CSC 231 Programming Languages Spring 2017
Programming Project Part II

Our 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 (a lexical analyzer) should be well behind us at this point in the semester. Our next three steps are:

1. Extend the simple example grammar (shown below) for expressions to include:
   - addition
   - subtraction
   - multiplication
   - division
   - exponentiation
   - assignment statement
   - print statement
   - statement sequence

   Be sure to keep the grammar LL(1) as you add productions. Use left factoring, and eliminating left recursion if necessary.
2. Implement a recursive descent parser for the language described by your grammar.
3. Add semantic actions to build a syntax tree for each expression. For each statement, evaluate the expression and perform the appropriate action.

A (Partial) LL(1) Grammar for Expressions:

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \ \mid\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \ \mid\ \epsilon \\
F & \rightarrow & (\ E\ ) \ \mid\ \text{identifier} \ \mid\ \text{number}
\end{array}
$$

The Source Language:

A program in our language is a sequence of statements; each statement is terminated by a semi-colon. Only two types of statements are allowed.

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

Sample Input:

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

Program Requirements:

Use recursive descent to construct your parser. Your parser must detect and report syntax errors. Your program can simply exit after reporting an error.

Left/Right Associativity Requirements:

The operators `+`, `-`, `*`, and `/` are left associative. That means expressions such as $a+b+c$ must be evaluated as $(a+b)+c$. The syntax tree you build should be consistent with left associativity for these operators. Note bene: `a - b - c` must be evaluated left-to-right as `(a - b) - c` . Exponentiation is right associative. The expression `a ** b ** c` must be evaluated as `a ** ( b ** c )`. The syntax tree you build should be consistent with right associativity for exponentiation.

Suggested Program Organization:

It is best to avoid a large monolithic program. A suggested[1] object-oriented organization into classes is as follows:

class buffer You will need to be able to read from the file. To properly report errors, you should keep an array holding the characters read on the current line. Keep track of the line number for error reporting.

class token This is your representation of the tokens in the language. The most important method of this class is `get_token()` which reads the next token from the input. Your parser will need to be able to get the next token as needed.

class parser This class will organize the mutually recursive functions used to implement a recursive descent parser. If you prefer to use global functions for the recursive descent parser, that is a reasonable alternative. In that case, a class will not be needed, but separate files "parser.h" and "parser.cc" are recommended.

Suggested Additional Classes:

In the final phase of our project, we will build a syntax tree using an attribute grammar and associated semantic actions. During this phase of the project you may need:

class symbol_table In the next phase of the project, you will build a syntax tree and evaluate it. When evaluating an expression, you will need to be able to store variables and their associated values. For example, if the input statements are:

```
x = 1.5 ;
y = 2.0 * x ;
```

At the point in time when you scan the expression `2.0 * x`, you will need to retrieve the value of `x` stored in by the previous statement. A good way to do this is to use a hash table, and let the variable names be the key.

class attribute This class will hold the attributes you need to construct the syntax tree. This will enable you to implement a syntax-directed definition for your attributes.

class syntax_tree This class will represent your syntax tree. You will need methods to build the trees and to evaluate the expressions they represent.

[1] There are many possible organizations that will do a good job with this project. This suggested organization is only one way to do it.

Hints:

- Check your grammar using `gcheck` on gottlieb to ensure there are no parse table conflicts. The `gcheck` program will provide the FIRST and FOLLOW sets for your grammar.
- Write out a predictive parsing table before writing your recursive descent C++ code.
- Carefully read and understand the outline for semantic actions.