

# Interpreter Construction Report

—A self-designed CMM interpreter

based on ANTLR

**Team member:**

何昊东 2014302580227

张树荣 2014302580014

柯磊 2014302580055

## Contents

1. Background .....	2
2. CMM Grammar .....	2
3. Design and Implementation .....	4
3.1 Experiment environment .....	4
3.2 Architecture design and main functions .....	4
3.3 Logical implementation based on ANTLR.....	5
4. Testing .....	7
5. How to use .....	12
6. Conclusion .....	12
6.1 Work Summary.....	12
6.2 Encountered problems.....	13
6.3 Experience and harvest.....	13
7. Reference .....	13

## 1. Background

Name	Student Number	Work content
何昊东	2014302580227	Lexical analysis/GUI design and implementation/Test script
张树荣	2014302580014	Semantic analysis/ Learning ANTLR API/ Generating the syntax tree/Project architecture
柯磊	2014302580055	Grammatical analysis /Learning ANTLR API /Test/Document

## 2. CMM Grammar

### Grammar overview:

1. Support arithmetic and relational operations
2. Support common type conversions, our base types contain: int, double, string, bool.
3. Support basic control flow: if-else, while, read, write, and so on.
4. Support assignment when declaration.
5. Utilizing the top-down LL (K) approach.
6. Add type string and arrays are declared with size information. The number of elements in an array is set when declared and cannot be changed.
7. Variables can be declared of base type and array type. Each variable has a level of scoping.

### Grammar representation:

```
grammar CMM;

program : (stmt)+

stmt : varDecl | ifStmt | whileStmt | breakStmt | assignStmt | readStmt | writeStmt | stmtBlock

stmtBlock : LBBRACKET (stmt)+ RBBRACKET

varDecl : type varList SEMICOLON

type : INT | DOUBLE | STRING #TypeString

array : IDENT LMBRACKET (INTCONSTANT | expr) RMBRACKET ;

varList : (IDENT | delassign | array)(COMMA (IDENT | delassign | array))*

elseiflist : elseif+

elseif : ELSEIF LSBRACKET expr RSBRACKET (stmtBlock | stmt)

ifStmt : IF LSBRACKET expr RSBRACKET (stmt | stmtBlock) #ONLYIF
      | IF LSBRACKET expr RSBRACKET (stmt | stmtBlock) ELSE (stmt | stmtBlock) #IFELSE
      | IF LSBRACKET expr RSBRACKET (stmt | stmtBlock) elseiflist #IFELSELIST
      | IF LSBRACKET expr RSBRACKET (stmt | stmtBlock) elseiflist ELSE (stmt | stmtBlock) #IFELSELISTELSE

whileStmt : WHILE LSBRACKET expr RSBRACKET (stmtBlock | stmt)

breakStmt : BREAK SEMICOLON

readStmt : READ LSBRACKET ((IDENT) | (array)) RSBRACKET SEMICOLON

writeStmt : WRITE LSBRACKET expr RSBRACKET SEMICOLON

assignStmt : value EQUAL expr SEMICOLON

delassign: IDENT EQUAL expr

value : (IDENT)|(array)

constant : (INTCONSTANT | DOUBLECONSTANT | STRINGCONSTANT) ##NUM
        | (TRUE | FALSE) ##BOOL

//优先级:  Comp < addMin < mulDiv < unaryMinus < Atom
expr : expr SEQUAL addMin | expr GEQUAL addMin | expr GREATER addMin | expr SMALLER addMin
     | expr DEQUAL addMin | expr NEQUAL addMin | addMin

addMin : addMin PLUS mulDiv | addMin MINUS mulDiv | mulDiv ;

mulDiv : mulDiv MULT unaryMinus | mulDiv DIV unaryMinus | unaryMinus ;

unaryMinus : MINUS unaryMinus | atom ;

atom : IDENT | constant | array | LSBRACKET expr RSBRACKET

READ : 'read';

WRITE : 'write';

IF : 'if';

ELSE : 'else';

ELSEIF : 'else if';

WHILE : 'while';

BREAK : 'break';

INT : 'int';

DOUBLE : 'double';

STRING : 'string';

BOOL : 'bool';

COMMA : ',';

SEMICOLON : ';';

IDENT : [A-Za-z_][A-Za-z0-9_]*;
```

```

INTCONSTANT : '0' | [1-9][0-9]* ;
DOUBLECONSTANT : INTCONSTANT('.'([0-9]+))? ;
STRINGCONSTANT : QUOTE SCharSequence? QUOTE

```

#### fragment

SCharSequence : SChar+

#### fragment

SChar: ~["\\|r\n| SimpleEscapeSequence /'\\n' / '\\r\n'

#### fragment

SimpleEscapeSequence : '\\'["'?abfntv\\];

### Other tokens

```

TRUE : 'true'  SEQUAL : '<='  SMALLER : '<'  GEQUAL : '>='  GREATER : '>'  DEQUAL : '=='  NEQUAL : '!='  EQUAL : '='
PLUS : '+'  MINUS : '-'  MULT : '*'  DIV : '/'  MOD : '%'  WS : ['\t\r\n]+ -> skip  FALSE : 'false'
LBBRACKET : '{'  RBBRACKET : '}'  LMBRACKET : '['  RMBRACKET : ']'  LSBRACKET : '('  RSBRACKET : ')'  QUOTE : '"'
SL_COMMENT : '//' ~[\r\n]* -> channel(HIDDEN)
MUL_COMMENT : '/*' .*? '*/' -> channel(HIDDEN)

