A Brief Look at More Serious Language Implementation

We have already played some with language design and implementation—you were asked in Project 1 to implement VPL, and I suffered greatly implementing Jive. But, those languages have relatively simple structure and can be described just using finite automata. Now we want to learn a little about implementing a language that is described using a *context free grammar*. It turns out that if a language is described using a finite automaton to describe its meaningful chunks, and a context free grammar of the right form to describe its main structure, then we can implement the language using *recursive descent parsing*.

> Not that long ago, probably every computer science program had a rather nasty required course known as "the compiler course." As this technology became more well-known and accepted, this course gradually disappeared from the curriculum, but scraps of it remain in this course.
>
> We will make our work much easier by mostly ignoring issues of *efficiency*, both in space and time. For example, a real compiler tries to recover from errors and go on to report other errors, but we will be happy to detect one error and stop. And, a real compiler tries to generate code that uses as little memory as is reasonable, and runs as quickly as is reasonable. We won't be crazy about it in our upcoming work, but we will compromise efficiency whenever it saves us work.

Language Specification

To specify a programming language, we have to specify its *grammar*—what sequences of symbols (or diagrammatic elements, if it's a non-textual language) form a syntactically legal program, and its *semantics*—the meaning of those sequences of symbols.

To precisely specify the grammar of our language, we need to do some fairly abstract work (all of which overlaps the CS 3240 course, but with a more practical focus).

We have already seen a little of both finite automata and context free grammars, but the following will be more detailed and organized.

As we do the abstract stuff, we will have in mind two languages. The first will be a simple calculator language. The second will be the language that you will be asked to implement in Project 2. Both of these languages will be designed and specified in the upcoming work. The implementation of the simple calculator language through a lexical phase and recursive descent parsing will be worked out fully as an example, leaving you to do the same things on Project 2.

Definition of a Language

Given a finite set, which is known as an *alphabet* (and can be thought of as some set of ordinary symbols), we define a *string* to be any finite sequence of members of the alphabet, and we define a *language* to be any set of strings over the alphabet.

> For this course, we are interested in situations where a single string is a program in some programming language, and the language is the set of all syntactically legal programs.

Finite Automata

We already introduced finite automata, so let's do some examples that will be used in the simple calculator language.

Exercise 9

Working in your small group, draw a finite automaton for each of the following languages:

a. all strings that start with a lowercase letter, followed by zero or more letters or digits

b. all strings that represent a real number in the way usually used by humans (no scientific notation allowed), say in a math class.

The Concept of Lexical Analysis as Just a First Step in Parsing

It turns out that the FA technique is not expressive enough to describe the syntax we want to have for a typical programming language.

> Here's a proof of this fact: we clearly want to have, in a typical imperative language, the ability to write expressions such as $((((((a))))))$, where the number of extra parentheses is unlimited. Our parser needs to be able to check whether there are as many left parentheses as right. But, the FA technique can't say this. Suppose to the contrary that we have an FA that accepts all strings that have n left parentheses followed by `a` followed by n right parentheses. Imagine that you are looking at this wonderful diagram. Since it is a *finite* state automaton, it has a finite number of states, say M. Now, feed this machine a string that has more than M left parentheses, followed by an `a`, followed by the same number of right parentheses. Since there are only M states, in processing the left parentheses, we will visit some state twice, and there will be a loop in the machine that processes just one or more left parentheses. Clearly this loop means that the machine will accept strings that it shouldn't accept.

Since the FA technique can't describe all the syntactic features we want in a programming language, we will be forced to develop a more powerful approach. But, the FA technique is still useful, because we will use it for the first phase of parsing, known as *lexical analysis*. This is a very simple idea: we take the stream of physical symbols and use an FA to process them, producing either an error message or a stream of *tokens*. A token is simply a higher-level symbol that is composed of one or more physical symbols. Then the remaining parsing takes place on the sequence of tokens. This is done purely as a matter of convenience and efficiency, because the technique we will develop to do the parsing is fully capable of doing the tokenizing part as well.

Formal Grammars and Parsing

We will now learn a technique for specifying the syntax of a language that is more powerful than the FA approach. One limited version of this approach will let us specify most of the syntax of a typical programming language.

A *formal grammar* consists of a set of *non-terminal symbols*, a set of *terminal symbols*, and a set of *production rules*. One non-terminal symbol is designated as the *start symbol*.

> As we saw in my specification of Jive, we will often use symbols written between angle brackets, like `<expr>`, to represent non-terminal symbols. Terminal symbols will be tokens produced by the lexical analysis phase.

For our early example we will tend to use uppercase letters for non-terminal symbols and lowercase letters for terminal symbols.
A production rule has one or more symbols (both terminal and non-terminal allowed) known as the "left hand side," followed by a right arrow, followed by one or more symbols known as the "right hand side."

