1 Introduction

This is a report for the program design of the lexical analysis parser about C language. The parser is programmed in Java, with 5 classes and approximately 400 lines which including the comments and blanks. Overall, it involves the functions following:

a. Detect input string errors: including some typical error types, and indicating the line number of error occurring;
b. Distinguish the lexical type: including **identifier**, **operator**, **separator**, **number type**, **preserved words**, **comments**, **literal string and some other delimiters**; More detail will be illustrated in the following section;
c. The code is read from filename.c (entered by user in the console), and it stores the results in output.txt;
d. The grammar rules could be simply modified though other 4 class, which stores the symbol in separate arrays.

2 Design Idea

The lexical analyzer is provided to the parser as an encapsulation class. The input of the lexical analyzer is char stream of code in C and the output is token stream. The main idea is matching the char stream with given lexical tables of grammar rules, which actually stored as the array of data structure in the program.

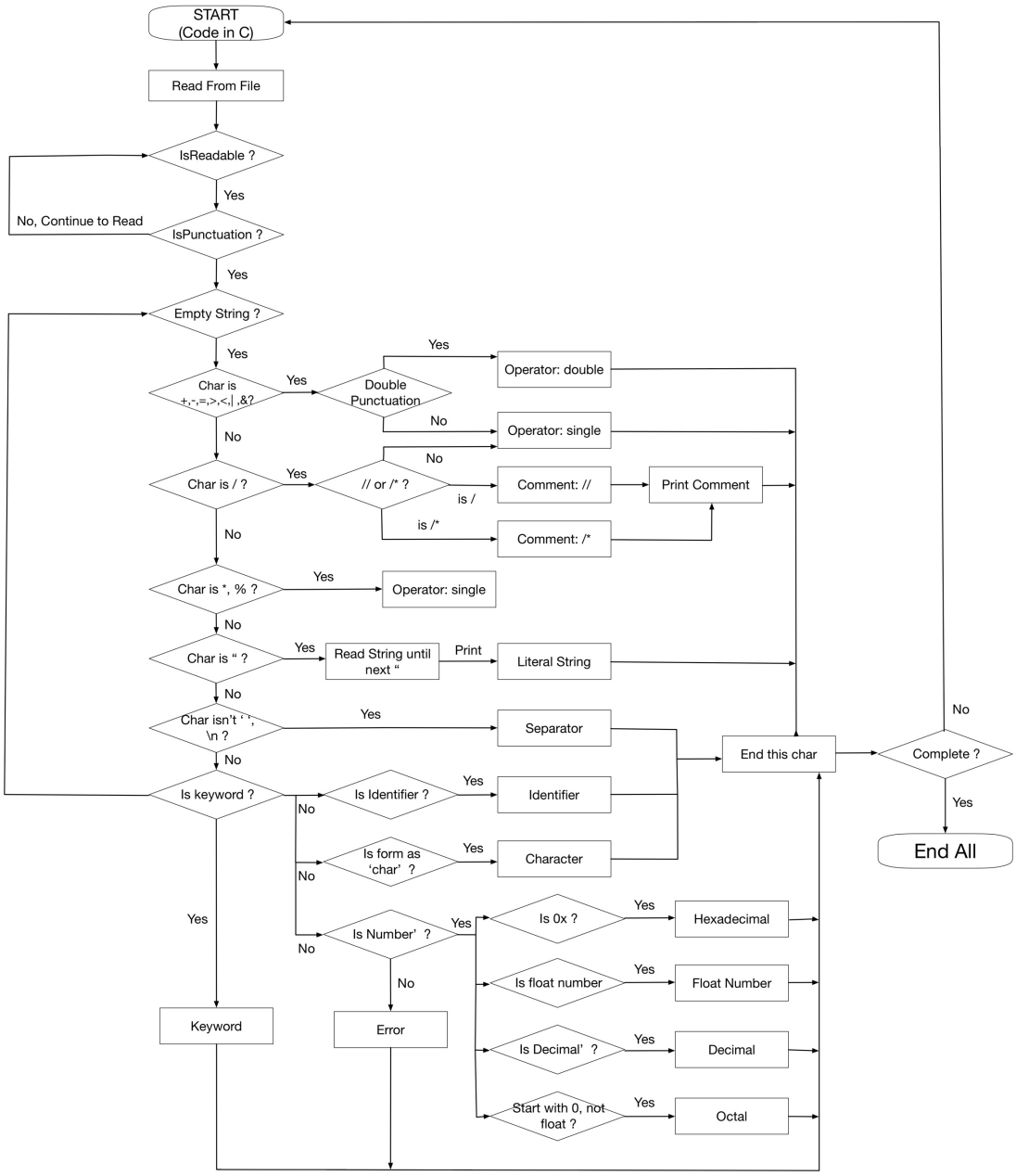
The program read the original code char by char, while it sometimes read the following char in advanced to detect the "double symbols" in an integrity, such as ==, >= and &&. It is firstly check whether the char is punctuation.

