Introduction

The goal is to implement an interpreter for an abstract stack machine language. Consider the following abstract stack machine bytecode instructions:

push NUM - pushes the integer value NUM on the stack.

push VAR - pushes the integer value stored in variable VAR on the stack.

pop - pops the value on the top of the stack and discards it.

print - pops the value on the top of the stack and prints it to the terminal screen.

store VAR - pops the value on the top of the stack and stores it in the variable VAR.

ask MSG - asks the user for an input value using the optional message MSG and then pushes that value on the stack.

dup - duplicate the value on the top of the stack; pop top of stack → temp, then push temp, and push temp again.

add - pop top of stack → temp1, pop top of stack → temp2, then push value temp2 + temp1.

sub - pop top of stack → temp1, pop top of stack → temp2, then push value temp2 - temp1.
mul - pop top of stack → temp1, pop top of stack → temp2, then push value temp2 * temp1.

div - pop top of stack → temp1, pop top of stack → temp2, then push value temp2 / temp1.

equ - pop top of stack → temp1, pop top of stack → temp2, then push value (temp2 == temp1)?1:0.

leq - pop top of stack → temp1, pop top of stack → temp2, then push value (temp2 <= temp1)?1:0.

jumpT LABEL - pop top of stack → temp, if temp != 0 then jump to instruction labeled LABEL.

jumpF LABEL - pop top of stack → temp, if temp == 0 then jump to instruction labeled LABEL.

jump LABEL - jump to instruction labeled LABEL.

stop MSG - halts execution with an optional message string MSG.

noop - does nothing.

Notes

• Execution halts with an error if not enough operands are available on the stack for an operation to complete.

• The bytecode language allows for labeled instructions, e.g.,

L1:

push 1;
push 2;
add;
print;
jump L1;

• You can use the following regular expressions to define the token types in your language:

VAR : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
NUM : '-'? '0'..'9'+;
COMMENT : '//' ~('
'|'')* ''? '
' {$channel=HIDDEN;};
WS : ( ' ' | '	' | '' | '
' ) {$channel=HIDDEN;};
STRING : '"' ( ESC_SEQ | ~('["\r\n']' ) )* '"';
ESC_SEQ : '\\' ( 'b'|'t'|'n'|'f'|'r'|'"'|'\'|'\''|'\'''|'\""');
Example Program

Print out the sequence of integer values 10, 9, 8,...,1:

```plaintext
push 10;
L1:
dup;
print;
push 1;
sub;
dup;
jumpT L1;
stop "all done!";
```

Problems:

1. Construct a grammar for the stack machine bytecode and show that your grammar works by parsing some telling examples (examples above are sufficient).

2. Implement an interpreter for this language (**Hint**: you will essentially have to construct an abstract stack machine, see the implementation of the exp1bytecode interpreter discussed in class). Show that your interpreter works with the above two examples. The interpreter needs to be able to run from the command line. **You have to use ANTLR to generate your parser.**

3. (Extra Credit) Write a program in the stack machine bytecode that asks the user for an input value and then computes and prints the factorial value of that input value. Your program should test to make sure that the input value is a valid value for the factorial computation and if not should terminate its computation. Definitions of the factorial computation can be found here:

   http://en.wikipedia.org/wiki/Factorial

Hand in your source code and proof that your program works. Please. The easiest way to show that your program works is to include a screen capture of the window that is executing your parser/grammar. Good software engineering/coding practices count towards your grade. Submission is via Sakai.