A *derivation* starts with the start symbol and proceeds by replacing any substring of the current string that occurs as the left hand side of a rule by the right hand side of that rule, continuing until a string is reached that has only terminal symbols in it. At this point, that string of terminal symbols has been proven to be in the language accepted by the grammar.

A Simple Example

Here is a simple grammar, where the non-terminals are S, A, and B, and the S is the start symbol, and the terminal symbols are a, b, c, and d.

$S \to AB$
$A \to aAb$
$A \to c$
$B \to d$
$B \to dB$

Here is a sample derivation of a string from this grammar, where each line involves one substitution:

S
AB
$aAbB$
$aaAbbB$
$aacbbB$
$aacbbdB$
$aacbbddB$
$aacbbddd$

This derivation shows that `aacbbddd` is in the language accepted by this grammar.

⇒ Determine the language accepted by this grammar. Note that it has parts that could have been accomplished by the FA technique, and parts that could not.

Context Free Grammars

This formal grammar technique is a very general tool, but we will focus on a special version known as a *context free grammar* where the left hand side of a rule can only be a single non-terminal symbol.

Context free grammars are nice because on each step of a derivation, the only question is, which non-terminal symbol in the current string should we replace by using a rule where it occurs as the left hand side.

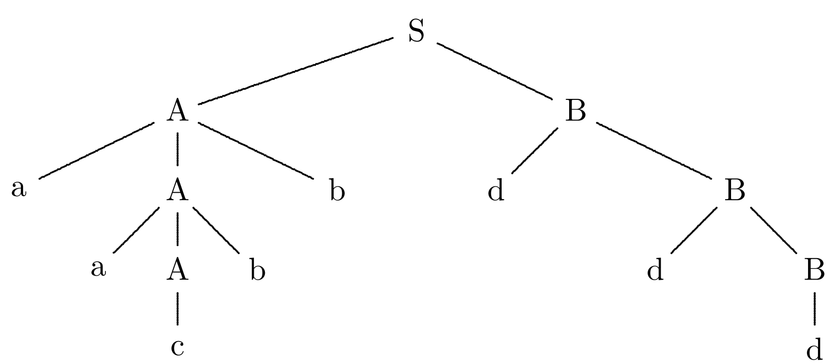
We can now think of what we really mean by parsing. For a context free grammar, the challenge is to take a given string—an alleged program—and try to find a derivation.

Context free grammars are used because they are powerful enough to specify most of what we want in the syntax of a language, but are still amenable to being parsed *efficiently*.

It is also important to note that for the languages people tend to use, apparently, the semantics (meaning) of a program are closely related to the syntactic structure.

> Here is a feature of a typical programming language that isn't captured, typically, by the context free grammar that specifies the syntax of the language: local variables in Java have to be defined before they are used. Typically the context free grammar says that the body of a method consists of a sequence of one or more statements, and statements are things like `int x;` and `x = 3;`, but the grammar doesn't specify that one has to occur before the other. Those rules of the language have to be handled in some other way.

For context free grammars, we can improve on the notation of a derivation a lot by the *parse tree* concept. Instead of recopying the current string on each step, we show the replacement of a left hand side non-terminal symbol by writing a child in the tree for each symbol in the right hand side. Here is a parse tree for the first example given:



A Bad Way to Try to Express Expressions

Consider the following grammar, where E is the start symbol, and V and N are viewed as terminal symbols namely tokens representing variables and numeric literals, and other symbols that are not uppercase letters are viewed as tokens.

$E \rightarrow N$
$E \rightarrow V$
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$

This grammar seems reasonable, but it turns out to be a rather poor way to specify a language consisting of all mathematical expressions using variables, numeric literals, addition, multiplication, and parentheses. For one thing, the grammar is ambiguous—there are strings that have significantly different parse trees. For another, the grammar has no concept of operator precedence.

⇒ Come up with some expressions and produce parse trees for them. Verify the complaints just made.

A Classic Example of a Context Free Grammar

Now consider this grammar for the language of simple expressions, where E is the start symbol, and V and N are viewed as terminal symbols, namely tokens representing variables and numeric literals, and other symbols that are not uppercase letters are viewed as tokens.

$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow F$
$T \rightarrow F * T$
$F \rightarrow V$
$F \rightarrow N$
$F \rightarrow (E)$

⇒ This grammar captures the syntax of an *expression* involving two binary operators with different precedence. Produce parse trees for some expressions and note that the grammar is not ambiguous and enforces operator precedence.

Parse Trees as Code

The previous example suggests a powerful idea: if someone gives you a parse tree for an expression, and values for all the variables occurring, you can easily compute the value of the expression—and you can easily write code that interprets the parse tree.

Thus, in a very real sense, the tree structure is a better programming language than the language described by the context free grammar, except for one big catch: it's hard to read and write parse trees, compared to reading and writing strings.

Exercise 10

Working in your small group, produce a parse tree for the following expression, using the good expression grammar. Then, assuming that $a = 12$, $b = 7$, and $c = 2$, show how the parse tree can be used to compute the value of the expression—note how the value of each node in the parse tree is obtained from the values of its children.

