CS224 – Parsing

Purpose: We would like to be able to use the grammar for a programming language to "parse" the tokens and see if they are in a legal configuration. A direct way of doing this is with a recursive descent parser.

Knowledge: This activity will help you become familiar with the following content knowledge:

- How to convert a grammar into a recursive descent parser.

Activity: With your group perform the following tasks and answer the questions. You will be reporting your answers back to the class in 30 minutes.

1. Consider this simple grammar for an expression:

   ```
   <exp> ::= <term>  | <term> + <exp>
   <term> ::=  id | const | ( <exp> )
   ```

   We have five types of tokens which we shall denote as ID, CONST, LPAREN, RPAREN, and PLUS.

   For a recursive descent parser we write a method for each of the non-terminals in the grammar and use the `advance` method to get the next token. Here is the method for `term`:

   ```
   public void term() {
          if (tokenType() == ID)
              advance();
          else if (tokenType() == CONST)
              advance();
          else if (tokenType() == LPAREN)
              { advance(); // consume left parenthesis
                expr();
                advance(); // consume right parenthesis
              }
   }
   ```

   Write the method for `exp`.

2. It is often useful to add extra bells and whistles to grammars. Two common grammar add-ons are `*` and `?`. The `*` indicates that a group of tokens may be repeated zero or more times. The `?` indicates that a group of tokens is optional.

   For example, consider the following grammar:

```
<x> ::= <y> b ( a b)*
<y> ::= c (d)* (e)?
```

   Some possible `<y>` strings would be `cddde` and `c`.

   Here is a method for `x`:

```
public void x() {
            y();
            advance(); // consume b
            while (tokenType() == a) {
                   advance(); // consume a
                   advance(); // consume b
            }
}
```

   Write the method for `y`

3. To give you a bit more practice, try writing methods for `x` and `y` with the following grammar:

```
<x> ::= <y> b | a (c | d)*
<y>::= c ((ee)?f)|g)*
```

   You should be ready to take the grammar for Jack and write a parser!