CS403: Programming Languages

Notes on Parsing

Once you have developed an context-free grammar for your language, the next task 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 recursive descent parser is composed of a set of parsing functions, each of which derives from a rule in the grammar. Not all grammars are suitable for recursive descent parsing. Suppose the grammar for your language is:

```
expression : unary
           | expression operator expression

operator : PLUS | MINUS | TIMES | DIVIDE

unary : INTEGER
      | VARIABLE
      | VARIABLE OPAREN optExpressionList CPAREN
      | OPAREN expression CPAREN

optExpressionList : expressionList | e

expressionList : expression
               | expression COMMA expressionList
```

If a grammar rule is left-recursive, then the associated parsing function can, in some cases, fall into an infinite recursive loop. For example, the rule for *expression* is left-recursive (that is, in one branch of *expression*, the first thing is *expression*) so we will need to transform that rule to a right-recursive format. There is a formal technique for dealing with immediately left-recursive rules such as expression. 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 | e
```

Performing such a transformation on the expression rule yields

```
expression : unary expressionR

expressionR : operator expression expressionR | e
```

These two rules, neither of which is left-recursive, replace the original expression rule. Unfortunately, the technique generates some non-intuitive rules. In this particular case, there is a more intuitive approach. We note that another way to look at the grammar rule:

```
expression : unary
           | expression operator expression
```

is that an expression is simply a number of primaries connected by operators. Thus we can rewrite the rule either as:

```
expression : unary
           | unary operator expression
```

or:

```
expression : unary
           | expression operator unary
```

We choose the first rewrite since it is not left recursive.

Unfortunately, in both the formal and intuitive rewrite of the left-recursive rule, we end up implementing a right associative grammar. That is, expressions such as:

```
a - b - c - d
```

will be recognized as:

```
(a - (b - (c - d)))
```

which is contrary to the view of subtraction we have likely been taught in grammar school. Does this mean that recursive descent parsing can only be used for *right-associative* expressions? The answer is no, but it will take an engineering trick to accomplish that goal (more on associativity later). Transforming grammars

Once all left recursion is eliminated, there is a straight-forward transformation from a grammar rule to a function which implements the grammar rule:

- each *lhs* side of a rule becomes a function definition with the *lhs* as the name of the function - the body of the function corresponds to *rhs*
- in the body, each terminal on the *rhs* corresponds to a call to a function named match
- in the body, each non-terminal on the *rhs* corresponds to a call to the function named by that non-terminal
- in the body, different branches of the *rhs* correspond to calls to a function named check (or a pending function - more on that later)

A small number of lexical helper functions will be needed though, two of which are mentioned above . These helper functions are:

function	***purpose***
advance	move to the next lexeme in the input stream
match	insist that the current lexeme is of the given type
	- if it is, advance is called
	- otherwise an error is reported
check	check whether or not the current lexeme is of the given type

They are consider lexical functions because they deal with the lexemes that make up the input stream. Here is a recursive descent parser which recognizes strings in the language specified by the above grammar. Note that the parser just recognizes, it does not build up a parse tree. Also note that good style insists that the grammar rule precede its associated parsing function. Implementing grammars via recursive descent
Let's implement the right associative grammar rule:

```
expression : unary operator expression | unary
```

as a recursive descent parsing function. Recall that each symbol to the left of the colon is called a *non-terminal.* Each non-terminal will correspond to a function. The right hand side of the rule, which the non-terminal heads, will correspond to the body of the function. Non-terminals on the right hand side correspond to a call to the function associated with that non-terminal. Here is the function corresponding to *expression*:

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

Since an expression can be two things, a *unary* or a *unary* followed by an operator and then another expression, we use *operatorPending* to see if an operator is indeed pending. The function operatorPending abstracts a series of calls to check. If an operator is pending, then we know we have the second version of expression.

Note that the function does not return anything. If the non-terminal functions are written in this manner, the resulting program can recognize sentences in the language, but cannot execute them. These types of programs are known as language recognizers or grammar checkers. Later we will use these functions to build parse trees and then manipulate those parse trees.

For another example, consider the rule:

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

Here is its implementation:

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

The function *advance* is used to advance the input to the next token in the stream. The function match is used to ensure that the current input token is the supplied terminal. If the input does not match, an error is reported. If it does, then *advance* is called to move to the next input token. In general, the function check and pending functions are used to implement decision points in the grammar rule, the function *advance* is used to skip past input that has already been analyzed, and the function match is used to ensure that terminals appear in the input in their proper places.

Note the close correspondence between the function *unary* and the grammar rule headed by *unary*.

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 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.

Here is a complete parser for the set of grammar rules shown at the very top of this page. The parser assumes that top-level expressions are terminated with semicolons.

```
function parse()
    {
    // Start symbol is expression, terminator is SEMI

    advance();
    expression();
    match(SEMI);
    }

/////////// recursive descent parsing functions ///////////

// expression : unary op expression | unary

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

```
// operator : PLUS | MINUS | TIMES | DIV

function operator()
    {
    // must be an operator pending or would not have gotten here
    // use advance instead of match since we don't know exactly
    // which operator is pending

    advance();
    }

// unary : INTEGER
//         | VARIABLE
//         | VARIABLE OPAREN optArgumentList CPAREN
//         | OPAREN expression CPAREN

function unary()
    {
    if (check(INTEGER))
        {
        match(INTEGER);
        }
    else if (check(VARIABLE))
        {
        advance();
        // two cases!
        if (check(OPAREN))
            {
            advance();
            optArgumentList();
            match(CPAREN);
            }
        }
    else // must be a parenthesized expression
        {
        match(OPAREN);
        expression();
        match(CPAREN);
        }
    }

// optArgumentList : argumentList | empty

function optArgumentList()
    {
    if (argumentListPending())
        {
        argumentList();
        }
    // else empty
    }

// argumentLists: expression
//              | expression COMMA argumentList

function argumentList()
    {
    expression();
    if (check(COMMA))
        {
        match(COMMA);
        argumentList();
        }
    }
    ////////////// first set functions ///////////////

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

    function argumentListPending()
        {
        return expressionPending();
        }

    function expressionPending()
        {
        return unaryPending();
        }

    function unaryPending()
        {
        return check(INTEGER) || check(VARIABLE) || check(OPAREN);
        }

    /////////////// parser utility functions //////////////////

    function advance()
        {
        var old = Current;

        // get the next lexeme in the input stream

        Current = lex();

        return old;
        }

    function match(int type)
        {
        if (check(type))
            {
            return advance();
            }

        ParseException("Parse error: looking for %s, found %s instead\n",
        TypeToStr(type), Current.display());

        return null;
        }

    function check(int type)
        {
        return type == Current->type;
        }
```

Notice how closely the implementations of the grammar rules follow the rules themselves. Also, note that match is returning the matched lexeme. This is in preparation for building parse trees.