```

## 3. Design and Implementation

### 3.1 Experiment environment

ANTLR 4.5.1

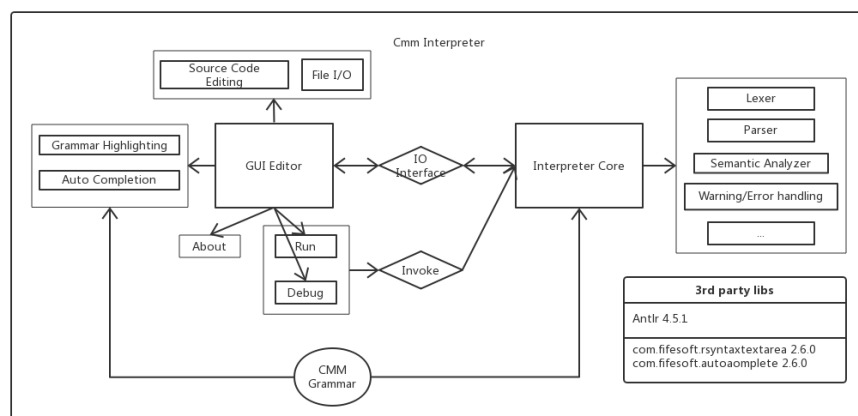
IDEA Ultimate 2016.2/3

JDK 1.8

### 3.2 Architecture design and main functions

#### 3.2.1 Architecture design

Two parts: Interpreter core part and GUI editor part.

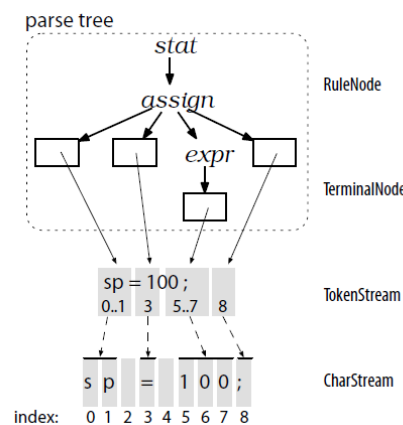


### 3.2.2 Main functions

1. Open/Save .cmm source file.
2. A GUI Editor custom-made for CMM grammar. Extra features include: (1) Grammar highlight (2) Key words auto-completion, some shorthand completions.(These completions don't require the input text to be the same thing as the replacement text) (3) Code folding (4) Other helpful features for editing code
3. Simply run a .cmm program or debug it, which will pop up a syntax tree window, meanwhile showing the lexical analysis results.
4. The interpreter support generating lexical analysis and syntax tree. If there is no error, the CMM codes will be translated, executed directly and show the corresponding result. If there are some errors, report them, so as to help the programmer fix the mistake and move on.
5. Help function will show the information of the developers.

## 3.3 Logical implementation based on ANTLR

To complete this compiler project, we have to finish four processes, lexical analysis, grammatical analysis, semantic analysis and execution and translation. ANTLR directly supported lexical analysis and generating syntax trees. Lexers process characters and pass tokens to the parser, which in turn checks syntax and creates a parse tree. The corresponding ANTLR classes are CharStream, Lexer, Token, Parser, and ParseTree. The “pipe” connecting the lexer and parser is called a TokenStream. The diagram below illustrates how objects of these types in our project connect to each other in memory:



To implement the parser, we utilized the ANTLR tool generates recursive-descent parsers from our designed grammar rules. Recursive-descent parsers are really just a collection of recursive methods, one per rule. The descent term refers to the fact that parsing begins at the root of a parse tree and proceeds toward the leaves report erratum (tokens). The rule we invoke first, the start symbol, becomes the root of the parse tree. That would mean calling method stat() for the parse tree in the previous section. A more general term for this kind of parsing is top-down parsing; recursive-descent parsers are just one kind of top-down parser implementation.

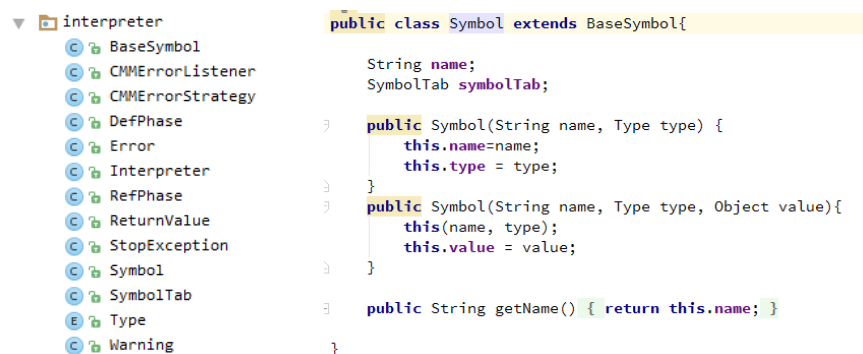
We utilized the two tree-walking mechanisms in its runtime library provide by ANTLR. By default, ANTLR generates a parse-tree listener interface that responds to events triggered by the

built-in tree walker. The convenient listener mechanism is all automatic which means we don't have to write a parse-tree walker, and our listener methods don't have to explicitly visit their children.

