Name : 羅翡瑩
ID : B11015010

Homework 1 - Scanner

1. Code Explanation

1.1 Lexical code file

To do a scanning for a qv program, I need to create a lexical file which is 11015010_scanner.lex. The code will be explain as below:

```
%{
#include "symbolTable.h"
#define LIST                strcat(buffer,yytext)
#define token(t)            {LIST; printf("<%s>\n",#t);}
#define tokenInteger(t,i)   {LIST; printf("<%s:%d>\n",#t,i);}
#define tokenString(t,s)    {LIST; printf("<%s:%s>\n",#t,s);}

#define MAX_BUFFER_SIZE 256

int linenum = 1;
char buffer[MAX_BUFFER_SIZE];

symbolTable* table;

%}
```

In the beginning, I include a header file named symbolTable.h which defines functions for storing and managing the symbols (variable identifiers) that the scanner might encounter in the input text. Then, later i define several macros using *#define* as these:

- LIST : accumulating the characters of the current token
- token : categorizing tokens into different types and print the categorize
- tokenInteger : include an integer argument i. This will be used for tokens that have an integer value
- tokenString : takes string argument and prints it as format later in the program running.

The linenum is to track the current line number which might be helpful for later handling the comment. While the buffer is for temporarily storing characters of the current token being scanned.

```
DIGITS              [0-9]+
REAL_NUMBER         [+-]?{DIGITS}\.({DIGITS})?([Ee][+-]?{DIGITS})?
IDENTIFIER          [a-zA-Z_][a-zA-Z_0-9]*
LINE_COMMENT        (\/\/[^\n]*)
PARA_COMMENT_OPEN   (\/\*)
PARA_COMMENT_CLOSE  (\*\/)

%x COMMENT STRING_TEXT CHAR_TEXT
```

In here will defines patterns for different types of tokens the scanner will recognize in the input text:

- DIGITS : the pattern use regular expression of [0-9]+ for occurrences of digits 0 to 9
- REAL_NUMBER : It has more complex regular expressions which consider + -, decimal, Ee, etc.
- IDENTIFIER : This will be used for variable names, function names, or other user-defined names, so it accepts the alphabet, number, underscore
- LINE_COMMENT : This for a single line comment
- PARA_COMMENT_OPEN : The starting comment for multiple line comment
- PARA_COMMENT_CLOSE : The closing comment for multiple line comment

In addition, there are also comment string_text and char_text which will be defined later below.

```
%%

"var"           {token(VAR);}
"val"           {token(VAL);}
"bool"          {token(BOOL);}
"char"          {token(CHAR);}
"int"           {token(INT);}
"real"          {token(REAL);}
"true"          {token(TRUE);}
"false"         {token(FALSE);}
"class"         {token(CLASS);}
"if"            {token(IF);}
"else"          {token(ELSE);}
"for"           {token(FOR);}
"while"         {token(WHILE);}
"do"            {token(DO);}
"switch"        {token(SWITCH);}
"case"          {token(CASE);}
"default"       {token(DEFAULT);}
"fun"           {token(FUNCTION);}
"ret"           {token(RETURN);}
"main"          {token(MAIN);}
"println"       {token(PRINTLN);}
```

```
"+"             {token('+');}
"-"             {token('-');}
"*"             {token('*');}
"/"             {token('/');}
"="             {token('=');}
"<"             {token('<');}
"≤"             {token('≤');}
"≥"             {token('≥');}
">"             {token('>');}
"=="            {token('==');}
"≠"             {token('≠');}
"%"             {token('%');}
"++"            {token('++');}
"--"            {token('--');}
"&&"            {token('&&');}
"||"            {token('||');}


","             {token(',');}
":"             {token(':');}
";"             {token(';');}
"("             {token('(');}
")"             {token(')');}
"["             {token('[');}
"]"             {token(']');}
"{"             {token('{');}
"}"             {token('}');}
```

The code above is to define the tokenization of keyword, operator and punctuation rules.

```
{IDENTIFIER}    { tokenString(id, yytext);
                  table->insert(yytext);}

{DIGITS}        { tokenInteger(int, atoi(yytext)); }

{REAL_NUMBER}   { tokenString(real, yytext);}
```

The IDENTIFIER defines a rule for tokens that match the identifier pattern, where tokenString(id, yytext) calls macro tokenString to print the constant

and the variable. Also the variable identifier will be inserted to the symbol table. While, DIGITS is rules for tokens that match the digits pattern which also call the tokenInteger macro for the processing. Lastly, the REAL_NUMBER is defined for tokens that match the real number pattern, where in here call the tokenString for the further processing.

```
"\'"            { yymore();
                  BEGIN CHAR_TEXT; }

<CHAR_TEXT>[^\']        { yymore(); }

<CHAR_TEXT>"\\n"        { yymore(); }

<CHAR_TEXT>"\\t"        { yymore(); }

<CHAR_TEXT>"\\\\"       { yymore(); }

<CHAR_TEXT>"\\\'"       { yymore(); }

<CHAR_TEXT>"\\\""       { yymore(); }

<CHAR_TEXT>"\\\?"       { yymore(); }
```

