PARSER PROJECT
THE C MINUS MINUS PROGRAMMING LANGUAGE

GODFREY MUGANDA
DEPARTMENT OF COMPUTER SCIENCE
NORTH CENTRAL COLLEGE

1. Project Overview

This is phase 2 of the compiler/interpreter development project. In this phase, you build a parser that uses the lexical analyzer developed in the first project to determine the syntax structure of a source program.

2. Language Definition

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**, **do**, **else**, **end**, **if**, **int**, **loop**,**then**, 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 I/O symbols

```
<<    >>
```

for insertion and extraction; the assignment operator =; and the punctuation symbols

```
,   ;   (   )
```

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

3. Modification to the Lexical Analyzer Project

Modify the lexical analyzer so that it recognizes additional tokens: the keywords **do**, **else**, and `then`; and the C++ -like operators for insertion << and extraction >>. Your class should look like this (Note the changes to the error-reporting functions)

In addition to the new tokens, we have changed the error-reporting functions. The `error()` method no longer terminates the program: it will be called by the parser. Your lexical analyzer should now call the `fatal_error()` function when a fatal error occurs.

```
enum class Token_Type
{
  // keywords
  t_begin, t_break, t_cin, t_continue, t_cout, t_do, t_else, t_end, t_if,
  t_int, t_loop, t_then, 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,
  // io operators
  t_insertion, t_extraction,
  // various punctuation symbols
  t_comma, t_semi, t_lparen, t_rparen,
  // unknown and eof tokens
  t_unknown, t_eof
};

// These establish the correspondence between tokens and
// the stringfyed versions of those tokens, for example
// the t_begin token corresponds to the string "t_begin"
extern const vector < string > token_tostring;
extern const map<string, Token_Type> keywords_map;

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 error(const string &message)
  {
    listing_file << "*** Error *** " << message << endl;
  }

  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;
};
```

You can also modify the lexical analyzer by making the following previously global variables member variables of the `Lex` class:

```
// Note: order of the vector entries is tied to order of
// Token_Type constants!
vector < string > token_tostring
{
    "t_begin", "t_break", "t_cin", "t_continue", "t_cout",
    "t_do", "t_else", "t_end", "t_if",
    "t_int", "t_loop", "t_then", "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_insertion", "t_extraction",
    "t_comma", "t_semi", "t_lparen", "t_rparen",
    "t_unknown", "t_eof",
};

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},
    { "do", Token_Type::t_do},
    { "else", Token_Type::t_else},
    { "end", Token_Type::t_end},
    { "if", Token_Type::t_if},
    { "int", Token_Type::t_int},
    { "loop", Token_Type::t_loop},
    { "then", Token_Type::t_then},
    { "while", Token_Type::t_while}
};
```

4. Organization of Project Files

You need to add a `parser.h` file that gives the specification of the parser class, and `parser.cpp` file that implements the various parsing functions. Here is an outline of the `parser.h` file:

```
#include <cstdlib>
#include <cassert>
#include "lex.h"
#include <set>


class Parser
{
public:
  Parser(const string &source_filename, const string& listing_filename)
  :
  lex{source_filename, listing_filename}
  {
    // get the lookahead token
    token = lex.get_token();
  }
  void program();
private:
  // Parsing methods for non-terminals
  void  vars(); void vardeclist();
  void type_id();  void stmtlist(); void stmt();
  void assignstmt(); void outputstmt(); void inputstmt();
  void ifstmt(); void whilestmt(); void loopstmt();
  void output_expr(); void expr(); void comp_expr();
  void simple_expr();  void factor();
  // Various utilities for token classification
  bool is_relop(Token_Type t)
     { return relops.find(t) != relops.end();}
  bool is_multop(Token_Type t)
     { return multops.find(t) != multops.end();}
  bool is_addop(Token_Type t)
     { return addops.find(t) != addops.end();}
  bool is_stmt_begin(Token_Type t)
     { return stmt_begin.find(t) != stmt_begin.end();}
  bool is_expr_begin(Token_Type t)
     { return expr_begin.find(t) != expr_begin.end();}
  // Recognize an expected token
  void accept(Token_Type t);
  // Various member variables
  Lex lex;                  // lexical analyzer
  Token_Type token;         // lookahead token

  // Interface to lexical analyzer
  inline Token_Type get_token() { return lex.get_token(); }
  inline string get_lexeme() { return lex.get_lexeme();}
  void  parser_fatal_error(const string &message)
  {
```

```
      lex.error(message);
      exit(1);
  }
  // Error message for when an unexpected token is found.
  void syntax_error(Token_Type found,
                    initializer_list<Token_Type> expected);
  // Generic error message
  void syntax_error(const string &message)
  {
      parser_fatal_error(message);
  }

  // member variables for token classification
  set<Token_Type> stmt_begin   // start symbols for STMT
  {
    // Fill this in
  };
  set<Token_Type> expr_begin   // start symbols for expressions
  {
    // Fill this in
  };
  set<Token_Type> relops
  {
    // Fill this in
  };
  set<Token_Type> addops
  {
    Token_Type::t_plus, Token_Type::t_minus,
  };
  set<Token_Type> multops
  {
    Token_Type::t_mult, Token_Type::t_div, Token_Type::t_mod,
  };
};
```

