Programming Assignment #1

CS 6353 Section 1, Spring 2016

. The requirements for your submission are listed in sections 1 (Program Specification) and 2 (non-functional requirements). *Follow the requirements carefully and precisely – they will be thoroughly tested.*

Section 3 provides a few example outputs. Section 4 provides basic information on accessing and using the reference environment.

I grade by running your program against a large test suite, which includes many examples that are valid, as well as some samples that are invalid. You will be graded by the percentage of test cases you get correct. I recommend you test thoroughly against good output to make sure your basic functionality is right; also test thoroughly against malformed output.

1. Program Specification

1. Your program must read 8-bit ASCII strings from standard input -- for instance, using the cin object in C++, or stdin in C. You must consume all input from standard input.
2. Your must tokenize the entire input stream, using the lexical specification below to dictate how to break the stream into tokens (using a "maximal munch" strategy).

Lexical Specification:

```
letter          -> [a-zA-Z]
digit           -> [0-9]
newline         -> \n
for             -> for
while           -> while
if              -> if
else            -> else
identifier      -> (letter|_)(letter|digit|_)*
integer         -> digit+
float           -> (digit+\.digit*)|(\.digit+)
string          -> "[^"\n]*"
whitespace      -> (' ' | \t)+
comment         -> #.*
operator        ->  '!' | '%' | '&' | '|' | '+' | '-' |
'*'
                    | '/' | '{' | '}' | '[' | ']' | ';' |
','
                    | '<' | '>' | '=' | '<=' | '>=' | '!='
                    | ':='
```

3. You must then output tokens to standard output (e.g., via cout in C++ or stdout in C), using the token information chart below to dictate what information you print about each token. Note that the lexical pattern, when italicized, refers to the pattern from the lexical specification above.

Lexical Pattern	Numeric Type	English Type	Value to output
identifier	1	ID	Exact match; exactly what the user input
string	2	STR	The user's input minus the quotes
integer	3	INT	Exact match
float	4	FLOAT	Exact match
whitespace	5	WS	None
newline	6	NEWLINE	None
comment	7	COMMENT	Exact match
for	8	FOR	Exact match
while	9	WHILE	Exact match
if	10	IF	Exact match
else	11	ELSE	Exact match
'!'	12	!	None
'%'	13	%	None
'&'	14	&	None
'|'	15	|	None
'+'	16	+	None
'-'	17	-	None
'*'	18	*	None
'/'	19	/	None
'{'	20	{	None
'}'	21	}	None
'['	22	[	None
']'	23	]	None
';'	24	;	None
','	25	,	None
'<'	26	<	None
'>'	27	>	None
'='	28	=	None
'<='	29	<=	None
'>='	30	>=	None
'!='	31	!=	None
':='	32	:=	None
n/a	97	ERR1	None
n/a	98	ERR2	None
n/a	99	ERR3	None
n/a	100	ERR4	None

1. Note that there's no "in between tokens"—every bit of input should map to a token (even if it's an ERR token… the specs for which are in English below). You are looking at the front of the input, finding the first token per the spec (using maximal munch), then looking at the rest of the input and repeating.
2. Remember that, in a regular expression, a dot matches any character in the alphabet EXCEPT newline. It does NOT match characters that aren't in the alphabet. Those need to be handled with an ERR4 (see below).
3. If the input contains unterminated strings (which include strings that contain characters not in the alphabet—anything that starts with a " and doesn't meet the spec for a string), then instead of generating a string token, generate a single ERR2 token. The position indicator for the token should correspond to the beginning quote starting the string. Consume all input up to (but not including) the first newline. If there is no next newline, consume all remaining input. The length associated with the token should be reported appropriately.
4. The alphabet for this assignment consists of the following:
    a. ASCII 0x09 and 0x0a (tab and newline)
    b. ASCII 0x20 through 0x7e (all printable ASCII characters).
5. If you are in the middle of processing a string (meaning, you have seen the opening quote but not the end quote), and you then see a character that is not part of the alphabet, treat the bad character as if it were a newline for the sake of processing the string. That is, generate an ERR2 token for an unterminated string, and have the token end right before the bad character (meaning tokenization should resume at the bad character).
6. When you see one or more consecutive characters that are not in the alphabet, group them together and generate an ERR3 token. The length associated with the token should be the number of consecutive characters that are not in our alphabet. Resume tokenizing as normal after the bad characters.
7. If you see a character in the alphabet that is not valid in its context, generate an ERR4 token for just the one character. If there are multiple such characters consecutively, generate multiple tokens.
8. When outputting a token, your output must consist of the following, in order:
    a. "TID:", with no spaces (or other characters) proceeding. All letters shall be output as capital ASCII letters.
    b. The colon may optionally be followed by spaces.
    c. The Token ID of the token you are outputting. Token IDs must start at 1 and increase by 1 for each token of the input.
    d. A single comma (note the token ID shall NOT be followed by spaces)
    e. The comma may optionally be followed by spaces
    f. "TYPE:". All letters shall be output as capital ASCII letters.
    g. The colon may optionally be followed by spaces.
    h. An integer representing the *Numeric Type* of the token.