This one is for handling the single quote (or the char handling), where we start by continuing matching characters (yymore) and switch to CHAR_TEXT state for further processing. Here, there are a lot of definitions for the escape sequences within the CHAR_TEXT state.

```
<CHAR_TEXT>"\'"       { int pos = 0;
                        int i = 1;
                        char char_text[MAX_BUFFER_SIZE];
                        if (yytext[i] == '\\')
                        {
                            if (yytext[i+1] == 'n')
                            {
                                char_text[pos] = '\n';
                                i += 2;
                            }
                            else if (yytext[i+1] == 't')
                            {
                                char_text[pos] = '\t';
                                i += 2;
                            }
                            else if (yytext[i+1] == '\\')
                            {
                                char_text[pos] = '\\';
                                i += 2;
                            }
                            else if (yytext[i+1] == '\'')
                            {
                                char_text[pos] = '\'';
                                i += 2;
                            }
                            else if (yytext[i+1] == '\"')
                            {
                                char_text[pos] = '\"';
                                i += 2;
                            }
                            else if (yytext[i+1] == '\?')
                            {
                                char_text[pos] = '\?';
                                i += 2;
                            }
                        }
```

Finally, it reaches until it matches the closing quotes, where in here it checks for \n, \t, \\, \', \", and \? and accept it and convert them into their corresponding special characters within a character array char_text.

```
  else
  {
      if (yyleng > 3)
      {
          exit(-1);
      }
      else
      {
          char_text[pos] = yytext[i];
          i++;
      }
  }
  char_text[pos+1] = '\0';
  tokenString(char, char_text);
  BEGIN 0;
}
```

If no escape sequence is found, it simply copies the character from yytext to the char_text array. In addition, the error handling (if (yyleng > 3) { exit(-1); }) for checking for invalid and exits with an error. Later, it will flush the buffer and print the tokenString.

```
"\""            { yymore();
                  BEGIN STRING_TEXT; }

<STRING_TEXT>[^\"]    { yymore(); }

<STRING_TEXT>"\\n"    { yymore(); }

<STRING_TEXT>"\\t"    { yymore(); }

<STRING_TEXT>"\\\\"   { yymore(); }

<STRING_TEXT>"\\\'"   { yymore(); }

<STRING_TEXT>"\\\""   { yymore(); }

<STRING_TEXT>"\\\?"   { yymore(); }

<STRING_TEXT>"\""     { int pos = 0;
                        char str_text[MAX_BUFFER_SIZE];
                        for (int i = 1; i < yyleng - 1; i++)
                        {
                            if (yytext[i] == '\\' && yytext[i+1] == '\"')
                            {
                                str_text[pos] = '\"';
                                i++;
                            }
                            else if (yytext[i] == '\\' && yytext[i+1] == '\'')
                            {
                                str_text[pos] = '\'';
                                i++;
                            }
                            else if (yytext[i] == '\\' && yytext[i+1] == '\?')
                            {
                                str_text[pos] = '\?';
                                i++;
                            }
                            else
                            {
                                str_text[pos] = yytext[i];
                            }
                            pos++;
                        }
                        str_text[pos] = '\0';
                        tokenString(string, str_text);
                        BEGIN 0;
                      }
```

This one is similar to the one above, but this section handles matching double quotes. It continue matching characters even if it encounters a newline character (yymore), and switches the scanner to a special state named STRING_TEXT.

Within the state, it also checks for \n, \t, \\, \', \", and \? and accept it and convert them into their corresponding special characters. Some I didn't convert (if alphabetical) because it won't be an error even though I didn't define it. But if it is symbol character (?,",',') then it needs to be converted to avoid the error. If no escape sequence is found, it simply copies the character from yytext to the str_text array. At last, it will call the tokenString for further processing.

```
{LINE_COMMENT}                  { LIST; }

{PARA_COMMENT_OPEN}             { LIST;
                                  BEGIN COMMENT; }

<COMMENT>[^\n]                  { LIST; }

<COMMENT>[\n]                   { LIST;
                                  printf("%d: %s", linenum, buffer);
                                  linenum++;
                                  buffer[0] = '\0'; }

<COMMENT>{PARA_COMMENT_CLOSE}   { LIST;
                                  BEGIN 0; }
```

The LINE_COMMENT matches any text from "//" to the end of line. While PARA_COMMENT_OPEN is defines a rule that match the opening delimiter of multiple-line comment (/*). In here, it will switch to COMMENT state which defines rules that match any character except new line (\n) and rule that matches newline character will further processing of increment linenum, store to buffer, and printing.

Finally, the closing delimiter of multi line comment (*/) which will exit the comment state and return to scanning state by BEGIN 0.

```
\n      {
        LIST;
        printf("%d: %s", linenum++, buffer);
        buffer[0] = '\0';
        }

\r\n    {
        LIST;
        printf("%d: %s", linenum++, buffer);
        buffer[0] = '\0';
        }

[ \t]*  {LIST;}

.       {
        LIST;
        printf("%d:%s\n", linenum+1, buffer);
        printf("bad character:'%s'\n",yytext);
        exit(-1);
        }
%%
```

Here, there is \n and \r\n which define the new line rules. It will print the current line number which is incremented by 1 and the content of the buffer to the console which prints the entire line that was just scanned. The buffer is then set to '\0' which means null to clear the buffer for the next line. The \r\n is to avoid the error of windows linux.

