词法分析程序的设计和实现

——使用C++语言设计和实现

陈童

2019211283

304班

1.实验题目

创建一个词法分析程序，它支持对正规文法的分析。必须使用DFA（确定性有限自动机）或NFA（非确定性有限自动机）来实现这一项目。该程序的输入是一个文本文件，包括一组由该正规文法产生的产生式以及待识别源代码字符串。该程序的输出是一个符号表（二元式），它由5种类型符号：关键词，识别符，常量，界符和操作符。

2. 实验内容及要求

1. 可以识别出用C语言编写的源程序中的每个单词符号，并以记号的形式输出每个单词符号。

2.可以识别并跳过源程序中的注释。

3.可以统计源程序中的语句行数、各类单词的个数、以及字符总数，并输出统计结果。

4.检查源程序中存在的词法错误，并报告错误所在的位置。

5.对源程序中出现的错误进行适当的恢复，使词法分析可以继续进行，对源程序进一次扫描，即可检查并报告源程序中存在的所有词法错误。

3. 程序设计说明

2.1 词法分析程序的功能

1.扫描源程序字符流

2. 按照源语言的词法规则识别出各类单词符号

3. 产生用于语法分析的记号序列

4. 词法检查

5. 跳过源程序中的注释和空白，提供错误信息。

6. 实现一定程度的错误处理能力，当读入不符合词法规则的词，可以

2.2 源程序输出为的单词归类为：

1.关键词（keyword）

2. 标识符（id）

3. 数字常量（num）

4. 分界符和运算符（op）

2.3 实现思路

根据有限自动机的概念，把源代码分解规则分解为一个个状态，用if/else 或者switch函数控制状态之间的转移实现读入字符的分类和读入。

如果遇到错误的字符（自动机中无法读入），那么连续读入字符至下一个空格或换行，退出自动机，读入下一个字符并记录错误。

4. 源程序

main.cpp

```
/*
Lexical Syntax Analysis

Author: Linnkid_Chen
contact: linnkid.chen@gmail.com

Publish @ https://github.com/LinnkidChen/Lexical_Syntax_Analysis/tree/master

*/

#include "analysis/analysis.cpp"
// #include "analysis/analysis.h"
#include <cstring>
#include <fstream>
#include <iostream>

int main(int argc, char *argv[]) {
    std::ifstream input_strm;
    std::string input_pth;
    int statistic_output = 0; // determine whether output statistic information
    switch (argc) {
    case 2:
      input_pth = argv[1];
      break;
    case 3:
      if (argv[1][1] == 's') // view addition statics
        statistic_output = 1;
      input_pth = argv[2];
      break;

    default:
      std::cout << "Invalid parameters.\n";
      break;
    }

    Analysis ana(input_pth);
    error eor;
    statistic sta;

    if (ana.is_file_valid()) {
      ana.run(eor, sta);
    } else
      std::cout << "Invalid file name\n";

    eor.print_error();
```

```
  if (statistic_output)
    sta.print_sta();
  return 0;
}
```

Analysis.cpp

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

Analysis::Analysis(std::string path) {
  inpt_file.open(path);
  file_valid = inpt_file.is_open();

  std::ifstream kyw_file;
  std::string kyw;
  kyw_file.open("keywords.txt");
  while (!kyw_file.eof()) {
    kyw_file >> kyw;
    keyword.insert(kyw);
  }
}
void reset_reslt(ana_reslt_retn *reslt) {
  reslt->attribute.clear();
  reslt->note.clear();
  reslt->error.clear();
  reslt->type = -1;
  reslt->val = -1;
}
void Analysis::run(error &error_, statistic &sta_) {
  //   int state = 0; // state: 0 normal 1 line_comment; 2 block_comment
  ana_reslt_retn result;
  while (!inpt_file.eof()) {
    read_word(result, error_);
    switch (result.type) {
    case NUM:
      sta_.num++;
      print_reslt(result);
      break;
    case ID:
      sta_.id++;
      print_reslt(result);
      break;
    case KEYWORD:
      sta_.keyword++;
      print_reslt(result);
      break;
    case op:
      sta_.op_++;
      print_reslt(result);
      break;
    case COMMENT:
      sta_.comment += result.val;
      break;
    case ERROR:
      sta_.error++;
    }

    reset_reslt(&result);
  }
}

bool Analysis::is_file_valid() { return file_valid; }
void Analysis::read_word(ana_reslt_retn &reslt, error &error_) {
  reslt.type = -1;
  reslt.val = 0;
  char c;
  c = inpt_file.get();
  // skip empty
  while (c == ' ' || c == '\n')
    c = inpt_file.get();
  if (c == '/') {
    if (inpt_file.peek() == '/') { // line comment
      while (c != '\n')
        c = inpt_file.get();
      reslt.type = COMMENT;
      reslt.val = 1;
    }

