CS403: Programming Languages

Recognizer/Parser Module

Building a recognizer

This is your third module for your Designer Programming Language assignment. You may develop your code using any of the allowed procedural language, but you must ensure it runs correctly under Linux before submission.

Your task is to write a *recognizer* for your language. A recognizer is like a souped-up scanner; additionally, it determines whether or not the lexemes are in the proper order. One builds a recognizer by implementing a grammar.

Recursive Descent Parsing

Once you have developed a grammar for your language, the next task is to build a *recognizer* for your language via recursive descent parsing. A recognizer is a program which says whether the expressions (sentences) in your source code are syntactically legal. A parser is an implementation of a grammar. A recursive descent parser is a parser composed of a set of parsing functions, each of which derives from a rule in the grammar. It should become apparent why the appellation recursive descent is used to describe parsers built in this fashion. Begin by reading *this*.

Transforming grammars

There is a straightforward transformation from a grammar to a set of recursive descent parsing functions implementing the grammar. Each grammar rule becomes a single function implementing the grammar rule. The terms *lhs* and *rhs* denote the left-hand side and the right-hand side of the grammar rule, respectively, with the colon serving as the line of demarcation.

- the *lhs* side of a rule becomes the name of the implementing function – the terminals and non-terminals of the *rhs* guide the implementation of the body of the function
- each terminal on the *rhs* corresponds to a call to a function named *match* – *match* is used to ensure the given terminal is pending in the data stream
- a vertical bar on the *rhs* indicates a need to call a function named *check* – *check* is used to see which alternative is present in the data stream
- each non-terminal on the *rhs* corresponds to a call to the function named by that non-terminal.

Even before we get into the details of *match* and *check*, an example will be quite useful. Consider the grammar rule for specifying an arbitrarily long sum of numbers:

```
sum : NUMBER
    | NUMBER PLUS sum
```

An implementation of this grammar rule using the above transformation strategy yields:

```
function sum()
    {
    match(NUMBER);
    if (check(PLUS))
        {
        match(PLUS);
        sum();
        }
    // else done
    }
```

Note that both alternatives begin with the terminal `NUMBER`, so a call to `match(NUMBER)` is made. The function *match* is used to ensure that the next lexeme in the data stream is a `NUMBER`. If *match* fails, that means that the input is not valid, i.e. the words are not in the correct order. If *match* is successful, it advances the lexical stream (via a call to *lex*). Assuming a `NUMBER` was matched, we then check to see which alternative is present in the lexical stream. If a `PLUS` lexeme is pending, then the second alternative must be present – we match the `PLUS` and then recursively call *sum*. If not, the first alternative was present in the lexical stream; there is nothing more to do since we matched entirety of the sum with the initial call to *match*.

It is relatively easy to see that not all grammars are suitable for recursive descent parsing. Suppose the above grammar rule for sum had been written equivalently as:

```
sum : NUMBER
    | sum PLUS NUMBER
```

Using the transformation strategy, we get:

```
function sum()
    {
    if (check(NUMBER);
        {
        match(NUMBER);
        }
    else
        {
        sum();
        match(PLUS);
        match(NUMBER);
        }
    }
```

There are major problems with this function. Suppose a `NUMBER` lexeme is not pending in the lexical stream. The else branch is then taken and an immediate recursive call to *sum* is made. Since the lexical stream has not been advanced (only *match* advances the lexical stream), the function falls into a recursive infinite loop. If there is a NUMBER pending, followed by a PLUS, then the else branch is never taken and the following PLUS is never recognized.

For this reason, one of the primary rules of recursive descent parsing is that the grammar may not have any left recursion, either directly or indirectly.

Usually, common sense can be used to eliminate direct left recursion. Since all non-terminals must eventually ground out to terminals, we examine *sum* and see that a *sum* must begin with a `NUMBER`. as stated prior. Moreover, it is relatively easy to see from the rule that a sum is simply a string of numbers separated by plus signs. The right recursive version of sum is immediately suggested from this fact.

Support functions for recursive descent parsing.

A small number of lexical helper functions simplify the task of creating a recursive descent parser, two of which, *match* and *check*, are mentioned in the previous section. These helper functions and their uses are:

function	**purpose**
advance	move to the next lexeme in the input stream
match	like *advance* but checks the current lexeme
check	check whether or not the current lexeme is of the given type

The functions are considered lexical interface functions because they deal with the lexemes that make up the input stream and isolate the parser from the details of the lexical analyzer. In addition to simplifying the design of the parser, they, themselves, are also easy to implement. Here are basic implementations of *advance*, *match*, and *check*:

```
function check(type)
    {
    return type(CurrentLexeme) == type;
    }

function advance()
    {
    CurrentLexeme = lex();
    }

function match(type)
    {
    matchNoAdvance(type);
    advance();
    }

function matchNoAdvance(type)
    {
```

```
    if (!check(type))
        fatal("syntax error");
    }
```

The variable *currentLexeme* is global to these interface functions.
If it hasn't yet occurred to you, the set of terminals in a grammar is exactly the set of lexemes returned that can possibly be returned by *lex*.