This [\t]* {LIST;} line defines a rule that matches zero or more occurrences of whitespace characters (spaces and tabs). The asterisk (*) indicates zero or more repetitions. While, the dot (.) defines a rule that matches any character except the ones defined in previous rules, which here will give error code (-1) upon encountering an unexpected character.

```
int main(int argc, char** argv)
{
    table = new symbolTable();
    if(argc>1)
    {
        yyin = fopen(argv[1], "r");
    }
    else
    {
        yyin = stdin;
    }

    yylex();
    cout<<"\nSymbol Table:\n";
    table->dump();
    return 0;
}
```

In this main program, it will create a symbol table and open the file specified in argv[1] for reading using the fopen function. Also, here calls function yylex() to perform the task of identifying tokens. After the yylex feels that the symbol table has been populated with information, then it will print all the content in the symbol table.

1.2 Symbol Table header file

As we want to insert the variabel ID that we encounter to the symbol table, so we will handle te storing and managing the symbol table in the symbolTable.h

```h
#pragma once

#include<iostream>
#include<map>
#include<vector>
using namespace std;

class symbolTable
{
private:
  map<string, int> symbol;
  int i = 0;
public:
  symbolTable() { i = 0; }
  int lookup(string s);
  int insert(string s);
  int dump();
};
```

```
int symbolTable::lookup(string s)
{
  auto look = symbol.find(s);
  if (look != symbol.end())
  {
    return symbol[s];
  }
  else
  {
    return -1;
  }
}
```

Here, the lookup method takes string s as input and returns an index associated with the symbol s in the symbol table. If the symbol is not found the methods will return -1.

```
int symbolTable::insert(string s)
{
  int look = lookup(s);
  if (look != -1)
  {
    return look;
  }
  else
  {
    symbol[s] = i;
    i++;
  }

  return i;
}
```

takes a string s as input and inserts it into the symbol table. If the symbol already exists in the symbol table, the method returns the existing index. Otherwise, it assigns a new unique index to the symbol and returns the new index.

```
int symbolTable::dump()
{
  vector<string> temp;
  for (int pos = 0; pos < i; pos++)
  {
    for (const auto& name : symbol)
    {
      if (name.second == pos)
      {
        temp.push_back(name.first);
        break;
      }
    }
  }

  for (int pos = 0; pos < i; pos++)
  {
    cout << pos << " " << temp[pos] << "\n";
  }

  return i;
}
```

This dump method prints the contents of the symbol table. It iterates over the symbol table and prints each symbol-index pair.

2. Run Program

For running the code, I created a Makefile which could be run by typing *make all* in the linux environment. Currently I run the code in mobaxterm wsl ubuntu.

This Makefile defines how to build a scanner program called B11015010_scanner. The all below is to make the program B11015010_scanner executable by using g++ compiler to create the executable. To achieve this, it needs to generate the lex.yy.cc file, which is in C++ source code for the scanner.

Makefile

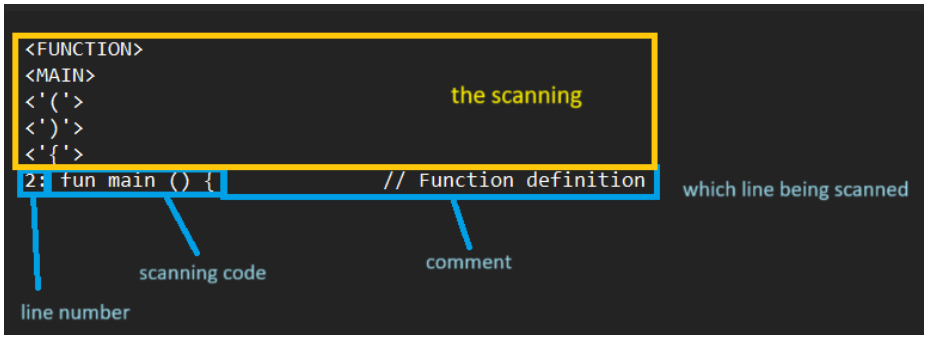
```
all:  B11015010_scanner

B11015010_scanner:  lex.yy.cc
  g++ -o B11015010_scanner -O lex.yy.cc -ll

lex.yy.cc:  11015010_scanner.lex
  lex 11015010_scanner.lex
  mv lex.yy.c lex.yy.cc

clean:
  rm lex.yy.cc B11015010_scanner
```

```
b11015010@B:~/B11015010_scanner$ ./B11015010_scanner < sample1.qv
```

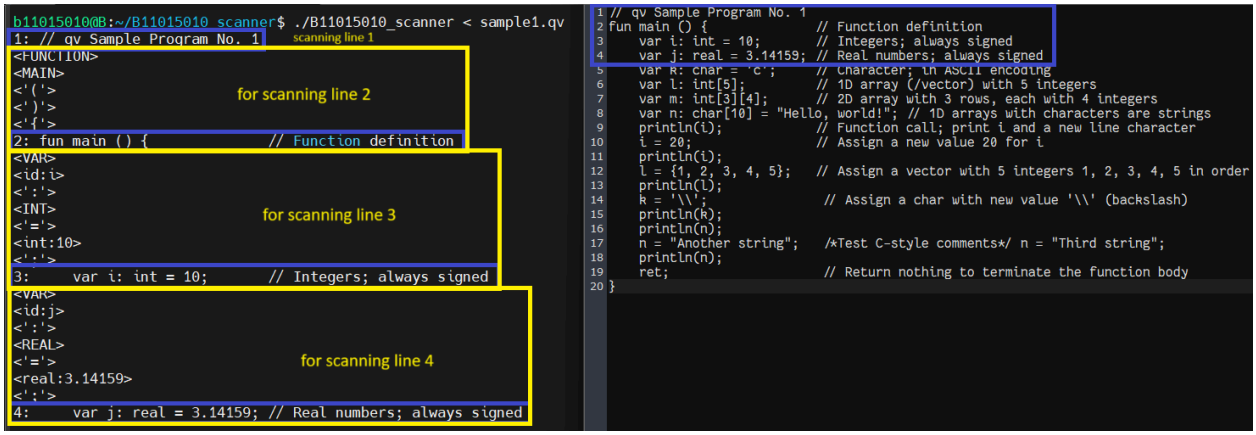
After all the building of the scanner program, then can be executed and provided the input file by typing ./B11015010_scanner < sample1.qv in linux terminal.