i. The integer must only be followed by a left parenthesis— "(", meaning no spaces before the "(".
j. The left parenthesis must be followed by the "English Type" of the token (as indicated in the table above – case sensitive!), with no spaces preceding.
k. The English Type must be followed by a right parenthesis and comma— "),", meaning no spaces before the "),"
l. The comma may optionally be followed by spaces.
m. "POS:". All letters shall be output as capital ASCII letters.
n. The colon may be optionally followed by spaces.
o. An integer representing the position of the first character in the original input that led to the token match. The position is numbered from 0, and represents the number of 8-bit ASCII characters in the input stream that precede the character in question.
p. The integer must be followed by a single comma, with NO spaces (or other characters) in between.
q. The comma may optionally be followed by spaces.
r. "LEN:". All letters shall be output as capital ASCII letters.
s. The colon may be optionally followed by spaces.
t. An integer representing the number of bytes matched in the current token.
u. If the chart above indicates "None" in the "Value to output" column, then print a single newline (ASCII 0x0a – '\n'). The newline may optionally be preceded by spaces. IF THERE IS NO VALUE TO OUTPUT, YOU ARE DONE PRINTING THIS TOKEN.
v. The rest of the bullets are for when you are outputting a value only.
w. Output a comma (with NO preceding spaces), optionally followed by spaces, followed by "VALUE:", optionally followed by spaces.
x. Output the value, per the "Value to output" column above. All items that have a value should be a simple copy of the input, except for strings, which should omit the quotation marks around the string.
y. Print a single newline (ASCII 0x0a – '\n'). The newline may optionally be preceded by spaces.

9. Your program must take an optional command-line argument that dictates which tokens get output.
    a. If no command line argument is given, then you must output all tokens in the token stream.
    b. If the command line argument is a 0, you must also output all tokens in the token stream.
    c. If the command line argument is a 1, you must output all tokens EXCEPT comments, whitespace, errors and newlines.
    d. If the command line argument is a 2, you must output ONLY tokens for comments, whitespace, errors and newlines.
    e. If the command line consists of anything else other than the above four options, then you should IGNORE all input from stdin, assume the

input length is 0, and populate the token stream with only a single token of type ERR1, which you will then output, per below.

10. After reading in the entire input and generating tokens, output all tokens per the above specification. When you are done outputting all tokens you are supposed to output, then output the following:
    a. An additional newline (creating a blank line)
    b. The string "Totals:" (case sensitive, as with all strings in this assignment)
    c. Optional space(s)
    d. The string "len"
    e. Optional space(s)
    f. An equals sign
    g. Optional space(s)
    h. An integer indicating the length of the input stream (always 0 with ERR1, remember!)
    i. A comma
    j. Optional space(s)
    k. The string "tokens"
    l. Optional space(s)
    m. An equals sign
    n. Optional space(s)
    o. An integer indicating the number of tokens in the token stream.
    p. A comma
    q. Optional space(s)
    r. The string "printed"
    s. Optional space(s)
    t. An equals sign
    u. Optional space(s)
    v. An integer indicating the number of tokens you OUTPUT
    w. Optional space(s)
    x. A single newline.
11. After you finish outputting, your program must exit.

2. Other Requirements

You will receive a 0 on this if any of these requirements are not met!