Our major work is to inherit from the parent classes and override and supplement it to finish our own semantic analysis part and execution part. Our code structures and design details for symbols are as following graphs. Also we added the warning and error report part to make it robust and friendly .We implemented for types of the grammar by adding the type string and its corresponding operation .The main problems we met is the variable domain conflicts, the logic of loop, if-else, break functions and so on which is fixed in the class Symbol, RefPhase and DefPhase.

```
public enum Type {
    tInt,
    tDouble,
    tIntArray,
    tDoubleArray,
    tBool,
    tString
}
```

Semantic analysis is also performed while parsing semantics .For all variables, we store the variable name, add a level, to indicate the scope of the variable. Variables declared at different levels belong to the corresponding level, if you leave the hierarchy, the responsibility should be removed to declare the hierarchy variable in the queue.



The screenshot shows an IDE with a project structure on the left and the code for the `Symbol` class on the right. The project structure includes:

- interpreter
  - BaseSymbol
  - CMMErrorListener
  - CMMErrorStrategy
  - DefPhase
  - Error
  - Interpreter
  - RefPhase
  - ReturnValue
  - StopException
  - Symbol
  - SymbolTab
  - Type
  - Warning

The `Symbol` class code is as follows:

```
public class Symbol extends BaseSymbol{
    String name;
    SymbolTab symbolTab;

    public Symbol(String name, Type type) {
        this.name=name;
        this.type = type;
    }
    public Symbol(String name, Type type, Object value){
        this(name, type);
        this.value = value;
    }

    public String getName() { return this.name; }
```

The logic for us to implement variable declaration is when the same scope to check whether there are duplicate variables, if not, then join the symbol table.Also we check whether the variable is declared, whether the two sides of the assignment type match, whether the array subscript is an integer.

```
public class SymbolTab {
    private SymbolTab enclosingSymbolTab;
    private Map<String, Symbol> symbolsMap = new LinkedHashMap<>();
    public SymbolTab(SymbolTab enclosingSymbolTab) { this.enclosingSymbolTab = enclosingSymbolTab; }
    public SymbolTab getEnclosingSymbolTab() { return enclosingSymbolTab; }
    //定义一个符号
    public void define(Symbol sym)
    {
        symbolsMap.put(sym.name, sym);
        sym.symbolTab = this;
    }
    //在当前作用域内是否重复
    public boolean redundant(String name) { return symbolsMap.get(name) != null; }
    //向上递归查找一个符号
    public Symbol resolve(String name)
    {
        Symbol s = symbolsMap.get(name);
        if ( s!=null ) return s;
        if ( enclosingSymbolTab != null ) return enclosingSymbolTab.resolve(name);
        return null;
    }
    public void clear() { symbolsMap.clear(); }
}
```

If in the process of building the tree, the source code and grammar is different, then throw an error, the use of Error and Warning Class.Compile the error message, terminate the compilation and output. The message contains the error message and the number of lines.

{...}

## 4. Testing

We mainly test **eleven** major functions of CMM, which includes:

- (1) Variable declaration (variable type, variable)
- (2) Assignment statement (assignment, whether the type of assignment on both sides match)
- (3) Expression operation (operator, operator priority judgment, divisor zero error)
- (4) If conditional statement (if, if-else)
- (5) Loop (While)
- (6) Loop nesting (if-while, if-if)
- (7) Array (including array implementation and array subscripts cross-border error)
- (8) Input and output (including input and output, and whether the input type match)
- (9) Annotation functions (single-line comments, multi-line comments)
- (10) Break function
- (11) Array sorting function

The following is a screenshot of the test cases and results:

1) Variable declaration (variable type, variable)

The type of variable: int, double, bool, string.

Text Editor Demo : test1\_变量声明.mmm

File Run Help

Text	Type
4	
line 24 :	
2	Text : write
23	Text : (
24	Text : 0
24	Text : )
12	Text : ;
line 25 :	
2	Text : )
20	
line 26 :	
2	Text : write
23	Text : (
23	Text : m
13	

```

1
2 int a_2;
3 double r_2,r;
4 double i,j=23,k;
5
6 int c = 23;
7 string s="fds";
8 double b[2];
9 b[0] = 23;
10 b[1] = 0;
11 if(b[1] == 0)
12 {
13     double c = 23.5;
14     if(c == 23.5)
15         write(1);
16     else
17         write(0);
18
19     double b[1];
20     b[0] = 0.05;
21     if(b[0] == 0.05)
22         write(1);
23     else
24         write(0);
25 }
26 write(m);

```

Console

```

warning: unmatched type in 'b'
           in line 9:0
warning: unmatched type in 'b'
           in line 10:0
1
1
fds