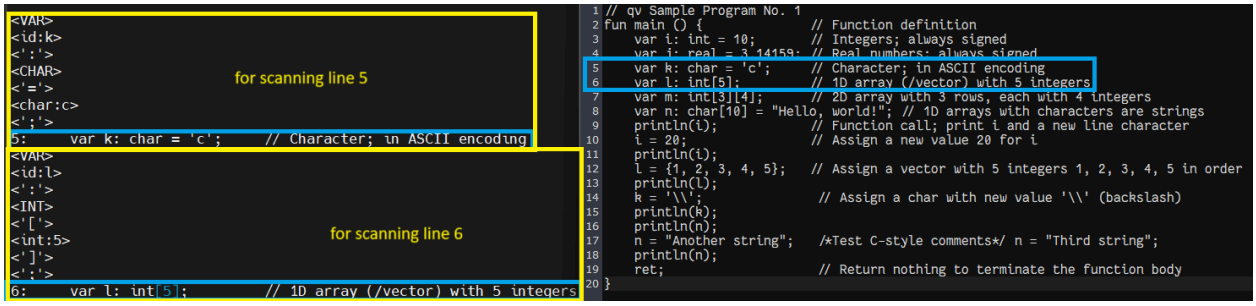
3. Result of the scanning

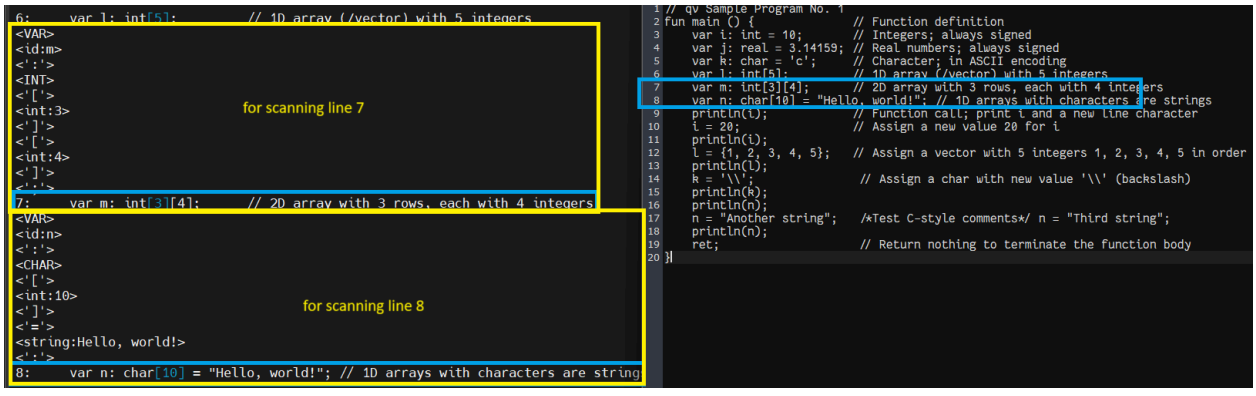
To read the scanning printing, take look for example:



3.1 sample1.qv







```
<PRINTLN>
<'('>
<id:i>
<')'>
<';'>
9:      println(i);             // Function call; print i and a new line character
<id:i>
<'='>
<int:20>
<';'>
10:     i = 20;                 // Assign a new value 20 for i
<PRINTLN>
<'('>
<id:i>
<')'>
<';'>
11:     println(i);
```

```
1  // qv Sample Program No. 1
2  fun main () {               // Function definition
3      var i: int = 10;        // Integers; always signed
4      var j: real = 3.14159;  // Real numbers; always signed
5      var k: char = 'c';      // Character; in ASCII encoding
6      var l: int[5];          // 1D array (/vector) with 5 integers
7      var m: int[3][4];       // 2D array with 3 rows, each with 4 integers
8      var n: char[10] = "Hello, world!"; // 1D arrays with characters are strings
9      println(i);             // Function call; print i and a new line character
10     i = 20;                 // Assign a new value 20 for i
11     println(i);
12     l = {1, 2, 3, 4, 5};    // Assign a vector with 5 integers 1, 2, 3, 4, 5 in order
13     println(l);
14     k = '\\';               // Assign a char with new value '\\' (backslash)
15     println(k);
16     println(n);
17     n = "Another string";   /*Test C-style comments*/ n = "Third string";
18     println(n);
19     ret;                    // Return nothing to terminate the function body
20 }
```