Use this expression:

$$7 * a + (c * a * c + 4 * a) * b$$

Designing and Specifying the Simple Calculator Language

Now we want to design (and then implement) a language that allows the user to perform a sequence of computations such as a person could do on a calculator. But, we want the language to allow use of expressions and assignment statements.

⇒ As a whole group, invent a name for this language, and write some example programs so everyone will see the sorts of things we want to allow, and the sorts of things we don't want to allow.

Exercise 11

Working in your small group, specify the simple calculator language by determining its tokens (categories of conceptual symbols, such as probably `<var>` for variable), drawing an FA for each kind of token, and writing a context free grammar for the language.

Use `<program>` for the start variable, which should produce any possible program in the language.

⇒ Working back together as a whole group, write down the final specification for the simple calculator language.

Implementing the Lexical Phase

⇒ As a whole group, produce a Java class named `Lexer` that will take a file containing a program in the simple calculator language as input and produce a sequence of tokens. In addition to producing this stream of tokens, `Lexer` should have the ability to "put back" a token, for convenience in the recursive descent parsing phase. The point is that we will produce a grammar that will allow us to look ahead at the next token, thinking it will be part of the entity we are producing, and then realize from that next token that we are done with the entity we were producing, so it is convenient to put back the look ahead token.

Implementing The Simple Calculator Language

⇒ Working as a whole group, finish the draft design of the simple calculator language, and write down a context free grammar to specify it.

As we do this, we want to try to write the rules so that they process symbols on the left, leaving the recursive parts on the right. For example, if we want to say "1 or more `a`'s" we could either use the rules

$$A \rightarrow Aa \mid a$$

or we could say

$$A \rightarrow aA \mid a$$

with the same result from a theoretical perspective. But, because we want to use the *recursive descent parsing* technique (there are other parsing techniques that we will not study that like grammars in different forms), we will want to use the second approach.

⇒ Working as a whole group, begin work on a class `Parser` that will take a `Lexer` as a source of tokens and produce a *parse tree* that represents a simple calculator language program in a way that it can be directly executed.

As a practical matter, to save some time we will begin with some old code that I produced for a different language and modify it massively to do the job for our new language.

Note that along with the `Lexer` and `Parser` classes, this old code includes `Token` and `Node` classes that will be used extensively in our work.

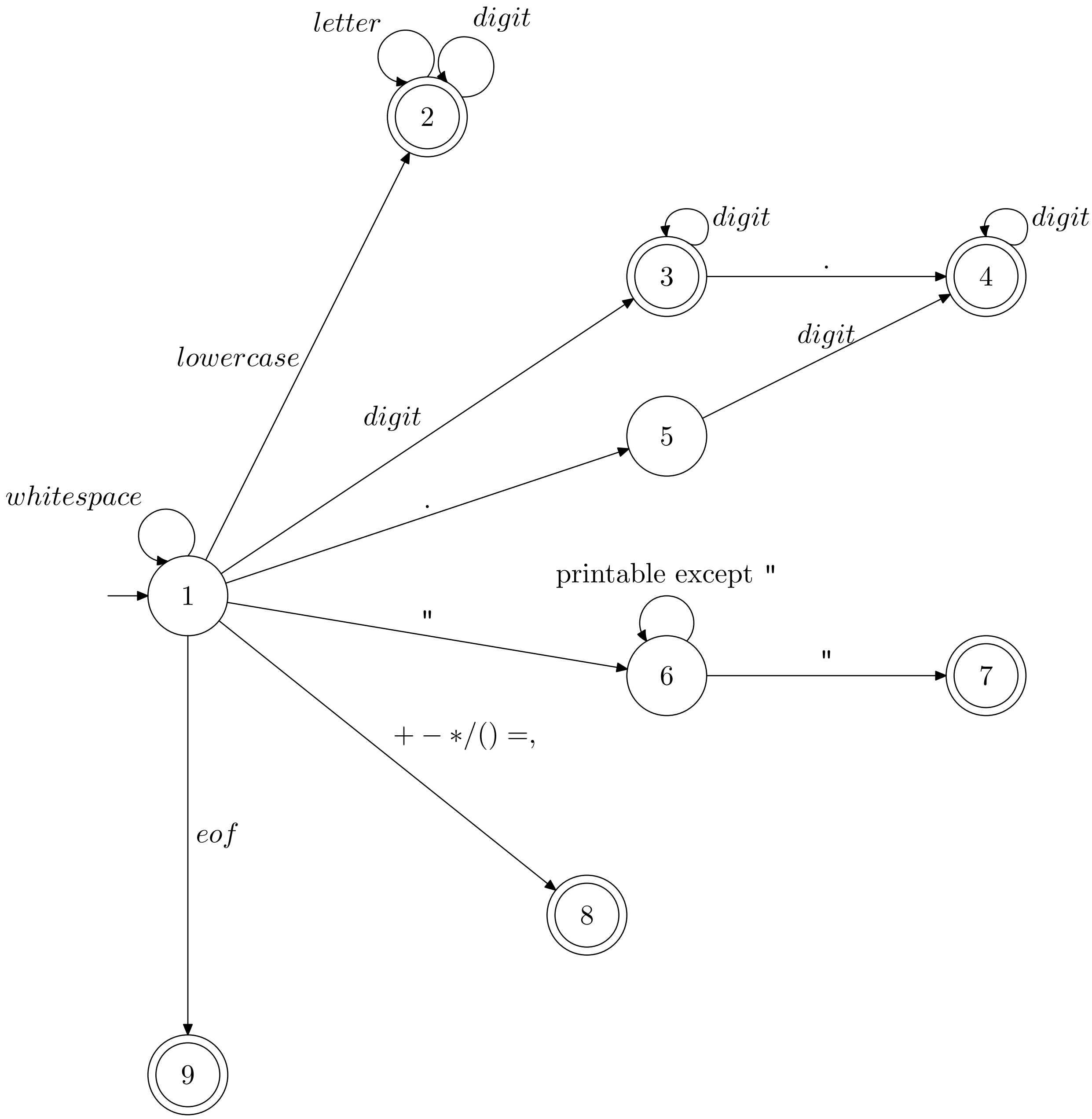
The classes `Basic`, `Camera`, and `TreeViewer` support visualizing a parse tree for debugging and testing purposes.

