CMSI 488 Homework #1

Zane Kansil & Edward Bramanti

February 11, 2014

1. Write Regular Expressions for:

   (a) Canadian Postal Codes:

   ```
   [^DFIOQUWZ][\d][^DFIOQU] [\d][^DFIOQU][\d]
   ```

   (b) Legal Visa Card Numbers, not including checksums

   ```
   4\d{3} (\d{4} ){3}
   ```

   (c) MasterCard Numbers, not including checksums

   ```
   5\d{3} (\d{4} ){3}
   ```

   (d) Ada 95 numeric literals

   ```
   \d(_?\d)*#[\dA-F](_?[\dA-F])*#(E[+-]+[\dA-F](_?[\dA-F])*)?
   |\d(_?\d)*(\.\d(_?\d)*)?(E[+-]+\d(_?\d)*)?
   ```

   (e) Strings of letters and numbers beginning with a letter, EXCEPT those strings that are exactly three letters ending with two Latin letter ohs, of any case.

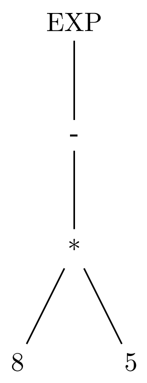
   ```
   ^\w(?![Oo][Oo]$)[\w\d]*
   ```

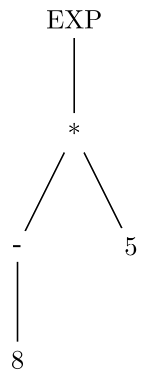
2. Syntax tree for Program in JSON (http://cs.lmu.edu/ ray/notes/syntax/)

```
{
    "Program" : [
        {"Var" : "x"},
        {"Var" : "y"},
        {"While" : [
            {"Minus" : ["y", 5]},
            [
                {"Var" : "y"},
                {"Read" : "x"},
                {"Read" : "y"},
                {"Assign" : ["x", {
                    "Times" : [2, {
                        "Plus" : [3, "y"]
                    }]
                }]}
            ]
        ]}
        {"Write" : 5}
    ]
}
```

3. In the Ada language comments are started with "–" and go to the end of the line. Therefore the designers decided not to make the unary negation operator have the highest precedence. Explain why this choice was made. Also, give an abstract syntax tree for the expression -8 * 5 and explain how this is similar to and how it is different from the alternative of dropping the negation from EXP2 and adding - EXP5 to EXP4.

   The designers made this choice so that you can have a `NEG|POS - POS` or a `NEG|POS + POS`, but never a `NEG|POS [+-] NEG`. Essentially, only positive numbers can be the second parameter of a binary addition. If someone were to do `9--5` it would be tokenized as the expression 9 followed by the comment containing '5'. It is important that comments be flexible and insertable in-line as opposed to only at the beginning of the line. Because comments are in-lineable, the language syntax must be able to tokenize the comment for grouping (so it can be excluded). This is done by disambiguating comments from similar-looking things in the language, like its `ADDOP` expressions.





   The effect of the change in the grammar is that negation can only be applied to an entire mathematical expression (an `EXP5`) and *not* individual int-literals. This is similar to Ada where mathematical operators are also applied between positive values. In order to get a negative mathematical value, the negative sign is prefixed to the evaluated mathematical expression. So for this example, `-5 * 4 -> (-(* (5 4)))` (prefix not.)

4. Programs in this language are made up of a non-empty sequence of function declarations, followed by a single expression. Each function declaration starts with the keyword **fun** followed by the function's name (an identifier), then a parenthesized list of zero or more parameters (also identifiers) separated by commas, then the body, which is a sequence of one or more expressions terminated by semicolons with the sequence enclosed in curly braces. Expressions can be numeric literals, string literals, identifiers, function calls, or can be made up of other expressions with the usual binary arithmetic operators (plus, minus, times, divide) and a unary prefix negation and a unary postfix factorial ("!"). There's a conditional expression with the weird syntax `"x if y else z"`. Factorial has the highest precedence, followed by negation, the multiplicative operators, the additive operators, and finally the conditional. Parentheses are used, as in most other languages, to group subexpressions. Numeric literals are non-empty sequences of decimal digits with an optional fractional part and an optional exponent part. String literals are delimited with double quotes with escape sequences followed by four hexadecimal digits. Identifiers are non-empty sequences of letters, decimal digits, underscores, at-signs, and dollar signs, beginning with a letter or dollar sign, that are not also reserved words. Function calls are formed with an identifier followed by a comma-separated list of expressions bracketed by parentheses. There are no comments in this language, and whitespace can be used liberally between tokens. Write the micro and macrosyntax of this language.

```
PROGRAM ::=    FUNDEC*  EXP
BLOCK   ::=    '{'  (EXP  ';')+  '}'
PARAMS  ::=    '('  ID  (','  ID)*  ')'
PARAMC  ::=    '('  (EXP | ID)  (','  (EXP | ID))*  ')'
FUNDEC  ::=    'fun'  ID  PARAMS  BLOCK
FUNCALL ::=    ID  PARAMC

EXP     ::=    EXP  ('if'  EXP1  ('else'  EXP)?)*
EXP1    ::=    EXP2  ([+-]  EXP2)*
EXP2    ::=    EXP3  ([*/]  EXP3)*
EXP3    ::=    '~'  EXP4
EXP4    ::=    EXP5  '!'
EXP5    ::=    '('  EXP  ')' | NUMLIT | STRLIT | ID | FUNCALL

CHAR    ::=    '\p{L}' | ESCCHR
ESCCHR  ::=    '\\' (['"rn\\] | u[\p{Nd}A-Fa-f]{4})
NUMLIT  ::=    '\p{Nd}'+ ('.'\p{Nd}+)? ([Ee][+-]?\p{Nd}+)?
ID      ::=    [\p{L}$][\p{L}\p{Nd}$@_]* that is not RESWORD
RESWORD ::=    ('fun' | 'if' | 'else')
STRLIT  ::=    '"'  CHAR*  '"'
```

5. Give an abstract syntax tree for the following Java code fragment:

```
if (x > 2 || !String.matches(f(x))) {
    write(- 3*q);
} else if (! here || there) {
    do {
        while (close) tryHarder();
        x = x >>> 3 & 2 * x;
    } while (false);
    q[4].g(6) = person.list[2];
} else {
    throw up;
}
```