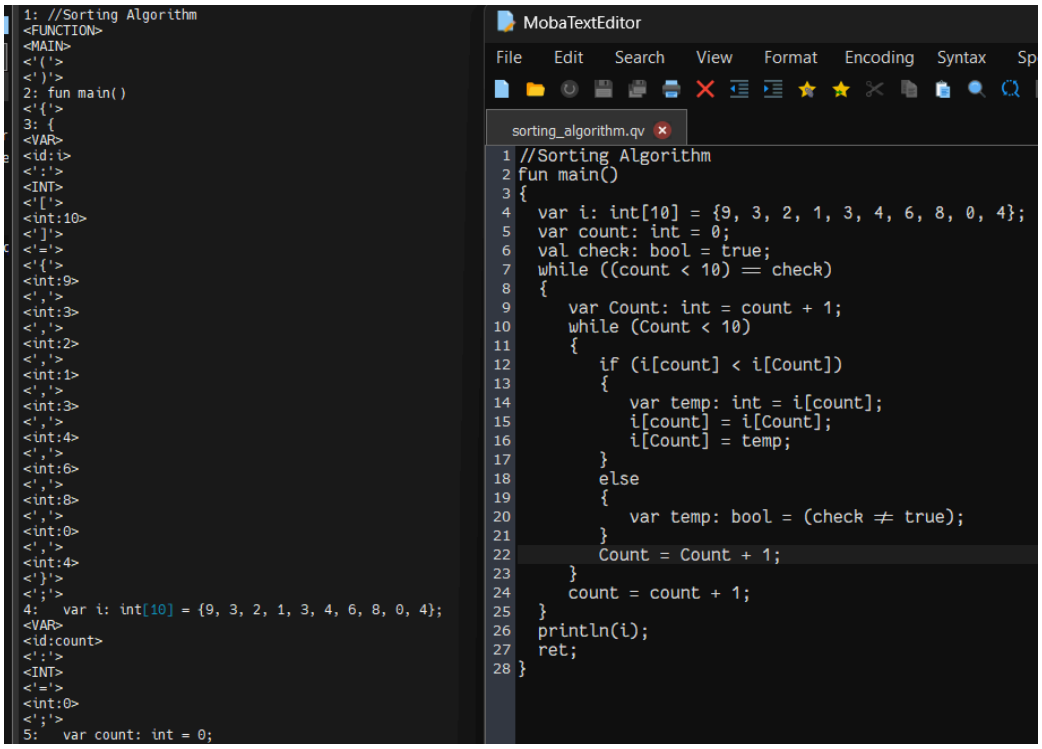
```
<id:l>
<'='>
<'{'>
<int:1>
<','>
<int:2>
<','>
<int:3>
<','>
<int:4>
<','>
<int:5>
<'}'>
<';'>
12:     l = {1, 2, 3, 4, 5};    // Assign a vector with 5 integers 1, 2, 3, 4, 5 in
rder
<PRINTLN>
<'('>
<id:l>
<')'>
<';'>
13:     println(l);
```

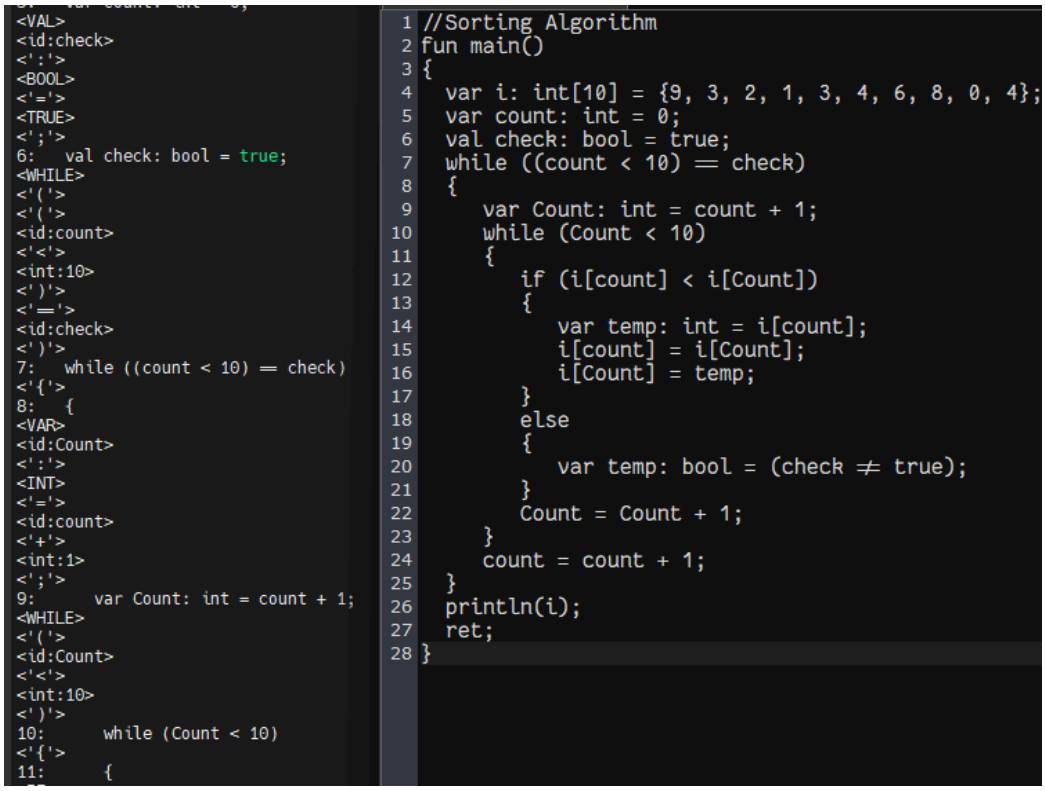
```
<id:k>
<'='>
<char:\>
<';'>
14:     k = '\\';               // Assign a char with new value '\\' (backslash)
<PRINTLN>
<'('>
<id:k>
<')'>
<';'>
15:     println(k);
```

```
<PRINTLN>
<'('>
<id:n>
<')'>
<';'>
16:     println(n);
<id:n>
<'='>
<string:Another string>
<';'>
<id:n>
<'='>
<string:Third string>
<';'>
17:     n = "Another string";   /*Test C-style comments*/ n = "Third string"
<PRINTLN>
<'('>
<id:n>
<')'>
<';'>
18:     println(n);
```

```
<RETURN>
<';'>
19:     ret;                    // Return nothing to terminate the function
<'}'>

Symbol Table:
0 i
1 j
2 k
3 l
4 m
5 n
b11015010@B:~/B11015010_scanner$
```

