CS403: Programming Languages

Recognizing Languages

Introduction

A recognizer is like a souped-up scanner; Like a scanner, it repeatedly calls *lex* to serve up lexemes. Additionally, it checks each lexeme against a grammar that describes the language in the text being scanned. Ultimately, a recognizer reports whether the entire text is syntactically correct or not.

Recursive Descent Parsing

Building a recognizer for a typical programming language grammar is rather easy. There are many tools for turning a grammar into a recognizer, but there is a technique called *recursive descent parsing* that obviates the need for tools. A *parser* is a recognizer that also builds an abstract representation of the text being scanned and is necessary for creating a language interpreter. However, it is useful to use the recursive descent technique to construct a recognizer first and then convert the recognizer into a parser.

A recursive descent recognizer is composed of a set of functions, each of which corresponds to a rule in the grammar. It will become apparent why the appellation *recursive descent* is used to describe recognizers and parsers built in this fashion.

Transforming grammars

There is a straightforward transformation from a grammar to a set of recursive descent functions. 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 predicate function named *check* or a *pending* function – these predicates are 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*, *check*, and the *pending* functions, 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 lexemes 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. that rule to a right-recursive format. There is a formal technique for If a right-recursive analog is not apparent, there is a formal way to deal with immediately left-recursive rules. Given a left recursive rule of the form:

```
a : a x | b
```

where *a* and *b* are strings of terminals and non-terminals, an equivalent pair of rules is:

```
a : b r
r : x r | *empty*
```

Performing such a transformation on the *sum* rule binds *x* to `PLUS NUMBER` and *b* to `NUMBER`, yielding:

```
sum : NUMBER sumR

sumR : PLUS NUMBER sumR | *empty*
```

These two rules, neither of which is left-recursive, replace the original *sum* rule.

Support functions for recursive descent parsing.

A small number of lexical helper functions simplify the task of creating a recursive descent recognizer, 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 forces the current lexeme to be matched
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 recognizer from the details of the lexical analyzer. In addition to simplifying the design of the recognizer, 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 : unary operator expression
               | unary

    operator : PLUS
             | TIMES

    unary : 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()
        {
        unary();
        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 *unary* rule so that expressions can involve function calls and can be grouped with parentheses:

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

Here is the implementation of the new unary:

```
function unary()
    {
    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 *unary* and the *unary* 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 *unary* could have been written more clearly as:

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

varExpression : VARIABLE
              | VARIABLE OPAREN optEexpressionList CPAREN
```

yielding implementations of:

```
function unary()
    {
    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.

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 *abstract syntax trees*, commonly referred to as *parse trees*.