# CSE304 : Compiler Design : Parser

*Phase III : Parser*

Due: 2-May-2025

**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'.

There are a few interesting things you should notice but may not be obvious from the productions below:

1. The last statement in a statement list should NOT have a semicolon at the end.
2. If a function does not return a value (is 'void') and you simply wish to call it from a separate statement, you must precede the invocation with the 'call' keyword. This helps the language distinguish between an assignment statement and a standalone function call.
3. Array indices CANNOT be full arithmetic expressions as they can be in some languages. For declarations, they must be integer constants. For assignment statements and use of values, they must be an integer constant or an identifier.

I may discuss some other peculiarities in class.

**Important Note: Looking at the token list below, you may notice I neglected to ask you to have your lexer recognize certain keywords. You will have to modify your lexical analyzer to add those keywords [call, return] (it should be easy to do this).**

## TERMINALS

```
SEMICOLON==>";"
LBRACE==>"{"
RBRACE==>"}"
COMMA==>","
INT==>"int"
FLOAT==>"float"
VOID==>"void"
ID==>"[a-zA-Z][a-zA-Z0-9]*"
FUNCTION==>"function"
ICONST==>"\d+"
FCONST==>"\d+(.\d*)?"
LBRACKET==>"["
RBRACKET==>"]"
PRINT==>"print"
READ==>"read"
ASSIGN==>"="
LPAREN==>"("
RPAREN==>")"
MULT==>"*"
DIV==>"/"
PLUS==>"+"
MINUS==>"-"
WHILE==>"while"
IF==>"if"
ELSE==>"else"
RETURN==>"return"
CALL==>"call"
NOT==>"!"
AND==>"&&"
OR==>"||"
EQUAL==>"=="
LT==>"<"
LE==>"<="
GT==>">"
GE==>">="
NOTEQUAL==>"!="
DD==>"$$"
```

## Productions

Following are the productions for the *rascl* grammar. They are designed for top down predictive parsing.

Start symbol: **Program**

Program => decllist funcdecls DD

funcdecls => funcdecl funcdecls
funcdecls => maindecl
funcdecl => FUNCTION ftypespec simplevar fdeclparms LBRACE dec  llist statementlist RBRACE
maindecl => main LPAREN RPAREN LBRACE decllist statementlist R  BRACE
ftypespec => VOID | INT | FLOAT
simplevar => ID
fdeclparms => LPAREN fparmlist RPAREN
fparmlist => fparm fparmlistrem
fparmlist => e
fparm => typespec parmVar
parmVar => ID parmVarTail
parmVarTail => LBRACKET RBRACKET parmVarTail
parmVarTail => e
fparmlistrem => COMMA fparm fparmlistrem
fparmlistrem => e
decllist => decl decllist
decllist => e
bstatementlist => LBRACE statementlist RBRACE
statementlist => statement SEMICOLON statementlist
statementlist => e

decl => typespec variablelist

variablelist => variable variablelisttail
variablelisttail => COMMA variable variablelisttail
variablelisttail => SEMICOLON
variable => ID variabletail
variabletail => LBRACKET ICONST RBRACKET variabletail
variabletail => e

typespec => INT
typespec => FLOAT

usevariable => ID usevariabletail
usevariabletail => arraydim
usevariabletail => e

arraydim => LBRACKET arraydimtail
arraydimtail => Iconst RBRACKET arraydim

arraydimtail => ID RBRACKET arraydim
arraydim => e

statement => whilestatement
statement => ifstatement
statement => assignmentstatement
statement => printstatement
statement => readstatement
statement => returnstatement
statement => callstatement
basicexpr => basicterm basicexprtail
basicexprtail => PLUS basicterm basicexprtail
basicexprtail => MINUS basicterm basicexprtail
basicexprtail => e
basicterm => basicfactor basictermtail
basictermtail => MULT basicfactor basictermtail
basictermtail => DIV basicfactor basictermtail
basictermtail => e

basicfactor => ID
basicfactor => ICONST

assignmentstatement => usevariable ASSIGN otherexpression
otherexpression => term otherexpressiontail
otherexpressiontail => PLUS term otherexpressiontail
otherexpressiontail => MINUS term otherexpressiontail
otherexpressiontail => e
term => factor termtail
termtail => MULT factor termtail
termtail => DIV factor termtail
termtail => e
factortail => usevariabletail
factortail => funccalltail

funccalltail => LPAREN arglist RPAREN
arglist => otherexpression arglistrem
arglist => e
arglistrem => COMMA otherexpression arglistrem
arglistrem => e