3.2 sorting_algorithm.qv





```
<IF>
<'('>
<id:i>
<'['>
<id:count>
<']'>
<'<'>
<id:i>
<'['>
<id:Count>
<']'>
<')'>
12:         if (i[count] < i[Count])
<'{'>
13:         {
<VAR>
<id:temp>
<':'>
<INT>
<'='>
<id:i>
<'['>
<id:count>
<']'>
<';'>
14:             var temp: int = i[count];
<id:i>
<'['>
<id:count>
<']'>
<'='>
<id:i>
<'['>
<id:Count>
<']'>
<';'>
15:             i[count] = i[Count];
<id:i>
<'['>
<id:Count>
<']'>
<'='>
<id:temp>
<';'>
16:             i[Count] = temp;
<'}'>
17:         }
<ELSE>
18:         else
```

sorting_algorithm.qv

```
1  //Sorting Algorithm
2  fun main()
3  {
4    var i: int[10] = {9, 3, 2, 1, 3, 4, 6, 8, 0, 4};
5    var count: int = 0;
6    val check: bool = true;
7    while ((count < 10) == check)
8    {
9       var Count: int = count + 1;
10      while (Count < 10)
11      {
12         if (i[count] < i[Count])
13         {
14            var temp: int = i[count];
15            i[count] = i[Count];
16            i[Count] = temp;
17         }
18         else
19         {
20            var temp: bool = (check ≠ true);
21         }
22         Count = Count + 1;
23      }
24      count = count + 1;
25   }
26   println(i);
27   ret;
28 }
```

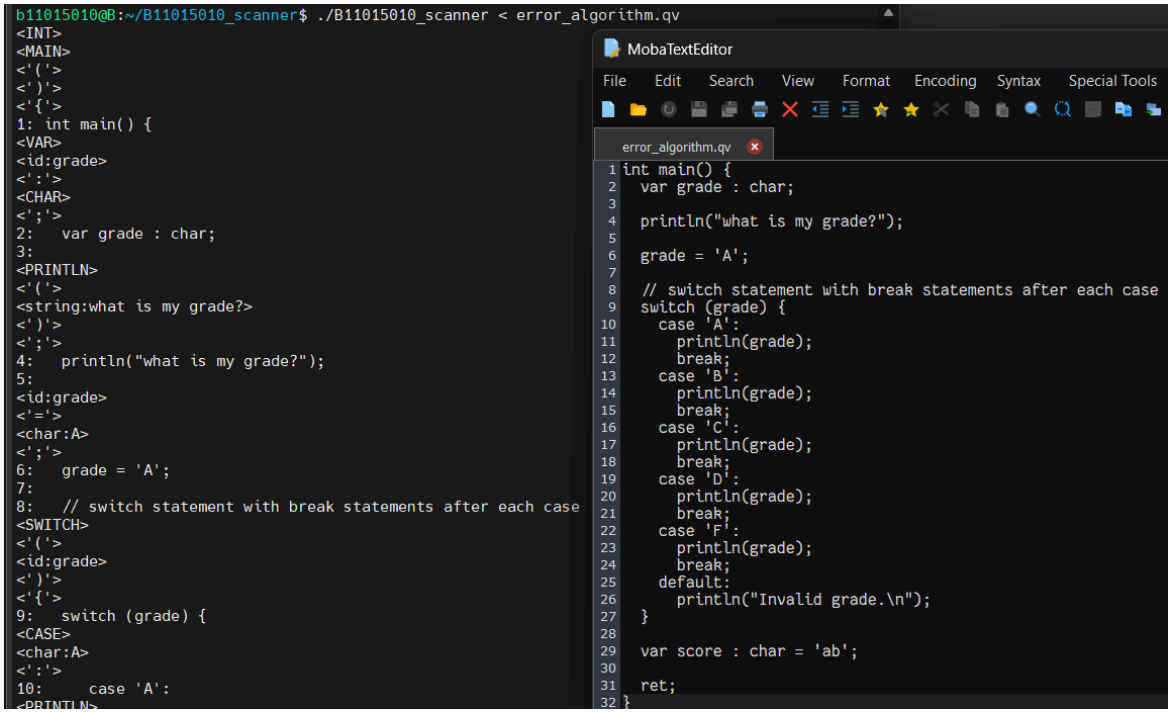
```
<'{'>
19:         {
<VAR>
<id:temp>
<':'>
<BOOL>
<'='>
<'('>
<id:check>
<'≠'>
<TRUE>
<')'>
<';'>
20:             var temp: bool = (check ≠
<'}'>
21:         }
<id:Count>
<'='>
<id:Count>
<'+'>
<int:1>
<';'>
22:         Count = Count + 1;
<'}'>
23:     }
<id:count>
<'='>
<id:count>
<'+'>
<int:1>
<';'>
24:     count = count + 1;
<'}'>
25:   }
<PRINTLN>
<'('>
<id:i>
<')'>
<';'>
26:   println(i);
<RETURN>
<';'>
27:   ret;
<'}'>
```

