tgz archive. Submit that single archive.

Sample Run

input.txt

```
s01 = computer - copter + empty - emu
s01 = computer - (copter + empty - emu)
s01 = computer - copter - empty + emu

s02 = abc^9 + d^9 - c^9
s03 = (abc^9 + d^9)[6,14] - c^9
s04 = (abc^9 + d^9)[<15] - c^9
s05 = (abc^9 + d^9)[>15] - c^9
s06 = $s02^3 - $s05 - $s05 - $s03^10
s07 = $s1 + ab^5

s08 = (abc^3 + d^5)[3,10] - (c + d^3)^2
s09 = (abc + d)^3[3,10] - (c + d^3)^2
s10 = (abc + d)^3[3,10]^2 - (c + d^3)^2
s11 = ab^4^3-b^2^6

s12 = abcdEFGHijklMNOPqrstUVWXyz[<12][>6]
s12 = abcdEFGHijklMNOPqrstUVWXyz[>12][<6]
```

Output

```
    Stored s01 = mupty
    Stored s01 = mue
    Stored s01 = muemu

    Stored s02 = abababababababababddddddddd
    Stored s03 = ababab
    Stored s04 = ababababab
    Stored s05 = ababddddddddd
    Stored s06 = ddddddddd
*** Undefined string s1
    Stored s07 = ababababab

    Stored s08 = ababc
    Stored s09 = dababc
    Stored s10 = dababcabcab
    Stored s11 = aaaaaaaaaaaa

    Stored s12 = GHijkl
    Stored s12 = OPqrst
```