factor => ID factortail
factor => ICONST
factor => FCONST
factor => LPAREN otherexpression RPAREN
factor => MINUS factor

whilestatement => WHILE relationalexpr bstatementlist
ifstatement => IF relationalexpr bstatementlist istail
istail => ELSE bstatementlist
istail => e

relationalexpr => condexpr relationalexprtail
relationalexprtail => AND condexpr
relationalexprtail => OR condexpr
relationalexprtail => e

condexpr => LPAREN otherexpression condexprtail RPAREN
condexpr => otherexpression condexprtail
condexpr => NOT condexpr
condexprtail => LT otherexpression
condexprtail => LE otherexpression
condexprtail => GT otherexpression
condexprtail => GE otherexpression
condexprtail => EQUAL otherexpression
printstatement => PRINT otherexpression
readstatement => READ usevariable
returnstatement => RETURN otherexpression
returnstatement => RETURN
callstatement => CALL Ident funccalltail

## Integration Notes

For now, your symbol manager routines are NOT needed. Only the lexical analyzer from assignment 1 must be integrated with your parser. In the next assignment, you will be integrating the symbol manager as well as adding functionality to do semantic checks (code meaning) and generating some intermediate form of code.

## Testing

With this writeup, I provided a collection of sample test files in *parser_tests.zip*. There are two folders: basic_rascl and basic_rascl_output. The first has 21 test input files. The latter has a subset of outputs from some of the input files (7 or 8). This should provide enough guidance as to what your parser should output. I'll have notes below on the form of output it should generate but the output from this assignment is not needed for the final project and should be designed in a way that it can easily be disabled.

For this submission, your parser (or a test main program written to call the parser) should accept a filename from the command line. This way, I can easily run it and provide different test files each time. It can send output to the console (I will redirect that to an output file.) Since the remaining assignment(s) will mainly be adding functionality called from that parser routines (as the parser drives the compilation process), this will likely be the last set of external tweaks that are needed.

**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 something like:

**whilestatement => WHILE relationalexpr bstatementlist**

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

**relationalexpr => condexpr relationalexprtail**
**condexpr => L_PAREN condexpr condexprtail R_PAREN**
**condexpr => otherexpression condexprtail**
**otherexpression => term otherepxressiontail**

You should capture your parser's 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.rsp (rascal productions).*

Two test files have syntax errors to simply verify your parser correctly identifies a couple of basic problems. For error handling, you should simply report the error and that the parse failed.

**Finally**

If you have any questions about the required output or operation of the parser, please ask via email or come to my office hours. The assignment writeup is as specific as I can get without producing a book that will take a week to read!

**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.
3. A Dockerfile that builds an image including your source code and then loads any tools that will be needed to build and run the parser. The Dockerfile should also include the creation of two directories (/app/tests and /app/output). I will use those to connect to folders on my machine where I can read the test inputs and write the test outputs. The Dockerfile should also copy in a script that builds and links the code (if it is in a compiled language like C).
4. In addition to your code, Dockerfile, etc, please add a README.txt file that describes how to build and run your code.

**Now…**

1. Place the deliverable source files for your parser and lexer (which may include a Docker file to create the required virtual container) into a folder by themselves (if you wish, you can place the source code in a folder called src below this top-level folder). The folder's name should be CSE304_Phase3_<yourname>_<yourid>. So, if your name is Amy Kim and your id is 12345678, the folder should be named 'CSE304_Phase3_AmyKim_12345678.

2. Compress the folder and submit the zip file.
    a. On Windows, do this by clicking the right-mouse button while hovering over the folder. Select 'Send to -> Compressed (zipped) folder'. The compressed folder will have the same name with a .zip extension. You will upload that file to the Brightspace.
    b. On Mac, move the mouse over the folder then right-click (or for single button mouse, use Control-click) and select **Compress**. There should now be a file with the same name and a .zip extension. You will upload that file to the Brightspace.
3. Navigate to the course Brightspace site. Click **Assignments** in the top navbar menu. Look under the category 'CompilerProject'. Click **Compiler_Phase3.**
    a. Scroll down and under **Submit Assignment**, click the **Add a File** button**.**
    b. Click **My Computer** (first item in list).
    c. Find the zip file and drag it to the 'Upload' area of the presented dialog box.
    d. Click the **Add** button in the dialog.
    e. You may write comments in the comment box at the bottom.
    f. Click **Submit**. ← Be sure to do this so I can retrieve the submission!

***** 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. *****