# LEXICAL ANALYSIS PROJECT

GODFREY MUGANDA

## 1. C minus minus

The grammar for the C minus minus programming language is as follows.

PROGRAM ::= VARS **begin** STMTLIST **end**
VARS ::= { VARDECLIST }
VARDECLIST ::= TYPEID id { , id };
TYPEID ::= **int**
STMTLIST ::= { STMT; }
STMT ::= ASSIGNSTMT | OUTPUTSTMT | INPUTSTMT
STMT ::= IFSTMT | WHILESTMT | LOOPSTMT
STMT ::= **break** | **continue**
ASSIGNSTMT ::= id = EXPR
OUTPUTSTMT ::= **cout** << OUTPUTEXPR { << OUTPUTEXPR }
INPUTSTMT ::= **cin** >> id { >> id }
IFSTMT ::= **if** EXPR **then** STMTLIST **end if**
IFSTMT ::= **if** EXPR **then** STMTLIST **else** STMTLIST **end if**
WHILESTMT ::= **while** EXPR **do** STMTLIST **end while**
LOOPSTMT ::= **loop** STMTLIST **end loop**
OUTPUTEXPR ::= EXPR | **string**
EXPR ::= COMPEXPR { RELOP COMPEXPR }
COMPEXPR ::= SIMPLEEXPR { ADDOP SIMPLEEXPR }
SIMPLEEXPR ::= FACTOR { MULTOP FACTOR }
FACTOR :: = number | id | (EXPR) | !FACTOR | - FACTOR
RELOP ::= < | <= | > | >= | != | ==
ADDOP ::= + | −
MULTOP ::= ∗ | / | %

The tokens of this language are the keywords (also called reserved words) **begin**, **break**, **cin**, **continue**, **cout**, **end**, **if**, **int**, **loop**, and **while**. Each of these tokens correspond to a single string of characters known as the *lexeme* for the token. For example, the string `"cout"` corresponds to the **cout** token.

Closely related to the keywords is the identifier token **id**. The **id** token can be specified by many different strings, according to the lexical rules of the language. In C minus minus, a lexeme for an identifier is a sequence of characters consisting of alphanumeric characters and underscores, but which cannot start with a numeric character. For example, the following are legal lexemes for identifiers:

```
_
_1
x
x17y
this_is_an_id
```

There is also the **string** token, which can be specified by many different lexemes. A string is a sequence of characters delimited by double quotes.

In addition, there are various tokens that correspond to operators and punctuation symbols. These are operators tokens

```
+    -    *    /    %     !
```

the relational operators

```
<    <=     >     >=    !=     ==
```

the assignment operator =; and the punctuation symbols

```
,   ;   (   )
```

for the comma, semicolon, left parenthesis (, and right parenthesis ).

## 2. The Lexical Analyzer Project

The purpose of the lexical analyzer is to two-fold:

(1) Open the source file and turn the stream of characters in the source file into a stream of token,
(2) Generate a listing of the source file interspersed with error messages found in the source file by the compiler.

A *token* is the smallest building block of the language. To represent tokens, we will use the enumeration type `Token_Type`, which is declared in a header file `"lex.h"`.

```
enum class Token_Type
{
    // keywords
    t_begin, t_break, t_cin, t_continue, t_cout, t_end, t_if, t_int,
    t_loop, t_while,
    // identifier, number, and string tokens
    t_id, t_number, t_string,
    // various operators
    t_plus, t_minus, t_mult, t_div, t_mod, t_assign, t_not,
    // relational operators
    t_lt, t_le, t_gt, t_ge, t_ne, t_eq,
    // various punctuation symbols
    t_comma, t_semi, t_lparen, t_rparen,
    // unknown and eof tokens
    t_unknown, t_eof
};
```

Think of a token as an internal representation for the building blocks of the language.

The string of characters from the source file that forms the token is called the *lexeme* for that token.

The lexical analyzer header file also contains declarations of the following vector and map. You will see their definitions later in the `"lex.cpp"` implementation file.

```
extern  const vector < string > token_tostring;
extern const map<string, Token_Type> keywords_map;
```

The lexical analyzer is an object of the class `Lex`;

```
class Lex
{
public:
  Token_Type get_token();

  string get_lexeme() { return lexeme; }

  string token_stringfy(Token_Type t)
  {
    return token_tostring[static_cast<int> (t)];
  }

  void fatal_error(const string &message)
  {
    listing_file << "*** Error *** " << message << endl;
    exit(1);
  }
  // Constructor
  Lex(const string &source_filename, const string& listing_filename);
  // Destructor
  ~Lex();

private:
  // Get next character from file and echo to listing file
  char get_char()
  {
    char ch = source_file.get();
    if (ch != -1)
    {
      listing_file.put(ch);
    }
    return ch;
  }
  // various member variables
  string lexeme;
  ifstream source_file;
  ofstream listing_file;
  // string source_filename, listing_filename;
};
```

The main members of the `Lex` class are

(1) `Token_Type get_token()` gets and returns the next token available from the source file.
(2) `string get_lexeme() { return lexeme; }` returns the lexeme corresponding to the token returned by the last call to `get_token()`.
(3) `string token_stringfy(Token_Type t)`: returns the stringfied version of a token.
(4) `void fatal_error(const string &message)`: outputs an error message to the listing file and terminates the program.