As we will see, the recursive descent parsing technique involves writing a method for every non-terminal symbol in the grammar. Each of these methods will consume a sequence of tokens and produce a node that is the root of a tree that is the translation of the corresponding chunk of code.

Also, these methods will return `null` if they realize that the next few tokens don't fit what they are expecting to see.

We will not take the time to be very careful about detecting and reporting errors in the source code. We will basically assume that the source code is correct. But, when it makes sense to do so, we will notice errors and halt the parsing process with some error message to the programmer.

Final (?) Finite Automata for Lexer for Corgi



⇒ Instructor will briefly discuss `Lexer` and `Parser` in preparation for Exercises 12 and 13.

Exercise 12 Working in your small group, write the code for state 3, 4, and 5 in the `Lexer.java` file for Corgi. To help you understand `Lexer` and do this Exercise effectively, a number of trees have been killed to provide you with a printout of the existing code on the next pages.

```
import java.util.*;
import java.io.*;
public class Lexer {

   public static String margin = "";

   // holds any number of tokens that have been put back
   private Stack<Token> stack;

   // the source of physical symbols
   // (use BufferedReader instead of Scanner because it can
   //  read a single physical symbol)
   private BufferedReader input;

   // one lookahead physical symbol
   private int lookahead;

   // construct a Lexer ready to produce tokens from a file
   public Lexer( String fileName ) {
     try {
       input = new BufferedReader( new FileReader( fileName ) );
     }
     catch(Exception e) {
       error("Problem opening file named [" + fileName + "]" );
     }
     stack = new Stack<Token>();
     lookahead = 0;  // indicates no lookahead symbol present
   }// constructor

   // produce the next token
   private Token getNext() {
      if( ! stack.empty() ) {
         //  produce the most recently putback token
         Token token = stack.pop();
         return token;
      }
      else {
         // produce a token from the input source

         int state = 1;  // state of FA
         String data = "";  // specific info for the token
         boolean done = false;
         int sym;  // holds current symbol