12. The assignment is due on February 15 at 8am Eastern time. Late assignments will lose one letter grade per 24 hours.
13. The program must be written entirely in C or C++
14. You must submit a single source code file, unless you choose to use multiple files, in which case you must submit a single ZIP file, and nothing else.
15. If submitting a ZIP file, when the file unzips, your source files must unzip into the same directory (including any header files you need).
16. If submitting a ZIP file, there must not be ANY other files contained within the ZIP file. Again, you will get a 0 if there are.

17. If your program is written in C, it must compile ON MY REFERENCE ENVIRONMENT into an executable with the following command line: `cc *.c –o assignment1`
18. If your program is written in C, it must compile ON MY REFERENCE ENVIRONMENT into an executable with the following command line: `c++ *.cpp –o assignment1`
19. Your program should print nothing to stderr under any circumstances.
20. Your program's output will be tested in the reference environment only. Even if it works on your desktop, if it doesn't work in the reference environment, you will get a 0. With C and C++ this is a common occurrence due to memory errors, so be sure to test in the reference environment!
21. You must submit the homework through the course website, unless otherwise pre-approved by the professor.
22. You may not give or receive any help from other people on this assignment.
23. You may NOT use code from any other program, no matter who authored it.

3. Test Cases

Below are six sample test cases for you, which I will use in my testing. Typically, I use anywhere from 20-50 test cases (generally more than fewer). I will definitely use the below cases. I strongly recommend you create your own test harness and come up with a large number of test cases to help you get the best possible grade.

For test cases, what one would type on the command line is **BLACK**, input is in **GREEN**, and output is in **BLUE**.

Note that all the below test cases end with a final newline, and have no spaces before or after each line of input.

Case 1

```
./assignment1
1
x
TID: 1, TYPE: 3(INT), POS: 0, LEN: 1, VALUE: 1
TID: 2, TYPE: 6(NEWLINE), POS: 1, LEN: 1
TID: 3, TYPE: 1(ID), POS: 2, LEN: 1, VALUE: x
TID: 4, TYPE: 6(NEWLINE), POS: 3, LEN: 1

Totals: len = 4, tokens = 4, printed = 4
```

Case 2

```
./assignment1
1 # This is a comment
TID: 1, TYPE: 3(INT), POS: 0, LEN: 1, VALUE: 1
TID: 2, TYPE: 5(WS), POS: 1, LEN: 1
TID: 3, TYPE: 7(COMMENT), POS: 2, LEN: 19, VALUE: # This is a
comment
```

```
TID: 4, TYPE: 6(NEWLINE), POS: 21, LEN: 1

Totals: len = 22, tokens = 4, printed = 4
```

Case 3

```
./assignment1 1
1 # This is a comment
TID: 1, TYPE: 3(INT), POS: 0, LEN: 1, VALUE: 1

Totals: len = 22, tokens = 4, printed = 1
```

Case 4

```
./assignment1 2
1 # This is a comment
TID: 2, TYPE: 5(WS), POS: 1, LEN: 1
TID: 3, TYPE: 7(COMMENT), POS: 2, LEN: 19, VALUE: # This is a
comment
TID: 4, TYPE: 6(NEWLINE), POS: 21, LEN: 1

Totals: len = 22, tokens = 4, printed = 3
```

Case 5

```
./assignment1 3
1 # This is a comment
TID: 1, TYPE: 97(ERR1), POS: 0, LEN: 0

Totals: len = 0, tokens = 1, printed = 1
```

Case 6

```
./assignment1
"foo" "bar
12
TID: 1, TYPE: 2(STR), POS: 0, LEN: 5, VALUE: foo
TID: 2, TYPE: 5(WS), POS: 5, LEN: 1
TID: 3, TYPE: 98(ERR2), POS: 6, LEN: 4
TID: 4, TYPE: 6(NEWLINE), POS:10, LEN: 1
TID: 5, TYPE: 3(INT), POS: 11, LEN: 2, VALUE: 12
TID: 6, TYPE: 6(NEWLINE), POS: 13, LEN: 1

Totals: len = 14, tokens = 6, printed = 6
```

4. Reference Environment

You will be able to access the reference environment via SSH to kimota.net
Your user name for kimota.net will be your NYU netid.
Your password for this account will be sent when your account is activated, which will happen on or before Feb 5th.