Computer Organization Module 8 Exercise: Grammar

Recall that a grammar specifies the behavior of a language. A grammar is composed of a set of production rules that (recursively) map "non-terminals" (variables) to either terminals (literal characters) or other non-terminals. Here is a sample grammar that supports positive and negative signed integers and allows leading 0's:

<number> = <sign> <digit>+ (where '+' is the regular expression

<sign> = '+' | '-'

<digit> = 0 | 1 | … | 9

Creating a Grammar

For this assignment you will create 2 grammars using the format in the example above. You will decide which terminals and non-terminals to implement and will write the set of recursive production rules.

Grammar #1: Hexadecimal Arithmetic

Your first grammar will support hex arithmetic. That is, specify a grammar that allows for signed addition and subtraction of two or more hex numbers of arbitrary length that consists of the 16 hexadecimal digits 0-F.

Grammar #2: A Mini Programming Language

Your other grammar will specify a miniature programming language that supports the following:

1. Creating a variable using the following format (VAR keyword followed by a variable name)

   ```
   VAR name
   ```

   Note: variable names may contain upper- and lower-case letters and numbers and must begin with a letter of either case

2. Defines a signed (decimal) integer data type that consists of an optional sign specifier and any number of digits 0-9

   ```
   1, -127
   ```

3. Defines a string data type that can hold an arbitrary-length string consisting of upper- and lower-case letters, spaces, and periods, and is contained within quotation marks:

   ```
   "Computer Organization", "Another valid string."
   ```

4. An assignment operator that assigns the value of one variable to another variable, an integer value to a variable, a string to a variable, or an expression (given in next step) to a variable:

   ```
   x = 0, y = x, s = "a string", x = y + z
   ```

5. The following 5 binary operations: addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**) of either two variables, two literal values, or one variable and one literal value

   ```
   x + y, 10 + 20, 2 * pi, "Hello " + "world"
   ```

   Note: your grammar should support string addition (concatenation), but all other operators are for the signed decimal integer type created in Step #2.

6. A print statement that prints the contents of a variable and has the following format:

   `PRINT <VAR>` where VAR is the 'name' value from a variable defined in Step #1

For each step in creating the second grammar, please also provide a regular expression to match each token. **You will use these to write a real tokenizer and parser for your grammar next week.** A list of tokens is provided below:

1. Variables
    1.1. Match the VAR keyword
    1.2. Match any string of length 1 or more that begins with an upper/lower-case character and contains only upper/lower-case characters and numbers
2. Signed Numbers
    2.1. Match a string of length 1 or more that contains only numbers and may be optionally preceded by a dash (indicating a negative number)
3. Strings
    3.1. Match a string of length 1 or more that contains only upper/lower-case letters, spaces, periods, and begins and ends with a quotations mark
4. Assignment Operator
    4.1. Match a liter equals sign (=)
5. Binary Operators
    5.1. Match a literal plus sign (+)
    5.2. Match a literal minus sign (-)
    5.3. Match a literal multiplication sign (*)
    5.4. Match a literal division sign (/)
    5.5. Match a literal exponentiation sign (**)
6. Print Method
    6.1. Match the PRINT keyword

Here is a regular expression cheatsheet to help get you started. You should strongly consider verifying the correctness of your regular expressions here.

To help get you started, your first element will be a 'program' non-terminal that contains a 'stmt-list' which, in turn, contains at least one statement. From there you will define how a 'stmt' can be constructed. For example, a 'stmt' should be able to represent an 'assignment' non-terminal which is, in turn, made up of a variable, followed by a literal equal sign (=), followed by either another variable, a (signed number), or a string (as defined in Step #4). You can hopefully begin to see the recursive nature of our grammar here.

```
<program> = <stmt-list>

<stmt-list> = <stmt>*
```