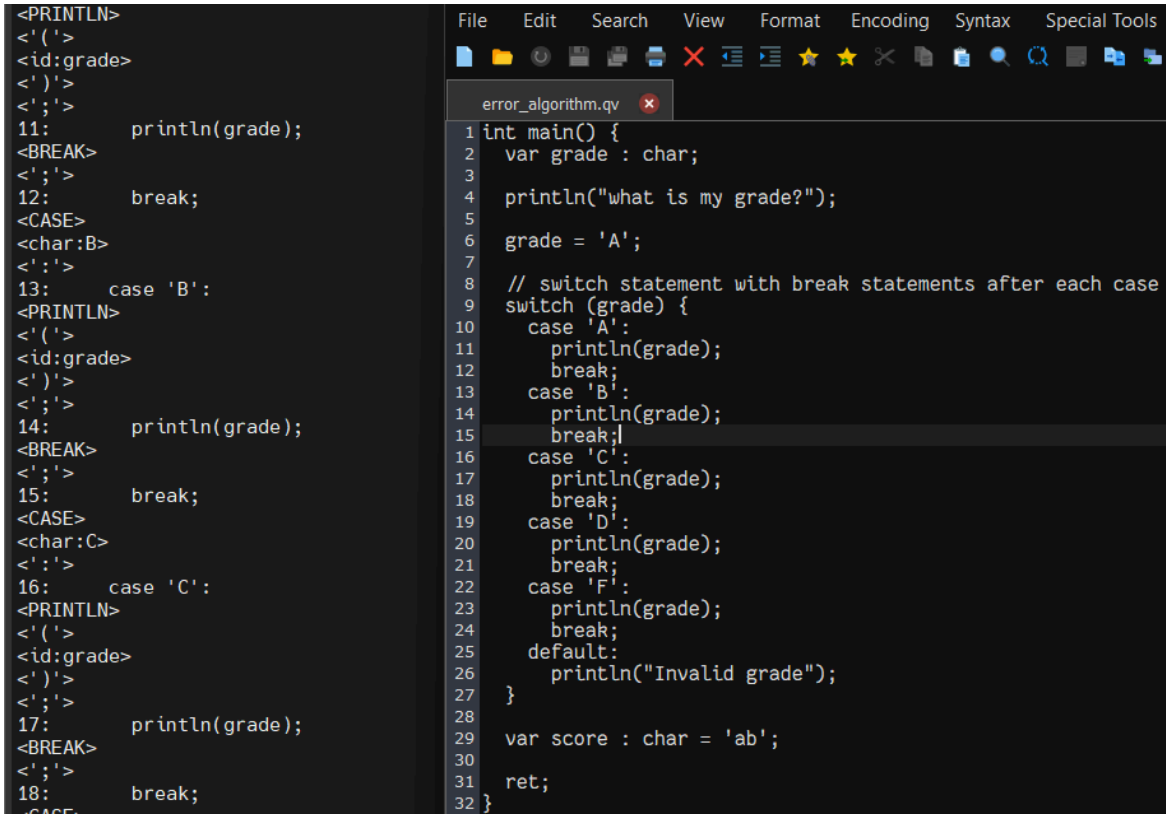
sorting_algorithm.qv

```
1  //Sorting Algorithm
2  fun main()
3  {
4    var i: int[10] = {9, 3, 2, 1, 3, 4, 6, 8, 0, 4};
5    var count: int = 0;
6    val check: bool = true;
7    while ((count < 10) == check)
8    {
9       var Count: int = count + 1;
10      while (Count < 10)
11      {
12         if (i[count] < i[Count])
13         {
14            var temp: int = i[count];
15            i[count] = i[Count];
16            i[Count] = temp;
17         }
18         else
19         {
20            var temp: bool = (check ≠ true);
21         }
22         Count = Count + 1;
23      }
24      count = count + 1;
25   }
26   println(i);
27   ret;
28 }
```

```
<'}'>

Symbol Table:
0 i
1 count
2 check
3 Count
4 temp
b11015010@B:~/B11015010_scanner$
```

3.3 error_algorithm.qv





```
<PRINTLN>
<'('>
<id:grade>
<')'>
<';'>
11:       println(grade);
<BREAK>
<';'>
12:       break;
<CASE>
<char:B>
<':'>
13:     case 'B':
<PRINTLN>
<'('>
<id:grade>
<')'>
<';'>
14:       println(grade);
<BREAK>
<';'>
15:       break;
<CASE>
<char:C>
<':'>
16:     case 'C':
<PRINTLN>
<'('>
<id:grade>
<')'>
<';'>
17:       println(grade);
<BREAK>
<';'>
18:       break;
```