## 3. The lex.cpp Implementation file

This file will contain all variables and member functions declared in the header file but not implemented there.

```
// Note: order of the vector entries is tied to order of Token_Type constants!
const vector < string > token_tostring
{
  "t_begin", "t_break", "t_cin", "t_continue", "t_cout", "t_end", "t_if",
  "t_int", "t_loop", "t_while",
  "t_id", "t_number", "t_string", "t_plus", "t_minus", "t_mult", "t_div",
  "t_mod", "t_assign", "t_not",
  "t_lt", "t_le", "t_gt", "t_ge", "t_ne", "t_eq",
  "t_comma", "t_semi", "t_lparen", "t_rparen",
  "t_unknown", "t_eof",
};

const map<string, Token_Type> keywords_map
{
  { "begin", Token_Type::t_begin},
  { "break", Token_Type::t_break},
  { "cin", Token_Type::t_cin},
  { "continue", Token_Type::t_continue},
  { "cout", Token_Type::t_cout},
  { "end", Token_Type::t_end},
  { "if", Token_Type::t_if},
  { "int", Token_Type::t_int},
  { "loop", Token_Type::t_loop},
  { "while", Token_Type::t_while}
};
```

The vector lists all the strings that correspond to the tokens, while the map associates lexemes for the keywords with corresponding tokens.

The skeletons for the rest of the implementation file is as follows

```
#include "lex.h"
#include <fstream>
#include <iostream>
#include <cctype>

using namespace std;

Token_Type Lex::get_token()
{
  static char ch = ' ';
  lexeme = "";

}

Lex::Lex(const string &source_filename, const string& listing_filename)
{
  this->source_file.open(source_filename);
  this->listing_file.open(listing_filename);
```

```
  if (!source_file)
  {
    cout << "Cannot open the file " << source_filename << endl;
    exit(1);
  }
  if (!listing_file)
  {
    cout << "Cannot open the file " << listing_filename << endl;
    exit(1);
  }
}

Lex::~Lex()
{
  source_file.close();
  listing_file.close();
}
```

Note that the constructor and destructor are fully implemented and do not have to be changed.

## 4. Sample Input/Output pairs

To test the lexical analyzer, we use a main method that creates a lexical analyzer object, and then calls the get_token() method repeatedly until all tokens in the source file have been fetched. Each token that is retrieved is printed to standard output, together with the lexeme from which it is formed:

```
#include "lex.h"
#include <iostream>

using namespace std;

int main(int argc, char * argv[])
{
  Lex lex{"test2.txt", "test2.lst"};

  Token_Type token = lex.get_token();
  while (token != Token_Type::t_eof)
  {
    cout << lex.get_lexeme() << ":  " << lex.token_stringfy(token) << endl;
    token = lex.get_token();
  }

  return 0;
}
```

Two sample input files are provided to help you test your lexical analyzer. The sample file `test1.txt` has contents

```
begin break cin continue cout end if int loop while
id  _id id12_ id_12  _
12  -34
```

```
+ = * / %
< <=  >  >=  != ==

! !!  // This is a comment
/* This is another
   comment but // this is not a comment*/12
, ; ( ) &
"This is a string"
```

The output from your program should be

```
begin:  t_begin
break:  t_break
cin:  t_cin
continue:  t_continue
cout:  t_cout
end:  t_end
if:  t_if
int:  t_int
loop:  t_loop
while:  t_while
id:  t_id
_id:  t_id
id12_:  t_id
id_12:  t_id
_:  t_id
12:  t_number
-:  t_minus
34:  t_number
+:  t_plus
=:  t_assign
*:  t_mult
/:  t_div
%:  t_mod
<:  t_lt
<=:  t_le
>:  t_gt
>=:  t_ge
!=:  t_ne
==:  t_eq
!:  t_not
!:  t_not
!:  t_not
12:  t_number
,:  t_comma
;:  t_semi
(:  t_lparen
):  t_rparen
&:  t_unknown
This is a string:  t_string

RUN SUCCESSFUL (total time: 52ms)
```

The second file, `test2.txt`, has contents

```
begin break cin continue cout end if int loop while
id  _id id12_ id_12  _
12  -34
+ = * / %
< <=  >  >=  != ==

! !!  / This is a comment
/* This is another
   comment but // this is not a comment/12
, ; ( ) & /
"This is a string"
```

This file has an error: the multi-line comment is never closed, the source file ends before the finding the end of the comment. This is an error. The output for this file should be

```
begin:  t_begin
break:  t_break
cin:  t_cin
continue:  t_continue
cout:  t_cout
end:  t_end
if:  t_if
int:  t_int
loop:  t_loop
while:  t_while
id:  t_id
_id:  t_id
id12_:  t_id
id_12:  t_id
_:  t_id
12:  t_number
-:  t_minus
34:  t_number
+:  t_plus
=:  t_assign
*:  t_mult
/:  t_div
%:  t_mod
<:  t_lt
<=:  t_le
>:  t_gt
>=:  t_ge
!=:  t_ne
==:  t_eq
!:  t_not
!:  t_not
!:  t_not
/:  t_div
```

```
This:  t_id
is:  t_id
a:  t_id
comment:  t_id

RUN FAILED (exit value 1, total time: 54ms)
```

For this program, the listing file will record the error:

```
begin break cin continue cout end if int loop while
id  _id id12_ id_12  _
12  -34
+ = * / %
< <=  >  >=  != ==

! !!  / This is a comment
/* This is another
   comment but // this is not a comment/12
, ; ( ) & /
"This is a string"
*** Error *** Program ended while scanning multi-line comment.
```

## 5. Miscellaneous Information

The C minus minus language supports the same commenting conventions as C++ and Java. A comment that starts with // ends at the end of line. The lexical analyzer should output an error message to the listing file and exit if the source file ends before a multi-line comment has ended, or before a string has ended. Strings are delimited by double quotes.

## 6. Due date

This is due Wednesday of Week 4 at midnight. To submit, zip up the entire project folder and attach to an email with subject: Lexical Project.