```

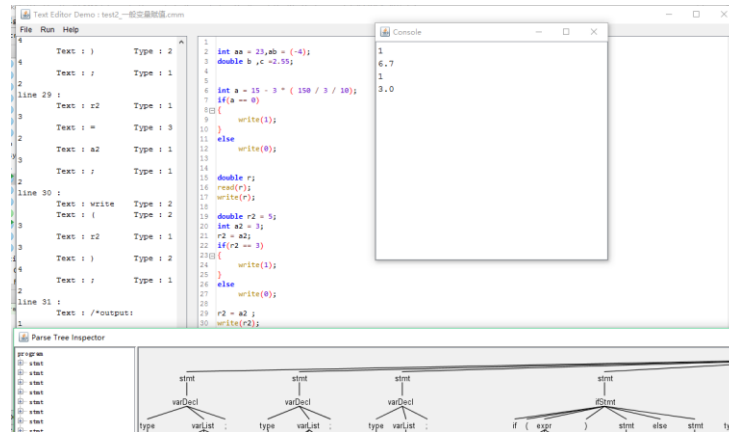
Parse Tree Inspector

```

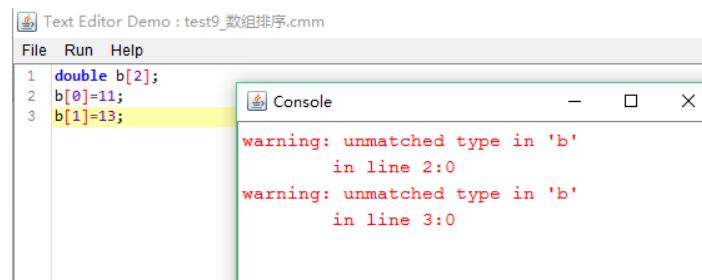
program
0: stmt
1: stmt
2: stmt
3: stmt
4: stmt
5: stmt
6: stmt
7: stmt
8: stmt
9: stmt
10: stmt
11: stmt
12: stmt
13: stmt
14: stmt
15: stmt
16: stmt
17: stmt
18: stmt
19: stmt
20: stmt
21: stmt
22: stmt
23: stmt
24: stmt
25: stmt
26: stmt
27: stmt
28: stmt
29: stmt
30: stmt
31: stmt
32: stmt
33: stmt
34: stmt
35: stmt
36: stmt
37: stmt
38: stmt
39: stmt
40: stmt
41: stmt
42: stmt
43: stmt
44: stmt
45: stmt
46: stmt
47: stmt
48: stmt
49: stmt
50: stmt
51: stmt
52: stmt
53: stmt
54: stmt
55: stmt
56: stmt
57: stmt
58: stmt
59: stmt
60: stmt
61: stmt
62: stmt
63: stmt
64: stmt
65: stmt
66: stmt
67: stmt
68: stmt
69: stmt
70: stmt
71: stmt
72: stmt
73: stmt
74: stmt
75: stmt
76: stmt
77: stmt
78: stmt
79: stmt
80: stmt
81: stmt
82: stmt
83: stmt
84: stmt
85: stmt
86: stmt
87: stmt
88: stmt
89: stmt
90: stmt
91: stmt
92: stmt
93: stmt
94: stmt
95: stmt
96: stmt
97: stmt
98: stmt
99: stmt
100: stmt
101: stmt
102: stmt
103: stmt
104: stmt
105: stmt
106: stmt
107: stmt
108: stmt
109: stmt
110: stmt
111: stmt
112: stmt
113: stmt
114: stmt
115: stmt
116: stmt
117: stmt
118: stmt
119: stmt
120: stmt
121: stmt
122: stmt
123: stmt
124: stmt
125: stmt
126: stmt
127: stmt
128: stmt
129: stmt
130: stmt
131: stmt
132: stmt
133: stmt
134: stmt
135: stmt
136: stmt
137: stmt
138: stmt
139: stmt
140: stmt
141: stmt
142: stmt
143: stmt
144: stmt
145: stmt
146: stmt
147: stmt
148: stmt
149: stmt
150: stmt
151: stmt
152: stmt
153: stmt
154: stmt
155: stmt
156: stmt
157: stmt
158: stmt
159: stmt
160: stmt
161: stmt
162: stmt
163: stmt
164: stmt
165: stmt
166: stmt
167: stmt
168: stmt
169: stmt
170: stmt
171: stmt
172: stmt
173: stmt
174: stmt
175: stmt
176: stmt
177: stmt
178: stmt
179: stmt
180: stmt
181: stmt
182: stmt
183: stmt
184: stmt
185: stmt
186: stmt
187: stmt
188: stmt
189: stmt
190: stmt
191: stmt
192: stmt
193: stmt
194: stmt
195: stmt
196: stmt
197: stmt
198: stmt
199: stmt
200: stmt
201: stmt
202: stmt
203: stmt
204: stmt
205: stmt
206: stmt
207: stmt
208: stmt
209: stmt
210: stmt
211: stmt
212: stmt
213: stmt
214: stmt
215: stmt
216: stmt
217: stmt
218: stmt
219: stmt
220: stmt
221: stmt
222: stmt
223: stmt
224: stmt
225: stmt
226: stmt
227: stmt
228: stmt
229: stmt
230: stmt
231: stmt
232: stmt
233: stmt
234: stmt
235: stmt
236: stmt
237: stmt
238: stmt
239: stmt
240: stmt
241: stmt
242: stmt
243: stmt
244: stmt
245: stmt
246: stmt
247: stmt
248: stmt
249: stmt
250: stmt
251: stmt
252: stmt
253: stmt
254: stmt
255: stmt
256: stmt
257: stmt
258: stmt
259: stmt
260: stmt
261: stmt
262: stmt
263: stmt
264: stmt
265: stmt
266: stmt
267: stmt
268: stmt
269: stmt
270: stmt
271: stmt
272: stmt
273: stmt
274: stmt
275: stmt
276: stmt
277: stmt
278: stmt
279: stmt
280: stmt
281: stmt
282: stmt
283: stmt
284: stmt
285: stmt
286: stmt
287: stmt
288: stmt
289: stmt
290: stmt
291: stmt
292: stmt
293: stmt
294: stmt
295: stmt
296: stmt
297: stmt
298: stmt
299: stmt
300: stmt
301: stmt
302: stmt
303: stmt
304: stmt
305: stmt
306: stmt
307: stmt
308: stmt
309: stmt
310: stmt
311: stmt
312: stmt
313: stmt
314: stmt
315: stmt
316: stmt
317: stmt
318: stmt
319: stmt
320: stmt
321: stmt
322: stmt
323: stmt
324: stmt
325: stmt
326: stmt
327: stmt
328: stmt
329: stmt
330: stmt
331: stmt
332: stmt
333: stmt
334: stmt
335: stmt
336: stmt
337: stmt
338: stmt
339: stmt
340: stmt
341: stmt
342: stmt
343: stmt
344: stmt
345: stmt
346: stmt
347: stmt
348: stmt
349: stmt
350: stmt
351: stmt
352: stmt
353: stmt
354: stmt
355: stmt
356: stmt
357: stmt
358: stmt
359: stmt
360: stmt
361: stmt
362: stmt
363: stmt
364: stmt
365: stmt
366: stmt
367: stmt
368: stmt
369: stmt
370: stmt
371: stmt
372: stmt
373: stmt
374: stmt
375: stmt
376: stmt
377: stmt
378: stmt
379: stmt
380: stmt
381: stmt
382: stmt
383: stmt
384: stmt
385: stmt
386: stmt
387: stmt
388: stmt
389: stmt
390: stmt
391: stmt
392: stmt
393: stmt
394: stmt
395: stmt
396: stmt
397: stmt
398: stmt
399: stmt
400: stmt
401: stmt
402: stmt
403: stmt
404: stmt
405: stmt
406: stmt
407: stmt
408: stmt
409: stmt
410: stmt
411: stmt
412: stmt
413: stmt
414: stmt
415: stmt
416: stmt
417: stmt
418: stmt
419: stmt
420: stmt
421: stmt
422: stmt
423: stmt
424: stmt
425: stmt
426: stmt
427: stmt
428: stmt
429: stmt
430: stmt
431: stmt
432: stmt
433: stmt
434: stmt
435: stmt
436: stmt
437: stmt
438: stmt
439: stmt
440: stmt
441: stmt
442: stmt
443: stmt
444: stmt
445: stmt
446: stmt
447: stmt
448: stmt
449: stmt
450: stmt
451: stmt
452: stmt
453: stmt
454: stmt
455: stmt
456: stmt
457: stmt
458: stmt
459: stmt
460: stmt
461: stmt
462: stmt
463: stmt
464: stmt
465: stmt
466: stmt
467: stmt
468: stmt
469: stmt
470: stmt
471: stmt
472: stmt
473: stmt
474: stmt
475: stmt
476: stmt
477: stmt
478: stmt
479: stmt
480: stmt
481: stmt
482: stmt
483: stmt
484: stmt
485: stmt
486: stmt
487: stmt
488: stmt
489: stmt
490: stmt
491: stmt
492: stmt
493: stmt
494: stmt
495: stmt
496: stmt
497: stmt
498: stmt
499: stmt
500: stmt
501: stmt
502: stmt
503: stmt
504: stmt
505: stmt
506: stmt
507: stmt
508: stmt
509: stmt
510: stmt
511: stmt
512: stmt
513: stmt
514: stmt
515: stmt
516: stmt
517: stmt
518: stmt
519: stmt
520: stmt
521: stmt
522: stmt
523: stmt
524: stmt
525: stmt
526: stmt
527: stmt
528: stmt
529: stmt
530: stmt
531: stmt
532: stmt
533: stmt
534: stmt
535: stmt
536: stmt
537: stmt
538: stmt
539: stmt
540: stmt
541: stmt
542: stmt
543: stmt
544: stmt
545: stmt
546: stmt
547: stmt
548: stmt
549: stmt
550: stmt
551: stmt
552: stmt
553: stmt
554: stmt
555: stmt
556: stmt
557: stmt
558: stmt
559: stmt
560: stmt
561: stmt
562: stmt
563: stmt
5
```