For the punctuation of **+ - = > < | & ,** the program will determine whether it is double-linked punctuations or single punctuations, then finding their corresponding type in the punctuation-array and saving the result with type, value and content in the output file.

For the punctuation of **/** , the program will what the next character is, **//** represents single-line comments, **/*** represents multi-line comments, and others means **single operator**.

For the punctuation of ***** and **%** , both of them indicate the single operator (not in literal string).

For the symbol **"** , it is the format of literal string, and the program should read all the characters for the single string from the first " to the last ".



For other punctuations except ' ' **\n** **\r**, there are still remain the separators, including **;** **,** **{** **}** **(** **)** **[** **]**

The involving punctuations are store in the data structure as the array in Punctuation.java, including the returning result.

The program is also could check the identifiers various types, such as letter, number, mixture and file name, such as **stdio.h**. Algorithms are in the Identifier.java. Additionally, it could justify the single character in the specific format which occupying 3 chars with ' starting and ' ending. Except for these

two conditions, the result is number and the parser could analyze which type the number is from algorithms in Number.java. Finally, the remaining should be preserved words which stored in the data structure of the array in PreservedWords.java.

The detail of design process shows in the flow chart above.

3 Functions

Type-Value Table		
Code	**Type**	**Including**
1	operator	+, -, =, >, <, *, /, %, |, &, ++, +=, --, -=, <=, >=, ==, ||, &&
2	identifier	(various types), including stdio.h
3	separator	, ; { } [ ] ( )
4	character	As the format like ‘c’
5	string	As the format like “String”
6	comment indicator	//, /*
7	comment	Any string with any line in // or /* */
8	keyword	"void", "double", "int", "struct", "float", "if", "else", "for", "do", "while", "case", "loop", "break", "long", "switch", "enum", "typedef", "char", "return", "union", "continue", "signed", "static", "default", "goto", "sizeof", "volatile", "const", "short","unsigned", "#include"
9	float	Eg. 3.8723
10	decimal	Eg. 893
11	octal	Eg. 012
12	hexadecimal	Eg. 0x13f

There are also the error identifier, indicate the line, error type and context of the wrong code. The error type including: Number Format Type Error, Format Error and Unexcepted Error.

4 Data Structure

There are mainly two array of string and char to store the preserved words and punctuations separately, which easily to modify. If there are more preserved words and punctuations required, the users could directly add the new one to the existing two arrays via the modifying the constant variable of array in the PreservedWords.java and Punctuations.java

The detail shows below:

```
private String p_word[] = {"void", "double", "int",
"struct", "float", "if", "else", "for", "do", "while",
"case", "loop", "break", "long", "switch", "enum",
   "typedef","char", "return", "union", "continue",
"signed", "static", "default", "goto", "sizeof",
"volatile", "const", "short","unsigned", "#include"};

char punctuations[] = {'+', '-', '=', '>', '<','*',  '/',
'%', ';', ',', '{', '}', '(', ')', ' ', '\n', '\r', '"',
'[', ']', '&', '|' };
```

5 Results

Instance 1:

```
1  //Author: Xinlong
2  #include <stdio.h>
3  int main()
4  {
5      int length;
6      char ch = 'c';
7      printf("The number of books is 8");
8
9      /* It is a loop */
10     for(int i=2; i<length; i++)
11     {
12         for(int j=1; j<length; j++)
13         {
14             if(i == 5) {
15                 break;
16             }
17             printf("*");
18         }
19     }
20
21 }
```

Output

input2_result.txt

```
<Comment Indicator, 6>:  //
<Comment, 7>:  Author: Xinlong
<Keyword, 8>:  #include
<Operator, 1>:  <
<Identifier, 2>:  stdio.h
<Operator, 1>:  >
<Keyword, 8>:  int
<Identifier, 2>:  main
<Separator, 3>:  (
<Separator, 3>:  )
<Separator, 3>:  {
<Keyword, 8>:  int
<Identifier, 2>:  length
<Separator, 3>:  ;
<Keyword, 8>:  char
<Identifier, 2>:  ch
<Operator, 1>:  =
<Character, 4>:  c
<Separator, 3>:  ;
<Identifier, 2>:  printf
<Separator, 3>:  (
<Literal String, 5>:  "The number of books is 8"
<Separator, 3>:  )
<Separator, 3>:  ;
<Comment Indicator, 6>:  /*
<Comment, 7>:  It is a loop
<Comment Indicator, 6>:  */
<Keyword, 8>:  for
<Separator, 3>:  (
<Keyword, 8>:  int
<Identifier, 2>:  i
<Operator, 1>:  =
<Decimal Number, 10>:  2
<Separator, 3>:  ;
<Identifier, 2>:  i
<Operator, 1>:  <
<Identifier, 2>:  length
<Separator, 3>:  ;
<Identifier, 2>:  i
<Operator, 1>:  ++
<Separator, 3>:  )
<Separator, 3>:  {
<Keyword, 8>:  for
<Separator, 3>:  (
<Keyword, 8>:  int
<Identifier, 2>:  j
<Operator, 1>:  =
<Decimal Number, 10>:  1
<Separator, 3>:  ;
<Identifier, 2>:  j
<Operator, 1>:  <
<Identifier, 2>:  length
<Separator, 3>:  ;
<Identifier, 2>:  j
<Operator, 1>:  ++
<Separator, 3>:  )
<Separator, 3>:  {
<Keyword, 8>:  if
<Separator, 3>:  (
<Identifier, 2>:  i
<Operator, 1>:  ==
<Decimal Number, 10>:  5
<Separator, 3>:  )
<Separator, 3>:  {
<Keyword, 8>:  break
<Separator, 3>:  ;
<Separator, 3>:  }
<Identifier, 2>:  printf
<Separator, 3>:  (
<Literal String, 5>:  "*"
<Separator, 3>:  )
<Separator, 3>:  ;
<Separator, 3>:  }
<Separator, 3>:  }
<Separator, 3>:  }
```