    else if (inpt_file.peek() == '*') // block comment
    {
      c = inpt_file.get();
      reslt.type = COMMENT;
      while (!inpt_file.eof()) {
        c = inpt_file.get();
        if (c == '\n')
          reslt.val++;
        if (c == '*' && inpt_file.peek() == '/') {
          c = inpt_file.get();
          break;
        }
      }
    }
  }
  if (reslt.type < 0) { // not a comment
                       // regonize id
    if (isalpha(c) || c == '_') {
      reslt.note += c;
      while (isalpha(inpt_file.peek()) || isdigit(inpt_file.peek()) ||
             inpt_file.peek() == '_') {
        c = inpt_file.get();
        reslt.note += c;
      }
      if (keyword.find(reslt.note) == keyword.end()) { // not a keyword
        reslt.attribute = "ID";
        reslt.type = ID;
      } else {
        reslt.attribute = "KEYWORD";
        reslt.type = KEYWORD;
      }
    } else if (isnumber(c)) {
      reslt.note += c;
      int state = 1;
      while (state > 0) {
```

```
switch (state) {
case 1:
  if (isnumber(inpt_file.peek())) {
    c = inpt_file.get();
    reslt.note += c;
    state = 1;
  } else if (inpt_file.peek() == '.') {
    c = inpt_file.get();
    reslt.note += c;
    state = 2;
  } else if (inpt_file.peek() == 'E' || inpt_file.peek() == 'e') {
    c = inpt_file.get();
    reslt.note += c;
    state = 4;
  } else {
    if (std::isalnum(inpt_file.peek())) {
      reslt.type = ERROR;
      while (std::isalnum(inpt_file.peek())) {
        c = inpt_file.get();
        reslt.note += c;
      }
      error_.add_error("INVALID WORD: " + reslt.note);
    }
    state = 0;
  }
  break;
case 2:
  if (isnumber(inpt_file.peek())) {
    c = inpt_file.get();
    reslt.note += c;
    state = 3;
  } else {
    while (std::isalnum(inpt_file.peek()) || inpt_file.peek() == '_') {
      reslt.note += c;
      c = inpt_file.get();
    }
    reslt.type = ERROR;
    error_.add_error("Invalid word: " + reslt.note);
  }
  break;
case 3:
  if (isnumber(inpt_file.peek())) {
    c = inpt_file.get();
    reslt.note += c;
    state = 3;
  } else if (inpt_file.peek() == 'E' || inpt_file.peek() == 'e') {
    c = inpt_file.get();
    reslt.note += c;
    state = 4;
  } else
    state = 0;
  break;
case 4:
  if (inpt_file.peek() == '+' || inpt_file.peek() == '-') {
    c = inpt_file.get();
    reslt.note += c;
    state = 6;
  } else if (isnumber(inpt_file.peek())) {
    c = inpt_file.get();
    reslt.note += c;
    state = 5;
  }

  else {
    while (std::isalnum(inpt_file.peek()) || inpt_file.peek() == '_') {

      c = inpt_file.get();
      reslt.note += c;
    }
    reslt.type = ERROR;
    error_.add_error("Invalid word: " + reslt.note);
  }
  break;
case 5:
  if (isnumber(inpt_file.peek())) {
    c = inpt_file.get();
    reslt.note += c;
    state = 5;
  } else
    state = 0;
  break;
case 6:
  if (isnumber(inpt_file.peek())) {
    c = inpt_file.get();
    reslt.note += c;
    state = 5;
  } else {
        while (std::isalnum(inpt_file.peek()) || inpt_file.peek() == '_') {

          c = inpt_file.get();
          reslt.note += c;
        }
        reslt.type = ERROR;
        error_.add_error("Invalid word: " + reslt.note);
      }
    }
  }
  if (reslt.type < 0) {
    reslt.type = NUM;
    reslt.attribute = "NUM";
  }
}
}
if (reslt.type < 0) {
// relop
if (c == '<') {
  switch (inpt_file.peek()) {
  case '=':
    reslt.attribute = "LE";
    reslt.note = "relop";
    reslt.type = op;
    c = inpt_file.get();
    break;
  case '>':
    reslt.attribute = "NE";
    reslt.note = "relop";
    reslt.type = op;
    c = inpt_file.get();
    break;
  default:
    reslt.attribute = "LT";
    reslt.note = "relop";
    reslt.type = op;
  }

} else if (c == '=') {
  reslt.attribute = "EQ";
  reslt.note = "relop";
  reslt.type = op;
} else if (c == '>') {
  if (inpt_file.peek() == '=') {
    reslt.attribute = "GE";
    reslt.note = "relop";
    reslt.type = op;
    c = inpt_file.get();
  } else {
    reslt.attribute = "GT";
    reslt.note = "relop";
    reslt.type = op;
  }
} else if (c == ':') {
  if (inpt_file.peek() == '=') {
    reslt.attribute = "";
    reslt.note = "assign-op";
    reslt.type = op;
```

```
      } else {
        reslt.attribute = "";
        reslt.note = ":";
        reslt.type = op;
      }
    }
  }
  if (reslt.type < 0) {
    switch (c) {
    case '+':
      reslt.attribute = "";
      reslt.note = "+";
      reslt.type = op;
      break;
    case '-':
      reslt.attribute = "";
      reslt.note = "-";
      reslt.type = op;
      break;
    case '*':
      reslt.attribute = "";
      reslt.note = "*";
      reslt.type = op;
      break;
    case '/':
      reslt.attribute = "";
      reslt.note = "/";
      reslt.type = op;
      break;
```