## (2) Assignment statement (assignment, whether the type of assignment on both sides match)

In this example, the variable of integer type a is transformed to the type double automatically.(from 3 to 3.0)

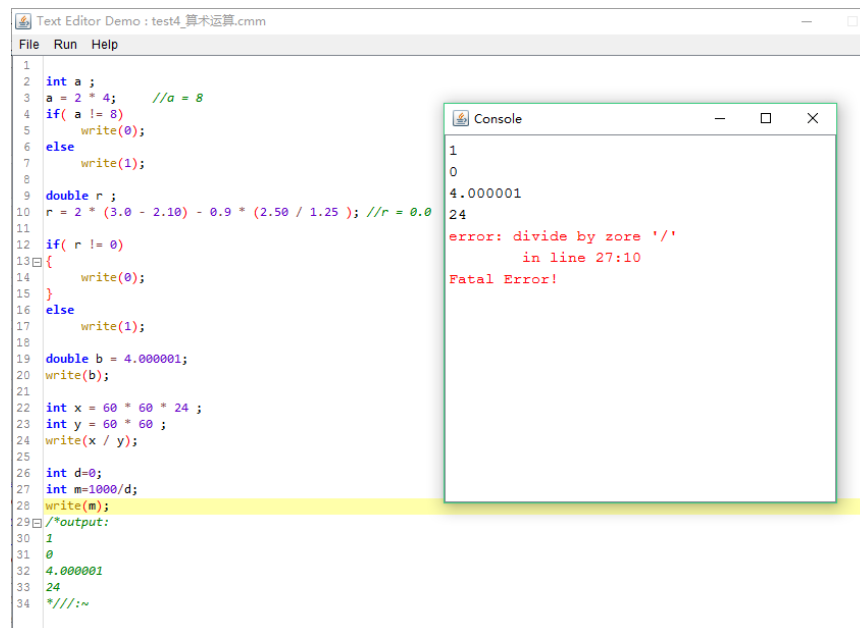


If the type of assignment on both sides doesn't match, the interpreter will report this error.



