CSE304 : Compiler Design : Parser

Phase III : Parser

Due: 12-May-2019

Overview

The third phase of developing a compiler for ***rascl*** is writing a parser to analyze the syntax of the program.

Parser Architecture

The parser drives the entire compilation process. So the parser should have a main routine that accepts from the command line a single argument (an input file name containing the program to be compiled). It will call initialization functions for the other components in the compiler (*initSymTab()*, *initLexer()*, etc)

Functions

The functions will not be specified by this assignment. There is a main function that drives the entire process. The functions you implement depends on what type of parser you write. There is no real API to a parser. It is up to you whether you implement a bottom up or top down parser (but it must not be generated by a tool (unless you write the generator tool! ;-)))

Personally, I recommend a predictive recursive decent parser (but not table driven). This is one of the easiest parsers to write. Any table driven parser (top-down or bottom-up) requires the generation of tables based on some complex set operations as you've seen in class.
In a table driven predictive LL parser (once the tables are generated), a stack is maintained containing symbols (terminals and non terminals) derived from the grammar's productions. The code is relatively small as it is a loop that processes input and the data on the stack. When a non-terminal is present, the next input symbol is used to decide on the production to use. The symbols in the production are pushed on to the stack in reverse order. When the top of stack symbol is a terminal, it is matched with the next input symbol. If the terminal does not match the symbol, the parser reports an error.

A simple predictive recursive decent parser involves writing one function for each non-terminal (not each production). There are no tables to generate. The code you will write is much longer but also much easier to derive. The function for a particular non-terminal checks the next token in the token stream and uses that to decide which production to use for the non-terminal. The code inside of the function progressively tries to match a terminal from the production with the next token from the stream. If the next symbol in a production is a non terminal, it calls the function for that non-terminal. Functions for non terminals with multiple productions decide on the correct production with the look ahead symbol. If the look ahead symbol does not indicate a valid production for the current non-terminal function, the parser reports a failure.

Recall that recursive decent (or any top-down) parser cannot handle left recursion. We also have 'factor' out any common prefixes by rewriting productions containing those common prefixes so the parser can uniquely identify a production to use based only on a single look ahead symbol. The grammar I give for Rascl below is written for top-down parsing so I've eliminated common prefixes and left recursion.

Should you write a bottom-up parser, you may have to rewrite parts of the grammar.

Grammar

RASCL (Really, a Small C Language) is a small subset of C. It is structured without functions at all. It also separates declarations from executable code in two different lists of statements.

Below is the grammar for RASCL. We provide a list of terminals first. These should be recognized by your lexical analyzer. 'program' is the start symbol for the language. DD represents 'end of input'.

TERMINALS

SEMICOLON ";"
LBRACE "{"
RBRACE "}"
COMMA ","
INT "int"
FLOAT "float"
ID <identifier>
ICONST <integer constant>
FCONST <floating point constant>
LBRACKET "["
RBRACKET “]"
PRINT "print"
ASSIGN "="
LPAREN "("
RPAREN ")"
MULT "*"
DIV "/"

PLUS "+"
MINUS "-"
WHILE "while"
IF "if"
ELSE "else"
NOT "!"
AND "&&"
OR "||"
EQUAL "=="
LT "<"
GT ">"
DD <end of input>

Productions

program -> decllist bstatementlist DD
decllist -> decl decllist
decllist -> ε
bstatementlist -> LBRACE statementlist RBRACE
statementlist -> statement SEMICOLON statementlist
statementlist -> ε
decl -> typespec variablelist
variablelist -> variable variablelisttail
variablelisttail -> COMMA variable variablelisttail
variablelisttail -> SEMICOLON
typespec -> INT
typespec -> FLOAT
variable -> ID variabletail
variabletail -> arraydim
variabletail -> ε
arraydim -> LBRACKET arraydimtail
arraydimtail -> ID RBRACKET
arraydimtail -> ICONST RBRACKET
statement -> whilestatement
statement -> ifstatement
statement -> assignmentexpression
statement -> printexpression
statement -> readstatement
assignmentexpression -> variable ASSIGN otherexpression
printexpression -> PRINT variable
otherexpression -> term otherexpressiontail
otherexpressiontail -> PLUS term otherexpressiontail
otherexpressiontail -> MINUS term otherexpressiontail
otherexpressiontail -> ε
term -> factor termtail
termtail -> MULT factor termtail
termtail -> DIV factor termtail
termtail -> ε

factor -> variable
factor -> ICONST
factor -> FCONST
factor -> LPAREN otherexpression RPAREN
factor -> MINUS factor
whilestatement -> WHILE condexpr whiletail
whiletail -> compoundstatement
compoundstatement -> LBRACE statementlist RBRACE
ifstatement -> IF condexpr compoundstatement istail
istail -> ELSE compoundstatement
istail -> ε
condexpr -> LPAREN vorc condexprtail RPAREN
condexpr -> vorc condexprtail
condexpr -> NOT condexpr
condexprtail -> LT vorc
condexprtail -> GT vorc
condexprtail -> EQUAL vorc
vorc -> variable
vorc -> ICONST
vorc -> FCONST
printstatement -> PRINT variable
readstatement -> READ variable

Testing

I will provide a moderate set of test code for you to compile. I may use additional tests when I grade your parser. Here are a couple of short examples of what RASCL code looks like:

=====
T00_rascl_test_exprs1.rsc
=====

```
int a, b;
float c, d;
{
  a = 5;
  b = a + 2 * (-b + -a);
}
```

=====
T41_rascl_test_io.rsc
=====

```
float a;
float b;

{
   a = 3.7;
   b = a * 2.0;
   print b;
}
```

=====
T62_rascl_test_all_features3.rsc
=====

```
int b, c;
int d;
float e;
float f;

{
 b = 1;
 c = 10;
 e = 5.0;
 f = e * c;
 if (c > b)
 {
   d = 5;
   c = c + -b;
 }
 else
 {
   while (b < 5) {
       c = -d * b;
          b = b + 1;
   };
 };
 print c;
}
```

The minimal set of test cases is attached to the dropbox as the file *basic_rascl_tests.zip.*
Again, the final compiler will have additional test cases.

Test Output: Parser Only

Since there is no intermediate code generation at this point, I will specify some output that will demonstrate the parser is working. The parser should indicate actions it takes at key points in the parse operation. For a top down parser, the code is looking at a token and determining which production to use. For each production expansion, write the production used.

For example, the parser is attempting to parse a *statement*. The next token is a *while* keyword. The parser should print:

```
statement -> whilestatement
whilestatement -> WHILE condexpr whiletail
```

After matching the *while*, if the next token is '(' (LPAREN). Your code should print:

```
condexpr -> LPAREN vorc condexprtail RPAREN
```

You should capture this output in a text file named according to each specific test. So, for *T60_rascl_test_all_features1.rsc,* write the production messages to *T60_rascl_test_all_features1.rsc.log.*

Submission Process:

The deliverables for this part of the project are:

1. Source code to the parser
2. Source code for the revised lexical analyzer (or original submission if it worked properly).
3. A script that builds and links the code (if it is in a compiled language like C).
4. A script that takes a file name of a file to compile from its command line and runs the parser giving it that source (.rsc) file.
5. A log file for each test compiled.

Zip (or tar/gzip) the source code, the scripts, and the log files for each .rsc file in the test suite provided.

Upload the ZIP or .tgz file to the *'Phase III : Parser'* dropbox.

***** Late Assignments will NOT be accepted, and any assignments emailed to the professor will also NOT be accepted. Start working early and plan to submit working code on time. *****