5. Input Output

The input to the parser will be `.cmm` source file containing a cmm program. The output will be a listing file that is a copy of the source file up to the occurrence of the first fatal error. (We will encounter non-fatal errors in the next project).

If there are no errors, then the listing file will be an exact copy of the source file. However, if there is an error, the listing file will be a copy of the source file up to the location of the error, followed by an error message.

6. Syntax Error Reporting

An error that will occur frequently, is when the parser is expecting one or more of a set of tokens, but the lookahead token in none of them. For example, when expecting a statement (STMT) you expect the lookahead token to be one of the symbols that begin a statement.

To report such an error, call the parser member function

```
void syntax_error(Token_Type found,
                  initializer_list<Token_Type> expected);
```

passing the lookahead token in place of `found`, and an initializer list of statement begin symbols in place of the `expected` parameter.

We have not yet discussed initializer lists, so here is how that function should be implemented:

```
void Parser::
syntax_error(Token_Type found, initializer_list<Token_Type> expected)
{
    assert(expected.size() != 0);
    ostringstream message{};
    message << "Expected ";
    if (expected.size() > 1)
    {
        message << " one of : ";
    }
    for (auto p = expected.begin();  ; )
    {
        message << lex.token_stringfy(*p);
        p++;
        // Is it the last one ?
        if (p!= expected.end())
        {
            message << ", ";
            continue;
        }
        break;
    }
    message << " but found " << lex.token_stringfy(found);
    parser_fatal_error(message.str());
}
```

To pass a parameter for an initializer list, enclose a comma-separated list of values inside a pair of braces, as shown in the examples below for the `accept()` and `program()` member functions:

```
// verify and skip an expected token
void Parser::accept(Token_Type t)
{
    if (token == t)
    {
        token = get_token();
    }
    else
    {
        syntax_error(token, {t});
    }
}

// PROGRAM ::= VARS begin STMTLIST end
void Parser::program()
```

```
{
    if (token == Token_Type::t_int || token == Token_Type::t_begin)
    {
        vars();
        accept(Token_Type::t_begin);
        stmtlist();
        accept(Token_Type::t_end);
    }
    else
    {
        syntax_error(token, {Token_Type::t_int, Token_Type::t_begin});
    }
}
```

Notice that a program can begin with either a `t_int` token or a a `t_begin`.

Parsing methods for non-terminals must be documented with a comment showing the productions for that nonterminal.

7. Testing

To test your parser, use the following main function

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

using namespace std;

int main(int argc, char * argv[])
{
    Parser parser{"loopstmt.cmm", "loopstmt.lst"};

    parser.program();
    cout << "Program compiled without errors."  << endl;

    return 0;
}
```

You can test for different files by hardcoding different file names for the source and listing files. Open the listing file after the program runs to verify correctness.

A zip folder of sample test files will be posted shortly. However, exhaustive testing of your code is your own responsibility. The instructor may test your code with additional test files not available to you.

8. Due Date

This is due Friday at the end of week 5.