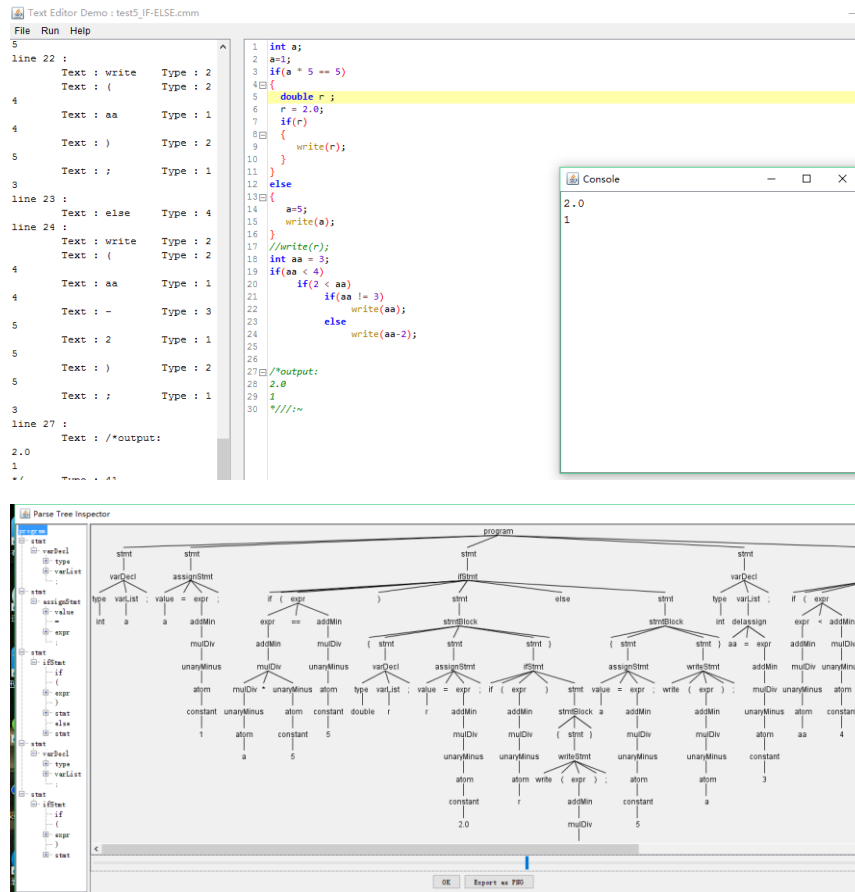
## (3) Expression operation (operator, operator priority judgment, divisor zero error)

The interpreter will report this error if divisor zero error occurred.

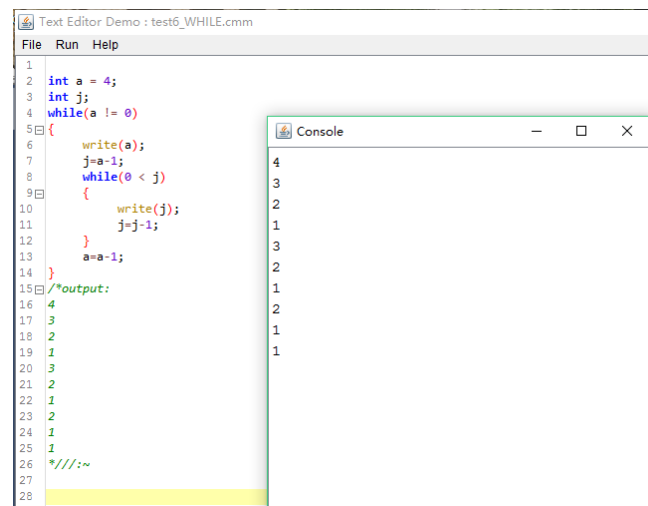


## (4) If conditional statement (if, if-else)





(5) Loop (While)



(6) Loop nesting (if-while, if-if)

The screenshot shows a text editor window titled "Text Editor Demo : test7\_IF-ELSE与WHILE.cmm". The code is as follows:

```

1 int a = 4;
2 while(a != 0)
3 {
4     int j = a;
5     while(j != 0)
6     {
7         if(j/2 != 1)
8             write(j);
9         j = j-1;
10    }
11    if(a < 2)
12    {
13        write(a);
14    }
15    else
16        write(a+3);
17    a = a - 1;
18 }
19 /*output:
20 4
21 1
22 7
23 1
24 6
25 1
26 5
27 1
28 1
29 1
30 */

```

The console window shows the output of the program:

```

4
1
7
1
6
1
5
1
1
1

```

(7) Array (including array implementation and array subscripts cross-border error)  
If the array subscripts cross-border, the interpreter will report this error.