```

```
         do {
            sym = getNextSymbol();

// System.out.println("current symbol: " + sym + " state = " + state );

            if ( state == 1 ) {
               if ( sym == 9 || sym == 10 || sym == 13 ||
                    sym == 32 ) {// whitespace
                  state = 1;
               }
               else if ( 'a'<=sym && sym<='z' ) {// lowercase
                  data += (char) sym;
                  state = 2;
               }
               else if ( digit( sym ) ) {
                  data += (char) sym;
                  state = 3;
               }
               else if ( sym == '.' ) {
                  data += (char) sym;
                  state = 5;
               }
               else if ( sym == '\"' ) {
                 state = 6;
               }
               else if ( sym == '+' || sym == '-' || sym == '*' ||
                         sym == '/' || sym == '(' || sym == ')' ||
                         sym == ',' || sym == '='
                       ) {
                  data += (char) sym;
                  state = 8;
                  done = true;
               }
               else if ( sym == -1 ) {// end of file
                  state = 9;
                  done = true;
               }
               else {
                 error("Error in lexical analysis phase with symbol "
                                      + sym + " in state " + state );
               }
            }

            else if ( state == 2 ) {
```

```
            if ( letter(sym) || digit(sym) ) {
               data += (char) sym;
               state = 2;
            }
            else {// done with variable token
              putBackSymbol( sym );
              done = true;
            }
         }

         else if ( state == 3 ) {
            if ( digit(sym) ) {
               data += (char) sym;
               state = 3;
            }
            else if ( sym == '.' ) {
               data += (char) sym;
               state = 4;
            }
            else {// done with number token
              putBackSymbol( sym );
              done = true;
            }

         }

         else if ( state == 4 ) {
            if ( digit(sym) ) {
               data += (char) sym;
               state = 4;
            }
            else {// done with number token
              putBackSymbol( sym );
              done = true;
            }
         }

         else if ( state == 5 ) {
            if ( digit(sym) ) {
               data += (char) sym;
               state = 4;
            }
            else {
              error("Error in lexical analysis phase with symbol "
```

```
                                      + sym + " in state " + state );
            }
         }

         else if ( state == 6 ) {
            if ( (' '<=sym && sym<='~') && sym != '\"' ) {
               data += (char) sym;
               state = 6;
            }
            else if ( sym == '\"' ) {
               state = 7;
               done = true;
            }
         }

         // note: states 7, 8, and 9 are accepting states with
         //       no arcs out of them, so they are handled
         //       in the arc going into them
      }while( !done );

      // generate token depending on stopping state
      Token token;

      if ( state == 2 ) {
         // see if data matches any special words
         if ( data.equals("input") ) {
            return new Token( "bif0", data );
         }
         else if ( data.equals("sqrt") || data.equals("cos") ||
                   data.equals("sin") || data.equals("atan")
                 ) {
            return new Token( "bif1", data );
         }
         else if ( data.equals("pow") ) {
            return new Token( "bif2", data );
         }
         else if ( data.equals("print") ) {
            return new Token( "print", "" );
         }
         else if ( data.equals("newline") ) {
            return new Token( "newline", "" );
         }
         else {// is just a variable
            return new Token( "var", data );
```

```
            }
         }
         else if ( state == 3 || state == 4 ) {
            return new Token( "num", data );
         }
         else if ( state == 7 ) {
            return new Token( "string", data );
         }
         else if ( state == 8 ) {
            return new Token( "single", data );
         }
         else if ( state == 9 ) {
            return new Token( "eof", data );
         }

         else {// Lexer error
           error("somehow Lexer FA halted in bad state " + state );
           return null;
        }

     }// else generate token from input

   }// getNext

   public Token getNextToken() {
     Token token = getNext();
     System.out.println("                     got token: " + token );
     return token;
   }

   public void putBackToken( Token token )
   {
     System.out.println( margin + "put back token " + token.toString() );
     stack.push( token );
   }

   // next physical symbol is the lookahead symbol if there is one,
   // otherwise is next symbol from file
   private int getNextSymbol() {
     int result = -1;

     if( lookahead == 0 ) {// is no lookahead, use input
       try{  result = input.read();  }
       catch(Exception e){}
```

```
    }
    else {// use the lookahead and consume it
      result = lookahead;
      lookahead = 0;
    }
    return result;
  }

  private void putBackSymbol( int sym ) {
    if( lookahead == 0 ) {// sensible to put one back
      lookahead = sym;
    }
    else {
      System.out.println("Oops, already have a lookahead " + lookahead +
           " when trying to put back symbol " + sym );
      System.exit(1);
    }
  }// putBackSymbol

  private boolean letter( int code ) {
     return 'a'<=code && code<='z' ||
            'A'<=code && code<='Z';
  }

  private boolean digit( int code ) {
    return '0'<=code && code<='9';
  }

  private boolean printable( int code ) {
    return ' '<=code && code<='~';
  }

  private static void error( String message ) {
    System.out.println( message );
    System.exit(1);
  }

  public static void main(String[] args) throws Exception {
    System.out.print("Enter file name: ");
    Scanner keys = new Scanner( System.in );
    String name = keys.nextLine();

    Lexer lex = new Lexer( name );
    Token token;
```

```
      do{
        token = lex.getNext();
        System.out.println( token.toString() );
      }while( ! token.getKind().equals( "eof" )  );

    }

}
```

Exercise 13 Working in your small group, write the code for `parseFactor` in the class `Parser`, listed on the next few pages.