Instance 2

```c
#include <stdio.h>
int main()
{
    int height, height_1 = 20;
    int width = 10;
    char ch = 'c';

    /* There is the if statement */
    if(height >= width) {
        int area = height * width;
        //If the area is more than 100
        if(area > 100) {
            int area += 100;
            float percentage = 0.2312;
            printf("The total area is " + total_area);
            if((@$ <= 1 && area <1000)) {
                height++;
            }
        }
        else {
            printf("It is a small area: " + area);
        }

    }
    return 1;
}
```

Output

input1_result.txt

```
<Keyword, 8>:  #include
<Operator, 1>:  <
<Identifier, 2>:  stdio.h
<Operator, 1>:  >
<Keyword, 8>:  int
<Identifier, 2>:  main
<Separator, 3>:  (
<Separator, 3>:  )
<Separator, 3>:  {
<Keyword, 8>:  int
<Identifier, 2>:  height
<Separator, 3>:  ,
<Identifier, 2>:  height_1
<Operator, 1>:  =
<Decimal Number, 10>:  20
<Separator, 3>:  ;
<Keyword, 8>:  int
<Identifier, 2>:  width
<Operator, 1>:  =
<Decimal Number, 10>:  10
<Separator, 3>:  ;
<Keyword, 8>:  char
<Identifier, 2>:  ch
<Operator, 1>:  =
<Character, 4>:  c
<Separator, 3>:  ;
<Comment Indicator, 6>:  /*
<Comment, 7>:   There is the if statement
<Comment Indicator, 6>:  */
<Keyword, 8>:  if
<Separator, 3>:  (
<Identifier, 2>:  height
<Operator, 1>:  >=
<Identifier, 2>:  width
<Separator, 3>:  )
<Separator, 3>:  (
<Separator, 3>:  (
Error Format at  @$
<Operator, 1>:  <=
<Decimal Number, 10>:  1
<Operator, 1>:  &&
<Identifier, 2>:  area
<Operator, 1>:  <
<Decimal Number, 10>:  1000
<Separator, 3>:  )
<Separator, 3>:  )
<Separator, 3>:  {
<Identifier, 2>:  height
<Operator, 1>:  ++
<Separator, 3>:  ;
<Separator, 3>:  }
<Separator, 3>:  }
<Keyword, 8>:  else
<Separator, 3>:  {
<Identifier, 2>:  printf
<Separator, 3>:  (
<Literal String, 5>:  "It is a small area: "
<Operator, 1>:  +
<Identifier, 2>:  area
<Separator, 3>:  )
<Separator, 3>:  ;
<Separator, 3>:  }
<Separator, 3>:  }
<Keyword, 8>:  return
<Decimal Number, 10>:  1
<Separator, 3>:  ;
<Separator, 3>:  }
```

```
<Separator, 3>:  )
<Separator, 3>:  {
<Keyword, 8>:  int
<Identifier, 2>:  area
<Operator, 1>:  =
<Identifier, 2>:  height
<Operator, 1>:  *
<Identifier, 2>:  width
<Separator, 3>:  ;
<Comment Indicator, 6>:  //
<Comment, 7>:  If the area is more than 10(
<Keyword, 8>:  if
<Separator, 3>:  (
<Identifier, 2>:  area
<Operator, 1>:  >
<Decimal Number, 10>:  100
<Separator, 3>:  )
<Separator, 3>:  {
<Keyword, 8>:  int
<Identifier, 2>:  area
<Operator, 1>:  +=
<Decimal Number, 10>:  100
<Separator, 3>:  ;
<Keyword, 8>:  float
<Identifier, 2>:  percentage
<Operator, 1>:  =
<Float Number, 9>:  0.2312
<Separator, 3>:  ;
<Identifier, 2>:  printf
<Separator, 3>:  (
<Literal String, 5>:  "The total area is "
<Operator, 1>:  +
<Identifier, 2>:  total_area
<Separator, 3>:  )
<Separator, 3>:  ;
<Keyword, 8>:  if
```