The screenshot shows a text editor window titled "Text Editor Demo : test9\_数组排序.cmm". The code is as follows:

```

1 double b[2];
2 b[0]=11.1;
3 b[1]=9.2;
4 b[3]=18.4;
5
6

```

The console window shows the error message:

```

error: index out of boundary of array 'b'
      in line 4:0

```

If the assignment for an array doesn't match its type, the error will be reported:

The screenshot shows a text editor window titled "Text Editor Demo : test9\_数组排序.cmm". The code is as follows:

```

1
2
3 double R[6];
4 int I[2];
5
6 int i=0;
7
8
9 while(i<7)
10 {
11     R[i] = i;
12     i=i+1;
13 }
14 I[-2] = -1;
15 /*output:
16 */

```

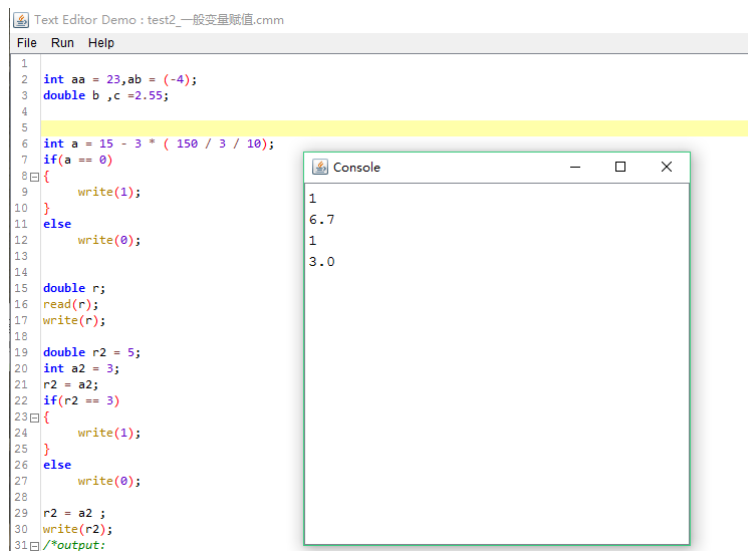
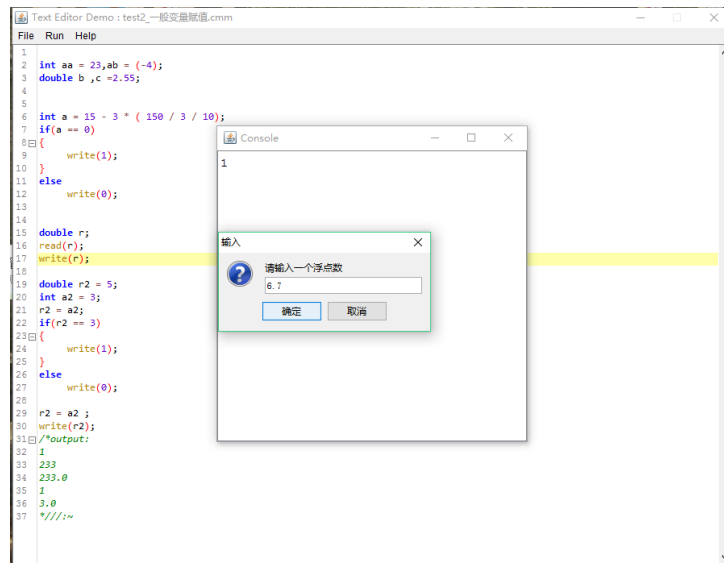
The console window shows the following messages:

```

warning: unmatched type in 'R'
      in line 11:1
warning: unmatched type in 'R'
      in line 11:1
warning: unmatched type in 'R'
      in line 11:1
warning: unmatched type in 'R'
      in line 11:1
warning: unmatched type in 'R'
      in line 11:1
warning: unmatched type in 'R'
      in line 11:1
error: index out of boundary of array 'R'
      in line 11:1
error: index out of boundary of array 'I'
      in line 15:0

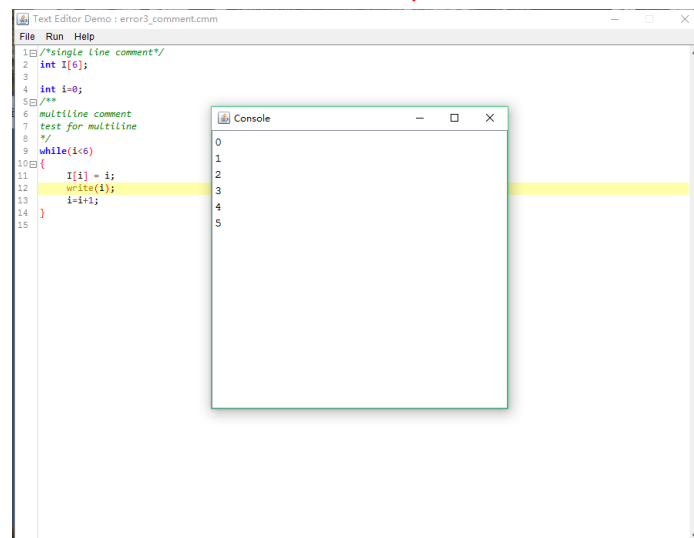
```

(8) Input and output (including input and output)

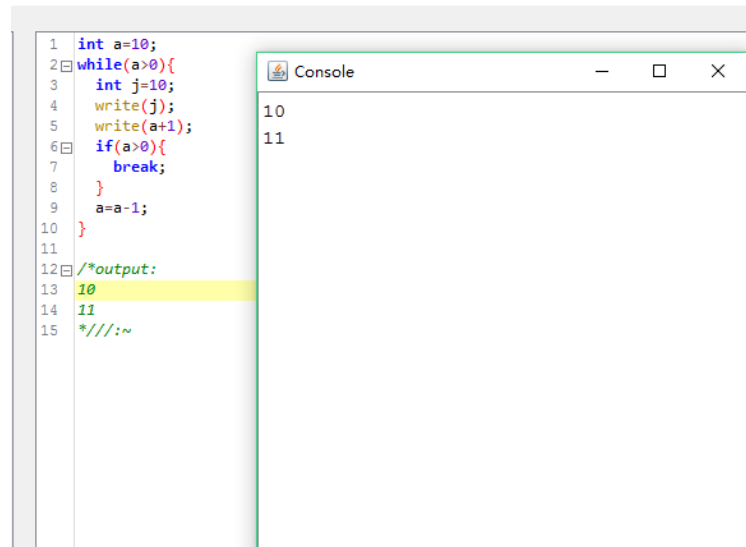


#### (9) Annotation functions (single-line comments, multi-line comments)

Single line and multiline comment both work correctly.