Here is the draft context free grammar for Corgi (note some changes from our earlier version):

```
<program> -> <statements>

<statements> -> <statement> |
                <statement> <statements>

<statement> ->  print <string> |
                print <expr> |
                newline |
                <var> = <expr>

<expr> -> <term> | <term> <addop> <expr>
<term> -> <factor> | <factor> <multop> <term>

<factor> -> <number> | <var> |
            ( <expr> ) |
            <bif0> () |
            <bif1> ( <expr> ) |
            <bif2> ( <expr>, <expr> ) |
            - <factor>
```

```
/*
    This class provides a recursive descent parser
    for Corgi (a simple calculator language),
    creating a parse tree which can be interpreted
    to simulate execution of a Corgi program
*/

import java.util.*;
import java.io.*;
```

```

public class Parser {

   private Lexer lex;

   public Parser( Lexer lexer ) {
      lex = lexer;
   }

   public Node parseProgram() {
      return parseStatements();
   }

   private Node parseStatements() {
      System.out.println("-----> parsing <statements>:");

      Node first = parseStatement();

      // look ahead to see if there are more statement's
      Token token = lex.getNextToken();

      if ( token.isKind("eof") ) {
         return new Node( "stmts", first, null, null );
      }
      else {
         lex.putBackToken( token );
         Node second = parseStatements();
         return new Node( "stmts", first, second, null );
      }
   }// <statements>

   private Node parseStatement() {
      System.out.println("-----> parsing <statement>:");

      Token token = lex.getNextToken();

      // ---------------->>>  print <string>  or   print <expr>
      if ( token.isKind("print") ) {
         token = lex.getNextToken();

         if ( token.isKind("string") ) {// print <string>
            return new Node( "prtstr", token.getDetails(),
                             null, null, null );
         }
```

```
         else {// must be first token in <expr>
            // put back the token we looked ahead at
            lex.putBackToken( token );
            Node first = parseExpr();
            return new Node( "prtexp", first, null, null );
         }
      // ---------------->>>  newline
      }
      else if ( token.isKind("newline") ) {
         return new Node( "nl", null, null, null );
      }
      // --------------->>>   <var> = <expr>
      else if ( token.isKind("var") ) {
         String varName = token.getDetails();
         token = lex.getNextToken();
         errorCheck( token, "single", "=" );
         Node first = parseExpr();
         return new Node( "sto", varName, first, null, null );
      }
      else {
         System.out.println("Token " + token +
                             " can't begin a statement");
         System.exit(1);
         return null;
      }

   }// <statement>

   private Node parseExpr() {
      System.out.println("-----> parsing <expr>");

      Node first = parseTerm();

      // look ahead to see if there's an addop
      Token token = lex.getNextToken();

      if ( token.matches("single", "+") ||
           token.matches("single", "-")
         ) {
         Node second = parseExpr();
         return new Node( token.getDetails(), first, second, null );
      }
      else {// is just one term
         lex.putBackToken( token );
```

```
         return first;
      }

   }// <expr>

   private Node parseTerm() {
      System.out.println("-----> parsing <term>");

      Node first = parseFactor();

      // look ahead to see if there's a multop
      Token token = lex.getNextToken();

      if ( token.matches("single", "*") ||
           token.matches("single", "/")
         ) {
         Node second = parseTerm();
         return new Node( token.getDetails(), first, second, null );
      }
      else {// is just one factor
         lex.putBackToken( token );
         return first;
      }

   }// <term>

   private Node parseFactor() {
      System.out.println("-----> parsing <factor>");

      Token token = lex.getNextToken();

      if ( token.isKind("num") ) {
         return new Node("num", token.getDetails(), null, null, null );
      }
      else if ( token.isKind("var") ) {
         return new Node("var", token.getDetails(), null, null, null );
      }
      else if ( token.matches("single","(") ) {
         Node first = parseExpr();
         token = lex.getNextToken();
         errorCheck( token, "single", ")" );
         return first;
      }
      else if ( token.isKind("bif0") ) {
```

```
            String bifName = token.getDetails();
            token = lex.getNextToken();
            errorCheck( token, "single", "(" );
            token = lex.getNextToken();
            errorCheck( token, "single", ")" );

            return new Node( bifName, null, null, null );
         }
         else if ( token.isKind("bif1") ) {
            String bifName = token.getDetails();
            token = lex.getNextToken();
            errorCheck( token, "single", "(" );
            Node first = parseExpr();
            token = lex.getNextToken();
            errorCheck( token, "single", ")" );

            return new Node( bifName, first, null, null );
         }
         else if ( token.isKind("bif2") ) {
            String bifName = token.getDetails();
            token = lex.getNextToken();
            errorCheck( token, "single", "(" );
            Node first = parseExpr();
            token = lex.getNextToken();
            errorCheck( token, "single", "," );
            Node second = parseExpr();
            token = lex.getNextToken();
            errorCheck( token, "single", ")" );

            return new Node( bifName, first, second, null );
         }
         else if ( token.matches("single","-") ) {
            Node first = parseFactor();
            return new Node("opp", first, null, null );
         }
         else {
            System.out.println("Can't have factor starting with " + token );
            System.exit(1);
            return null;
         }

      }// <factor>

   // check whether token is correct kind
```

```
  private void errorCheck( Token token, String kind ) {
    if( ! token.isKind( kind ) ) {
      System.out.println("Error:  expected " + token +
                         " to be of kind " + kind );
      System.exit(1);
    }
  }

  // check whether token is correct kind and details
  private void errorCheck( Token token, String kind, String details ) {
    if( ! token.isKind( kind ) ||
        ! token.getDetails().equals( details ) ) {
      System.out.println("Error:  expected " + token +
                          " to be kind=" + kind +
                          " and details=" + details );
      System.exit(1);
    }
  }

}
```

⇒ Instructor will discuss the minor changes to `Parser` (listed in its entirety below) and draw a parse tree for a Corgi program.