Recognizing expressions

Recall the expression rule:

```
expression : primary operator expression
           | primary

operator : PLUS
         | TIMES

primary : NUMBER
        | VARIABLE
```

At this point, let's try to implement the expression rule as a recursive descent parsing function. We get something like:

```
function expression()
    {
    primary();
    if (operatorPending())
        {
        operator();
        expression();
        }
    }
```

Note that the function *operatorPending* is used instead of *check* to distinguish between the two alternatives. Examination of *check* reveals that it is used to distinguish terminals pending on the lexical stream. Here, an attempt is being made to see if a non-terminal is pending. A non-terminal is *pending* if any of the terminals comprising its *first set* are pending on the lexical stream. The *first set* of a non-terminal is simply the set of terminals that can begin the non-terminal. In the case of operator, the first set is `{PLUS,TIMES}`, yielding the following implementation of *operatorPending*:

```
function operatorPending()
    {
    return check(PLUS) || check(TIMES);
    }
```

More generally, given a rule of the form:

```
a : A b | C d | e f
```

where lowercase letters are non-terminals and uppercase letters are terminals, the associated pending function for the rule would be:

```
function aPending()
    {
    return check(A) || check(C) || ePending();
    }
```

In general, use *check* to see if a terminal is pending in the lexical stream and define a *zzzPending* function to see if the non-terminal *zzz* is pending in the lexical stream.

Consider, as before, expanding the *primary* rule so that expressions can involve function calls and can be grouped with parentheses:

```
primary : NUMBER
        | VARIABLE
        | VARIABLE OPAREN optEexpressionList CPAREN
        | OPAREN expression CPAREN
```

Here is the implementation of the new primary:

```
function primary()
    {
    if (check(NUMBER))
        {
        match(NUMBER);
        }
    else if (check(VARIABLE))
        {
        // two cases!
        match(VARIABLE);
        if (check(OPAREN))
            {
            match(OPAREN);
            optExpressionList();
            match(CPAREN);
            }
        }
    else
        {
        match(OPAREN);
        expression();
        match(CPAREN);
        }
    }
```

Again, note the close correspondence between the function *primary* and the *primary* grammar rule. The only tricky part is handling the two alternatives that begin with `VARIABLE`. These are folded into a single top-level alternative. Inside that alternative, the two forms are distinguished with a call to `check(OPAREN)`. To reduce this complexity, the grammar rule for primary could have been written more clearly as:

```
primary : NUMBER
        | varExpression
        | OPAREN expression CPAREN

varExpression : VARIABLE
              | VARIABLE OPAREN optEexpressionList CPAREN
```

yielding implementations of:

```
function primary()
    {
    if (check(NUMBER))
        {
        match(NUMBER);
        }
    else if (varExpressionPending())
        {
        varExpression();
        }
    else
        {
        match(OPAREN);
        expression();
        match(CPAREN);
        }
    }

varExpression()
    {
    match(VARIABLE);
    if (check(OPAREN))
        {
        match(OPAREN);
```

```
            optExpressionList();
            match(CPAREN);
            }
        }

    varExpressionPending()
        {
        return check(VARIABLE);
        }
```

The grammar rule and implementation of *optExpressionList* is similar the grammar rules for list seen earlier.

Conditionals and iterations

Using the rules outlined above, implementing conditionals and iterations is straightforward. You should try to implement these grammar rules for practice:

```
ifStatement : IF OPAREN expression CPAREN block optElse

block : OPAREN  statementList CPAREN

statementList : statement
              | statement statementList

statement : expression SEMI
          | ifStatement
          | INT VARIABLE optInit SEMI

optElse : ELSE block
        | *empty*

optInit : ASSIGN expression
        | *empty*
```

and:

```
whileStatement : WHILE OPAREN expression CPAREN block
```

Summary

In summary, each grammar rule corresponds to a function. Each non-terminal on the right-hand side corresponds to a function call to the function that associated with that non-terminal. Each terminal corresponds to a call to *match*. Each "or" corresponds to a call to *check* or a pending function. It is as simple as that.

As a final note, the recursive descent functions described thus far do not return anything. The functions merrily move through the lexical input stream until it is exhausted or until *match* detects an error. Later, you will modify your recognizer so that it returns *parse trees*.

Coding Standards

Coding standards are as before.

Specifics

Specifically, you are to write a recognizer for your language. You should complete your grammar implement an appropriate recognizer. One should be able to run the recognizer on a file consisting of a single program by typing in the command:

```
recognizer <f>
```

where `<f>` is the name of the file to be recognized. The output of the program should be the string "legal" or "illegal". In the case of an illegal expression, the error generated by *match* should be displayed subsequently.

You should supply a number of clearly marked test cases. A README file should specify which test cases result in which outputs.

Submitting the assignment

To submit your assignment, delete all object or class files from your working directory, leaving only source code, a makefile, a README file, and any test cases you may have. Then, while in your working directory, type the command

```
submit proglan lusth recognizer
```

The submit program will bundle up all the files in your current directory and ship them to me. This includes subdirectories as well since all the files in any subdirectories will also be shipped to me.

You may submit as many times as you want before the deadline; new submissions replace old submissions.

Make sure you that I will be able to call your recognizer with the command named *recognizer*. The command *recognizer* must take a filename as a command line argument.

You must supply a makefile that will compile your recognizer when responding to the commands *make* and *make run*. The make run command should test your implementation on your own test files. There should be at least one "good" test file and a few "bad" ones illustrating various syntax errors.