#### (10) Break function



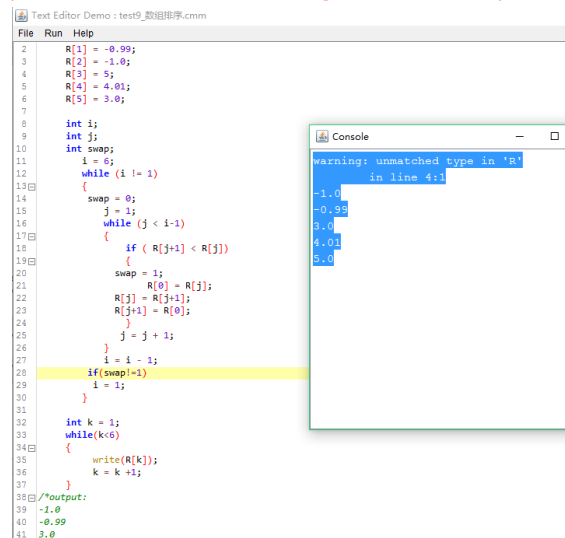
```
1 int a=10;
2 while(a>0){
3   int j=10;
4   write(j);
5   write(a+1);
6   if(a>0){
7     break;
8   }
9   a=a-1;
10 }
11
12 /*output:
13 10
14 11
15 *///:~
```

Console

```
10
11
```

## (11)Array sorting function

The element in the array was sorted in an ascending order correctly.



```
2 R[1] = -0.99;
3 R[2] = -1.0;
4 R[3] = 5;
5 R[4] = 4.01;
6 R[5] = 3.0;
7
8 int i;
9 int j;
10 int swap;
11 i = 6;
12 while (i != 1)
13 {
14   swap = 0;
15   j = 1;
16   while (j < i-1)
17   {
18     if ( R[j+1] < R[j])
19     {
20       swap = 1;
21       R[0] = R[j];
22       R[j] = R[j+1];
23       R[j+1] = R[0];
24     }
25     j = j + 1;
26   }
27   i = i - 1;
28   if (swap!=1)
29     i = 1;
30 }
31
32 int k = 1;
33 while(k<6)
34 {
35   write(R[k]);
36   k = k +1;
37 }
38 /*output:
39 -1.0
40 -0.99
41 3.0
```

Console

```
warning: unmatched type in 'R'
in line 4:1
-1.0
-0.99
3.0
4.01
5.0
```

## 5. How to use

Using the file menu to import and debug/ run menu to execute and run which is simple and convenient under the friendly GUI design.

## 6. Conclusion

### 6.1 Work Summary

In this work, we implemented the basic experimental requirements and completed a simple CMM language interpreter based on ANTLR and Java language. There are two types of text input ways to achieve a lexical analysis, importing from the source code file or direct text input. The

whole project contains two major parts, interpreter core part and GUI part. In the interpreter part, we successfully made Lexical analysis, grammatical analysis, Semantic Analysis and Execution and translation. Also we generated the parse tree and lexical analysis result, and output them to the interface through the GUI part in the debug mode.

By designing the grammar of the CMM language, we completed the creation of a simple high-level language and got a strongly typed static high-level language. Elementary / integer/Logical operations, if-else / while control flow, and basic IO interaction for read / write, etc. Finally, we tested this project with a lot of source code until it is correct and satisfying.

## 6.2 Encountered problems

During the development process, it takes us a lot a time to get familiar with the usage and API of ANTLR. Also, the problems of variable scope conflict and the implementation of break and nested loop also makes us blocked for several days. Grammatical analysis and semantic analysis are essentially based on the syntax tree of the formation / traversal process. So in order to eliminate the need for redundant code, making the code to maintain simple and easy to understand features, we put the syntax analysis and semantic analysis together, isolated the semantic analysis of the middle code. Fortunately, we fixed all this kind of problems one by one at last.

## 6.3 Experience and harvest

Through this practical project, we all have a deeper understanding of the process of compiler construction, including the compilation principles, design architecture and more details. Just implementing a simple, single syntax, weak functional interpreters are spent a lot of our effort and energy. Then what about the vast engineering, which contains the rigorous theory, coding skills and the awesome compilers that we used daily such as IntelliJ idea, eclipse and so on. Also, we have a better understanding of group cooperation. The process of putting forward ideas, the heated discussion and programming collaboratively within the group are all valuable and beneficial. Thanks to everyone in the team for their contributions to this project.

## 7. Reference

[1]Compiler.Construction-Principles.and.Practice,.K.C.Louden,.PWS.1997,.CMP.2002.592s

[2]Pragmatic. The Definitive ANTLR 4 Reference.2013

[3]Compilers: Principles, Techniques and Tools, Second Edition