Exercise 14 Working in your small groups, construct the parse tree for `quadroots` following the final (I hope!) version of `Parser`.

```
/*
    This class provides a recursive descent parser
    for Corgi (a simple calculator language),
    creating a parse tree which can be interpreted
    to simulate execution of a Corgi program
*/

import java.util.*;
import java.io.*;

public class Parser {

   private Lexer lex;

   public Parser( Lexer lexer ) {
      lex = lexer;
   }

   public Node parseProgram() {
      return parseStatements();
   }

   private Node parseStatements() {
      System.out.println("-----> parsing <statements>:");

      Node first = parseStatement();

      // look ahead to see if there are more statement's
      Token token = lex.getNextToken();

      if ( token.isKind("eof") ) {
         return new Node( "stmts", first, null, null );
      }
      else {
         lex.putBackToken( token );
         Node second = parseStatements();
         return new Node( "stmts", first, second, null );
      }
   }// <statements>

   private Node parseStatement() {
      System.out.println("-----> parsing <statement>:");

      Token token = lex.getNextToken();

      // ---------------->>>  print <string>  or   print <expr>
      if ( token.isKind("print") ) {
         token = lex.getNextToken();

         if ( token.isKind("string") ) {// print <string>
            return new Node( "prtstr", token.getDetails(),
```

```
                              null, null, null );
            }
            else {// must be first token in <expr>
               // put back the token we looked ahead at
               lex.putBackToken( token );
               Node first = parseExpr();
               return new Node( "prtexp", first, null, null );
            }
         // ---------------->>>  newline
         }
         else if ( token.isKind("newline") ) {
            return new Node( "nl", null, null, null );
         }
         // --------------->>>   <var> = <expr>
         else if ( token.isKind("var") ) {
            String varName = token.getDetails();
            token = lex.getNextToken();
            errorCheck( token, "single", "=" );
            Node first = parseExpr();
            return new Node( "sto", varName, first, null, null );
         }
         else {
            System.out.println("Token " + token +
                                " can't begin a statement");
            System.exit(1);
            return null;
         }

      }// <statement>

      private Node parseExpr() {
         System.out.println("-----> parsing <expr>");

         Node first = parseTerm();

         // look ahead to see if there's an addop
         Token token = lex.getNextToken();

         if ( token.matches("single", "+") ||
              token.matches("single", "-")
            ) {
            Node second = parseExpr();
            return new Node( token.getDetails(), first, second, null );
         }
         else {// is just one term
            lex.putBackToken( token );
            return first;
         }

      }// <expr>

      private Node parseTerm() {
         System.out.println("-----> parsing <term>");

         Node first = parseFactor();

         // look ahead to see if there's a multop
         Token token = lex.getNextToken();

```

```
         if ( token.matches("single", "*") ||
              token.matches("single", "/")
            ) {
            Node second = parseTerm();
            return new Node( token.getDetails(), first, second, null );
         }
         else {// is just one factor
            lex.putBackToken( token );
            return first;
         }

      }// <term>

      private Node parseFactor() {
         System.out.println("-----> parsing <factor>");

         Token token = lex.getNextToken();

         if ( token.isKind("num") ) {
            return new Node("num", token.getDetails(), null, null, null );
         }
         else if ( token.isKind("var") ) {
            return new Node("var", token.getDetails(), null, null, null );
         }
         else if ( token.matches("single","(") ) {
            Node first = parseExpr();
            token = lex.getNextToken();
            errorCheck( token, "single", ")" );
            return first;
         }
         else if ( token.isKind("bif0") ) {
            String bifName = token.getDetails();
            token = lex.getNextToken();
            errorCheck( token, "single", "(" );
            token = lex.getNextToken();
            errorCheck( token, "single", ")" );

            return new Node( bifName, null, null, null );
         }
         else if ( token.isKind("bif1") ) {
            String bifName = token.getDetails();
            token = lex.getNextToken();
            errorCheck( token, "single", "(" );
            Node first = parseExpr();
            token = lex.getNextToken();
            errorCheck( token, "single", ")" );

            return new Node( bifName, first, null, null );
         }
         else if ( token.isKind("bif2") ) {
            String bifName = token.getDetails();
            token = lex.getNextToken();
            errorCheck( token, "single", "(" );
            Node first = parseExpr();
            token = lex.getNextToken();
            errorCheck( token, "single", "," );
            Node second = parseExpr();
            token = lex.getNextToken();
            errorCheck( token, "single", ")" );
```

```
         return new Node( bifName, first, second, null );
      }
      else if ( token.matches("single","-") ) {
         Node first = parseFactor();
         return new Node("opp", first, null, null );
      }
      else {
         System.out.println("Can't have factor starting with " + token );
         System.exit(1);
         return null;
      }

   }// <factor>

  // check whether token is correct kind
  private void errorCheck( Token token, String kind ) {
    if( ! token.isKind( kind ) ) {
      System.out.println("Error:  expected " + token +
                         " to be of kind " + kind );
      System.exit(1);
    }
  }

  // check whether token is correct kind and details
  private void errorCheck( Token token, String kind, String details ) {
    if( ! token.isKind( kind ) ||
        ! token.getDetails().equals( details ) ) {
      System.out.println("Error:  expected " + token +
                          " to be kind=" + kind +
                          " and details=" + details );
      System.exit(1);
    }
  }

}
```