File Edit Search View Format Encoding Syntax Special Tools

error_algorithm.qv

```
1  int main() {
2    var grade : char;
3
4    println("what is my grade?");
5
6    grade = 'A';
7
8    // switch statement with break statements after each case
9    switch (grade) {
10     case 'A':
11       println(grade);
12       break;
13     case 'B':
14       println(grade);
15       break;
16     case 'C':
17       println(grade);
18       break;
19     case 'D':
20       println(grade);
21       break;
22     case 'F':
23       println(grade);
24       break;
25     default:
26       println("Invalid grade");
27   }
28
29   var score : char = 'ab';
30
31   ret;
32 }
```

```
<CASE>
<char:D>
<':'>
19:     case 'D':
<PRINTLN>
<'('>
<id:grade>
<')'>
<';'>
20:         println(grade);
<BREAK>
<';'>
21:         break;
<CASE>
<char:F>
<':'>
22:     case 'F':
<PRINTLN>
<'('>
<id:grade>
<')'>
<';'>
23:         println(grade);
<BREAK>
<';'>
24:         break;
<DEFAULT>
<':'>
25:     default:
<PRINTLN>
<'('>
<string:Invalid grade>
<')'>
<';'>
26:         println("Invalid grade");
<'}'>
27:   }
28:
```

MobaTextEditor

File Edit Search View Format Encoding Syntax Special Tools

error_algorithm.qv

```
1  int main() {
2    var grade : char;
3
4    println("what is my grade?");
5
6    grade = 'A';
7
8    // switch statement with break statements after each case
9    switch (grade) {
10     case 'A':
11       println(grade);
12       break;
13     case 'B':
14       println(grade);
15       break;
16     case 'C':
17       println(grade);
18       break;
19     case 'D':
20       println(grade);
21       break;
22     case 'F':
23       println(grade);
24       break;
25     default:
26       println("Invalid grade");
27   }
28
29   var score : char = 'ab';
30
31   ret;
32 }
```

```
<PRINTLN>
<'('>
<string:Invalid grade>
<')'>
<';'>
26:         println("Invalid grade");
<'}'>
27:   }
28:
<VAR>
<id:score>
<':'>
<CHAR>
<'='>
b11015010@B:~/B11015010_scanner$
```

```
20         println(grade);
21         break;
22     case 'F':
23       println(grade);
24       break;
25     default:
26       println("Invalid grade");
27   }
28
29   var score : char = 'ab';
30
31   ret;
32 }
```

aborted because char can not accept more than 1 character, except escape sequences

3.4 commenting.qv

```
b11015010@B:~/B11015010_scanner$ ./B11015010_scanner < commenting.qv
1: /* sample program for food menu */
2:
3: /* ------------------------------
4: welcome everyone to order
5: ------------------------------ */
6:
<FUNCTION>
<MAIN>
<'('>
<')'>
<'{'>
7: fun main () {
8:    // this is main function that will run the code
9:
10:    /* ============================
11:    defining variables in this section
12:    ============================ */
13:
<VAR>
<id:food>
<':'>
<STRING>
<'='>
<string:burger>
<';'>
14:    var food : string = "burger"; //most ordered
<VAR>
<id:drink>
<':'>
<STRING>
<'='>
<string:cola>
<';'>
15:    var drink : string = "cola"; //limited flavour variant
16:
```

nsitive; *VECTOR, Vector, and vecto*

MobaTextEditor

File Edit Search View Format Encoding Syntax Special Tools

commenting.qv

```
1  /* sample program for food menu */
2
3  /* ------------------------------
4  welcome everyone to order
5  ------------------------------ */
6
7  fun main () {
8    // this is main function that will run the code
9
10   /* ============================
11   defining variables in this section
12   ============================ */
13
14   var food : string = "burger"; //most ordered
15   var drink : string = "cola"; //limited flavour variant
16
17   var set : char = 'a'; //menu set
18
19   var upcoming_menu : char = '/' /* upcoming food and drink */
20   var sold_out : char = '\' // use the backslash as none sold out
21 }
```

```
<VAR>
<id:set>
<':'>
<CHAR>
<'='>
<char:a>
<';'>
17:    var set : char = 'a'; //menu set
18:
<VAR>
<id:upcoming_menu>
<':'>
<CHAR>
<'='>
<char:/>
19:    var upcoming_menu : char = '/' /* upcoming food and drink */
<VAR>
<id:sold_out>
<':'>
<CHAR>
<'='>
```

aborted because char can't accept single blackslash

```
1  /* sample program for food menu */
2
3  /* ------------------------------
4  welcome everyone to order
5  ------------------------------ */
6
7  fun main () {
8    // this is main function that will run the code
9
10   /* ============================
11   defining variables in this section
12   ============================ */
13
14   var food : string = "burger"; //most ordered
15   var drink : string = "cola"; //limited flavour variant
16
17   var set : char = 'a'; //menu set
18
19   var upcoming_menu : char = '/' /* upcoming food and drink */
20   var sold_out : char = '\' // use the backslash as none sold out
21 }
```