```
case '{':
  reslt.attribute = "";
  reslt.note = "{";
  reslt.type = op;
  break;
case '}':
  reslt.attribute = "";
  reslt.note = "}";
  reslt.type = op;
  break;
case '(':
  reslt.attribute = "";
  reslt.note = "(";
  reslt.type = op;
  break;
case ')':
  reslt.attribute = "";
  reslt.note = ")";
  reslt.type = op;
  break;
case 39: //'
  reslt.attribute = "";
  reslt.note = "\'";
  reslt.type = op;
  break;
case ';':
  reslt.attribute = "";
  reslt.note = ";";
  reslt.type = op;
```

```
      break;
    case ',':
      reslt.attribute = "";
      reslt.note = ",";
      reslt.type = op;
      break;
    case '!':
      reslt.attribute = "";
      reslt.note = "!";
      reslt.type = op;
      break;
    default:
    case '\"':
      reslt.attribute = "";
      reslt.note = "\"";
      reslt.type = op;
      break;
      reslt.note += c;
      error_.add_error("Illegal symbol: " + reslt.note);
      reslt.type = ERROR;
    }
  }
}

void Analysis::print_reslt(ana_reslt_retn const &reslt) {
  switch (reslt.type) {
  case NUM:
    std::cout << reslt.note << "  " << reslt.attribute << std::endl;
    break;
  case ID:
    std::cout << reslt.note << "  " << reslt.attribute << std::endl;
    break;
  case KEYWORD:
    std::cout << reslt.note << "  " << reslt.attribute << std::endl;
    break;
  case op:
    std::cout << reslt.note << "  " << reslt.attribute << std::endl;
    break;
  }
}
```

Analysis.h

```
#pragma once
#include "error/error.h"
#include "statistics/statistic.h"
#include <ctype.h>
#include <fstream>
#include <iostream>
#include <istream>
#include <set>
#include <string>
#define NUM 0
#define ID 1
#define KEYWORD 2
#define ERROR 3
#define COMMENT 4
#define op 5
class ana_reslt_retn {
```

```
public:
  std::string note, attribute;
  std::string error;
  int type;
  int val;
};

class Analysis {
public:
  Analysis(std::string path);
  void run(error &error_, statistic &sta_);
  std::string read_one_word;
  bool is_file_valid();
  void read_word(ana_reslt_retn &reslt, error &error_);

  void print_reslt(ana_reslt_retn const &reslt);

private:
  std::ifstream inpt_file;
  bool file_valid;
  std::set<std::string> keyword;
  int status; // distinguish current status. determine whether it is a comment;
  // 1 for not comment; 2 for // comment; 3 for /*comment;
};
```

error.cpp

```
#include <error.h>
#include <iterator>
void error::print_error() {
  std::list<std::string>::iterator it;
  it = errors.begin();
  while (it != errors.end()) {
    std::cout << "Error: " << *it << std::endl;
    it++;
  }
}

void error::add_error(std::string input) { errors.push_back(input); }
```

error.h

```
#pragma once
#include <iostream>
#include <list>
#include <string>
class error {
public:
  std::list<std::string> errors;
  void print_error();
  void add_error(std::string);
};
```

Statistic.h

```
#pragma once
#include <iostream>
class statistic {
public:
  statistic() {
    num = 0;
    id = 0;
    keyword = 0;
    error = 0;
    comment = 0;
    op_ = 0;
  }
  int num;
  int id;
  int keyword;
  int error;
  int comment;
  int op_;
  void print_sta();
};
```

Statistic.cpp

```
#include "statistic.h"
#include <ostream>
void statistic::print_sta() {
  std::cout << std::endl;
  std::cout << "NUM: " << num << std::endl
            << "ID: " << id << std::endl
            << "KEYWORD: " << keyword << std::endl
            << "OP: " << op_ << std::endl
            << "COMMENT: " << comment << "(lines)\n"
            << "ERROR: " << error << std::endl;
}
```

文件结构

```
├── Main.cpp
├── README.md
├── analysis
│   ├── CMakeLists.txt
│   ├── analysis.cpp
│   ├── analysis.h
│   ├── error
│   │   ├── CMakeLists.txt
│   │   ├── error.cpp
│   │   └── error.h
│   └── statistics
│       ├── CMakeLists.txt
│       ├── statistic.h
│       └── statistics.cpp    │
  ├── keywords.txt
  └── test.txt
```

5. 程序测试和分析

详见程序测试报告

6. 心得体会

经过这次词法分析程序的编写，我认识到词法分析的重要性。它作为独立的一遍，为语法分析提供充分的铺垫。

词法分析将源程序拆解为独立的词，利用<属性，记号>的二元表为语法分析程序提供待编译程序的信息。将处理源程序的一部分工作拆分出来，可以提高编译程序的效率，并且区隔开各个部分的功能的代码。

在本次实验过程中，我对自动机，词法分析的相关知识有更深一步的理解和运用。