CS304 : Compiler Design : Lexical Analyzer

Phase I : Lexical Analysis

Due: 23-Mar-2025

Overview

The first phase of developing a compiler for 'rascl' is writing a lexical analyzer. The analyzer should support the following tokens:

- Punctuation:
  - Semicolon (';')
  - LeftParen ('(')
  - RightParen (')')
  - Comma (',')
  - LeftBrace ('{')
  - RightBrace ('}')
  - LeftBracket ('[')
  - RightBracket (']')
- Operators:
  - Assign ('=')
  - LessThan ('<')
  - LessThanOrEqual ('<=')
  - GreaterThan ('>')
  - GreaterThanOrEqual ('>=')
  - Equal ('==')
  - NotEqual ('!=')
  - Add ('+')
  - Subtract ('-')
  - Multiply ('*')
  - Divide ('/')
  - Not ('!')
  - And ('&&')
  - Or ('||')
- Keywords:
  - If ('if')
  - Else ('else')
  - While ('while')
  - Integer ('int')
  - Float ('float')
  - Void ('void')
  - Call ('call')
  - Print ('print')
  - Read ('read')

    - Function (‘function’)
- Identifiers: Names following C lexical rules
- Integer constants: Whole numbers following C rules
- Floating constants: Numbers with decimal parts following C rules
- Comments: Strings enclosed between ‘/*’ and ‘*/’ : Note that for comments, no token should be returned. Simply scan past the text and then start over scanning for another token.

The lexical analyzer will have to read the source file which will be provided to it in an initialize call. You may wish to write a small file handler library to do buffering of the source file and provide characters to the lexical analyzer as needed.
Using the techniques from the textbook and taught in class, create a lexical analyzer that can read and provide information on any of the above tokens. Write your lexical analyzer manually. Do not use tools like lex.
You may write your lexical analyzer in almost any modern language (see my instructions in the project overview.) I can build and run your code. Check with me first on languages, or better, use Docker which I will briefly introduce in class. I strongly recommend using Docker as it will almost assure nearly problem free testing of your code. I hope to give a small intro to Docker in class but that may not happen for about 2 weeks. In the meantime, here is a free hour lecture on Docker that will give you everything you need for your project submissions: https://www.youtube.com/watch?v=pTFZFxd4hOI

Docker allows you to create a thin virtual environment so you can make available the toolchain required to build and run your code. Note: for your base Docker image, please choose a unix based environment (like ubuntu) rather than windows. Even with Docker, Windows base images take up a large amount of storage.

I will prove a collection of rascl source files on which you can test your code.

Lexical Analyzer API

Your code will need to provide some methods to allow it to be initialized and to request a token. This interface will be used by the parser when you write that in a later phase. Here are some suggestions for organizing the data and entry points (methods) you should provide. These are all 'notional' (so just an example) since I do not know which source language you will use to implement your compiler.

The code should provide an API similar to the following:

1. A type to represent token type values. This can be a collection of named integer constants or an enumerated type (depending on the implementation language you choose). As an **example**, if you use C, you may define these constants with:

```
enum tokens {SEMICOLON, LPAREN, RPAREN, COMMA, LBRACE,
        RBRACE, LBRACKET, RBRACKET, ASSIGN, LT, LE, GT, GE, EQ, NE,
        PLUS, MINUS, MULT, DIV, NOT, AND, OR, IF, ELSE, WHILE, INT,
        FLOAT, VOID, CALL, ID, ICONST, FCONST, READ, PRINT,
        FUNCTION };
```

Note the above is 'notional', an idea. It can be implemented in other ways, if desired and, of course, would differ if you are using a language other than C. You can list the token names in any order you wish.

2. A type to hold information about the token scanned (to be returned to the caller [parser]). This will be some type of struct, class, or object. For now, it can just hold the token found and a copy of the text that was matched. You can use a struct in C something like the following:

```
struct tokenInfo {
   enum tokens token;
   char tokenText[max_token_length];
   // for now, this struct is enough. Later, it may need additional fields
    // added to communicate more information to the parser or
    // semantic analysis code (like source line number and
    // source starting character for reporting errors, conversions
    //  of text for integer and floating point literals, etc.)
}
```

If you use a language like Python, you can do the same thing with a dictionary that had keys named like the fields in the struct above.

Following are the required API calls that the lexer must support (given as C prototypes but, again, *you can use any implementation language that is available to me*). There are only 2 methods required here as an external API.

initLexer

In C, this would be declared similar to the following:

```
int initLexer(char *filename);
```

This function accepts a string (or pointer to a character array as shown in C). This is the filename with the source code to be analyzed. It opens the file and creates a buffer for the contents of the file to be compiled (or uses a small i/o library you write to do file operations and buffering). It returns the integer 1 (true) if the file open succeeded or 0 (false) otherwise. You can return a 'boolean' if you implement your compiler in a language that supports that type.

getNextToken

In C, this would be declared:

```
struct tokeninfo *getNextToken(void);
```

This method will scan the text and return information for the next token it recognizes. It does this by returning a structure holding the information about the token (struct tokeninfo shown above or a dictionary if you are using Python, etc)
Remember, the scanner should be 'greedy'. It should match the longest string of characters possible to avoid confusion and parse errors.

Testing

You should provide a short test main program that will:

- Read a filename from the user
- Call the initLexer() function for the lexical analyzer passing the filename given
- Repeatedly call **getnexttoken**() until the analyzer indicates EOF has been reached
- For each call, print a text representation of the token returned and the text scanned for that token.

As an example, if the file entered contains the following fragment:

if (a < 5 && b > 10.5) { print a; }

it should produce the following output on the console (note that the scanned text is placed between vertical bars to clearly indicate what text was scanned.):

token: T_K_If : |if|
token: T_L_Paren : |(|
token: T_Ident : |a|
token: T_Less : |<|
token: T_Iconst : |5|
token: T_And : |&&|
token: T_Ident : |b|

token: T_Greater : |>|
token: T_Fconst : |10.5|
token: T_R_Paren : |)|
token: T_L_Brace : |{|
token: T_K_Print : |print|
token: T_Ident : |a|
token: T_SemiColon : |;|
token: T_R_Brace : |}|
token: T_EOF : ||

Submission Process:

Please follow this procedure for submission:

0. In addition to your code, please add a README.txt file that describes how to build and run your code.
1. Place the deliverable source files for your lexical analyzer (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_Phase1_<yourname>_<yourid>. So, if your name is Betty Lee and your id is 12345678, the folder should be named 'CSE304_Phase1_BettyLee_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_Phase1.**
    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!