CSC 231 Programming Languages Spring 2017

Our first programming project is intended to help your gain an understanding the fundamental steps of translating or interpreting a programming language. These steps include lexical analysis, parsing, error checking, and semantic actions. A translator for a complete language is far beyond the scope of this course. Instead, we will write an interpreter for a simplified language that is essentially limited to assignment statements, and arithmetic expressions.

The first step towards our goal is to write a lexical analyzer. The following is a table of the tokens we will need:

Token	**Lexeme**
lparen	'('
rparen	')'
add	'+'
subtract	'-'
multiply	'*'
divide	'/'
exponent	'**'
assign	'='
semi	';'
output	'print'
id	*a valid user-defined variable*
number	*a valid floating point number*
nomore	*end of file*

The Source Language

A program in our language is a sequence of statements separated by semi-colons. Only two types of statements are allowed.

- assignment statements of the form: *variable* = *expression*
- print statements of the form: *print variable*

Features and Limitations

The only types of expressions allowed are arithmetic expressions. Arithmetic expressions in our language are similar to arithmetic expressions in C/C++, with a few exceptions:

- An exponentiation operator (`**`) is included.
- The unary minus operator is not supported; e.g., `y = -x` must be written as `y = 0 - x` .
- The '-' character still has two roles. It can signify subtraction, or it can indicate the start of a negative number.
- All variable and numbers are type `float`.
- Numeric constants may contain a decimal point, but scientific "E" notation is not supported. E.g., `6.02e23` is not supported.

- Numbers between -1 and 1 might, or might not begin with a zero. E.g. `-0.5` and `-.5` are both valid.
- Numbers are not guaranteed to include a decimal point. E.g., `31` is a valid number.
- Numbers might or might not include digits after the decimal point. E.g. `32.` and `32.0` are both valid.
- Numbers must include at least one digit. E.g., ’`.`’ and ’`-.`’ are not valid.
- Variable names must begin with a letter: `a` – `z` or `A` – `Z`. After the first letter, variable names may contain only letters, digits, or the underscore character (`_`).
- Only scalar variables are supported. Arrays and aggregates are not supported.

All numbers must be real. Division by zero is a run-time error. Expressions leading to imaginary numbers, e.g., `(-1)**(1/2)` are not supported, and will be indicated as a run-time error. Program behavior is “undefined” for arithmetic expressions which result in values which can not be represented as type `float` in the underlying hardware.

NOTE: At this point in the semester, our goal is only to write a lexical analyzer which can take an input file and produce a stream of tokens.

Sample Input:

```
xabc = 3.141592  ;
y = -2.0         ;
z = xabc * y - 1 ;
print z      ;
```

Program Organization:

A lexical analyzer produces tokens “on demand”. I.e., a function (or method) named `get_token()` should return the next token. At end of file, a special token `nomore` should be returned. The main program should repeatedly call `get_token()` in a loop and print each token as it is discovered. The main program exits its loop when the token `nomore` is returned. The main program must accept a file name on the command line.

Lexical errors should be reported reasonably. No error recovery is necessary. Your program can simply exit after reporting an error.

Hints:

- Draw a complete DFA before starting to code the lexical analyzer. Numbers are more intricate than any other feature. Trace your DFA with several numerical forms to ensure correctness.
- When starting from state 0 and a ’`-`’ character is encountered, you will not yet know if the correct token is `subtract` or `number`. Simply accept the ’`-`’ and move to a new state. If the next character is a digit or a `.`, you can safely assume you have the start of a negative number. If the next character is not a digit, you should assume the ’`-`’ character corresponds to a `subtract` operation.
- Notice that the lexeme for `muliply` is a prefix of the lexeme for `exponent`.