Exercise 15 Below you will find a partial listing of the `Node` class, namely the partially completed `execute` and `evaluate` methods. Working in your small groups, write the missing code.

Note that nodes have instance variables `kind`, `info` (both strings), and `first` and `second` (both nodes).

Also, there is a `MemTable` class that has public methods
`void store( String name, double value)`
and
`double retrieve( String name)`

If you need further details about any of the classes in the `Corgi` project (namely `Corgi`, `Token`, `Lexer`, `Parser`, `Node`, and `MemTable`, you should look at them online (assuming someone in your group has a computer in class).

```
       // ask this node to execute itself
       // (for nodes that don't return a value)
        public void execute() {

           if ( kind.equals("stmts") ) {
               // insert code here for Exercise 15
           }

           else if ( kind.equals("prtstr") ) {
              System.out.print( info );
           }

           else if ( kind.equals("prtexp") ) {
               // insert code here for Exercise 15
           }

           else if ( kind.equals("nl") ) {
              System.out.print( "\n" );
           }

           else if ( kind.equals("sto") ) {
               // insert code here for Exercise 15
           }

           else {
              error("Unknown kind of node [" + kind + "]");
           }

        }// execute

        // compute and return value produced by this node
        public double evaluate() {

           if ( kind.equals("num") ) {
              return Double.parseDouble( info );
           }

           else if ( kind.equals("var") ) {
               // insert code here for Exercise 15
           }

           else if ( kind.equals("+") || kind.equals("-") ) {
               // insert code here for Exercise 15
           }
```

```

         else if ( kind.equals("*") || kind.equals("/") ) {
            // insert code here for Exercise 15
          }

          else if ( kind.equals("input") ) {
             return keys.nextDouble();
          }

          else if ( kind.equals("sqrt") || kind.equals("cos") ||
                    kind.equals("sin") || kind.equals("atan")
                  ) {
             double value = first.evaluate();

             if ( kind.equals("sqrt") )
                return Math.sqrt(value);
             else if ( kind.equals("cos") )
                return Math.cos( Math.toRadians( value ) );
             else if ( kind.equals("sin") )
                return Math.sin( Math.toRadians( value ) );
             else if ( kind.equals("atan") )
                return Math.toDegrees( Math.atan( value ) );
             else {
                error("unknown function name [" + kind + "]");
                return 0;
             }

          }

          else if ( kind.equals("pow") ) {
             // insert code here for Exercise 15
          }

          else if ( kind.equals("opp") ) {
             // insert code here for Exercise 15
          }

          else {
             error("Unknown node kind [" + kind + "]");
             return 0;
          }

       }// evaluate

```

Designing Project 2

To keep Project 2 manageable (a first serious submission for Project 2 must be submitted by the time of Test 2—October 29), I have decided to make this project be to simply (we hope!) add some features to Corgi as follows.

> First, we want to give the Corgi programmer the ability to define functions, and of course to call them.
>
> Second, we want to add branching to Corgi.

⇒ Working as a whole group, design the finite automata for the Corgi `Lexer`, and the context free grammar for the Corgi `Parser`. Discuss the features we are not adding to Corgi, and perhaps decide to add some of them. As part of this design process, write some sample Corgi programs (these will also provide a starting point for testing your Project 2 work).

⇒ Once the syntax of the language is specified, try to specify the *semantics* of the language, and think about how execution/evaluation of the parse tree will be done.

Note: after we have made decisions about the new and improved Corgi, henceforth to be known as "Corgi," I will document them. But, you should be able to start working on Project 2 right away, based on the informal documentation recorded during this class session.

Project 2 [10 points] [first serious submission absolutely due by October 29]

Implement the extended Corgi language as specified in class and in upcoming documents. Be alert for corrections to the Corgi specification that may need to be made as work proceeds.

Be sure to test your implementation thoroughly.

Email me your group's submission with a single `zip` or `jar` file attached, named either `corgi.jar` or `corgi.zip`. This single file must contain all the source code files. Be sure to CC all group members on the email that submits your work.

I must be able to extract all the Java source files from your submitted file into a folder, and then simply type at the command prompt:

```
javac Corgi.java
java Corgi test3
```

in order to run the Corgi program in the file named `test3`.
