



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

编译原理

大作业

(2019 年度春季学期)

姓 名	朱明彦
班 号	计算机 9 班
学 号	1160300314
学 院	计算机学院
教 师	辛明影

计算机科学与技术学院

目录

第 1 章 需求分析	3
第 2 章 关键字定义与文法设计	3
2.1 关键字定义	3
2.2 文法设计	4
第 3 章 系统设计	6
3.1 编译器框架	6
3.2 核心数据结构	7
3.3 主要函数功能	8
3.3.1 Lexer 主要函数功能	8
3.3.2 Parser 主要函数功能	8
第 4 章 系统实现	8
4.1 源文件	8
4.2 语法分析表	10
4.3 Token 序列	10
4.4 产生式序列	11
4.5 中间代码	14
4.6 附加：代码优化	15
A 源代码	16
B GUI 展示	17
C 参考文献	22

编译原理大作业

第 1 章 需求分析

在词法分析、语法分析和语义分析的实验基础之上,结合代码优化等技术,将之前的 `Lexer`, `Parser` 结合起来,形成一个完整的编译器前端程序。

第 2 章 关键字定义与文法设计

2.1 关键字定义

在词法分析部分对于关键字的定义是通过 Java 中的枚举类型来实现。

```
1 public enum Tag {
2     INT("int"), FLOAT("float"), BOOL("bool"), CHAR("char"), RECORD("record"),
3     IF("if"), ELSE("else"), DO("do"), WHILE("while"),
4     BREAK("break"), CONTINUE("continue"), TRUE("true"),
5     FALSE("false"), RETURN("return"), // keyword
6
7     ADD("+"), SUB("-"), MUL("*"), DIV("/"), MOD("%"), // arithmetic op
8
9     NE("!="), G(">"), GE(">="), L("<"), LE("<="), EQ("=="),
10    AND("&&"), OR("||"), NOT("!"), // logical op
11
12    SLP("("), SRP(")"), LP("{"), RP("}"), MLP("["), MRP("]"),
13    ASSIGN("="), SEMICOLON(";"), COMMA(","), // delimiters
14
15    REAL("real"), // float number
16    NUM("num"), // integer number
17    ID("id"), // identifier
18    STRING("string"),
19    STACK_BOTTOM(Parser.STACK_BOTTOM_CHARACTER),
20    PROC("proc"), CALL("call"),
21    NULL("null");
22 // 仅列出关键字部分 其余详见 src/lexer/Tag.java
23 }
```

2.2 文法设计

最终实现的文法以及语义动作的定义如下所示，其中红色字体为该产生式对应的语义动作。

1. $\text{Start} \rightarrow P$
2. $P \rightarrow P\text{Start } D P \mid P\text{Start } S P \mid \epsilon$
3. $P\text{Start} \rightarrow \epsilon \{ \text{env} = \text{new Env}(\text{env}); \text{offsetStack.push}(\text{offset}); \text{offset}=0; \}$
4. $D \rightarrow \text{proc } X \text{ id } (M) DM P \{ \text{pop}(\text{tableStack}); \text{pop}(\text{offset}) \} \mid \text{record id } P \mid T \text{ id } A ; \{ \text{enter}(\text{id.lexeme}, T.\text{type}, \text{offset}); \text{offset} = \text{offset} + T.\text{width}; \}$
5. $DM \rightarrow \epsilon \{ \text{table} = \text{mkTable}(\text{top}(\text{tableStack})); \text{push}(\text{table}); \text{push}(\text{offset}); \text{offset} = 0; \}$
6. $A \rightarrow = F A \mid , \text{id } A \mid \epsilon$
7. $M \rightarrow M , X \text{ id } \{ \text{enter}(\text{id.lexeme}, X.\text{type}, \text{offset}); \text{offset} = \text{offset} + X.\text{width}; M.\text{size} = M1.\text{size} + 1; \} \mid X \text{ id } \{ \text{enter}(\text{id.lexeme}, X.\text{type}, \text{offset}); \text{offset} = \text{offset} + X.\text{width}; M.\text{size} = 1; \}$
8. $T \rightarrow X \{ t = X.\text{type}; w = X.\text{width}; \} C \{ T.\text{type} = C.\text{type}; T.\text{width} = C.\text{width}; \}$
9. $X \rightarrow \text{int} \{ X.\text{type} = \text{interger}; X.\text{width} = 4; \} \mid \text{float} \{ X.\text{type} = \text{float}; X.\text{width} = 8; \} \mid \text{bool} \mid \text{char}$
10. $C \rightarrow [\text{num}] C \{ C.\text{type} = C1.\text{type} + '[' + \text{num.value} + ']' ; C.\text{width} = \text{num.value} * C1.\text{width}; \} \mid \epsilon \{ C.\text{type} = t; C.\text{width} = w; \}$
11. $S \rightarrow \text{id} = E ; \{ S.\text{nextList} = \text{null}; p = \text{loopUp}(\text{id.lexeme}); \text{if } p == \text{null} \text{ then error else } \text{gen}(p, '=', E.\text{addr}); \} \mid \text{if } (B) BM S N \text{ else } BM S \{ \text{backpatch}(B.\text{trueList}, BM1.\text{instr}); \text{backpatch}(B.\text{falseList}, BM2.\text{instr}); \text{temp} = \text{merge}(S1.\text{nextList}, N.\text{nextList}); S.\text{nextList} = \text{merge}(\text{temp}, S2.\text{nextList}); \} \mid \text{while } BM (B) BM S \{ \text{backpatch}(S1.\text{nextList}, BM1.\text{instr}); \text{backpatch}(B.\text{trueList}, BM2.\text{instr}); S.\text{nextList} = B.\text{falseList}; \text{gen}('goto', BM1.\text{instr}); \} \mid$

```

        call id ( Elist ) ; | return E ; | if ( B ) BM S {backpatch(B.trueList,
BM.instr); S.nextList = merge(B.falseList, S1.nextList); } |

        L = E ; {gen(L.array, L.addr, '=', E.addr)}

12. N -> ε {N.nextList = makeList(nextInstr); gen('goto'); }

13. L -> L [ E ] {L.array = L1.array; L.type = L1.type.elem; L.width = L.type.width;
t = new Temp(); L.addr = new Temp(); gen(L.addr, '=', E.addr, '*', L.width);
gen(L.addr, '=', L1.addr, '+', t); } |

        id [ E ] {p = lookUp(id.lexeme); if p == null then error else L.array
= p; L.type = id.type; L.addr = new Temp(); gen(L.addr, 'addr', E.addr, '*',
L.width)}

14. E -> E + G {E.addr = newTemp(); gen(E.addr, '=', E1.addr, '+', G.addr);} |
        G {E.addr = G.addr;}

15. G -> G * F {G.addr = newTemp(); gen(G.addr, '=', G1.addr, '*', F.addr);} |
        F {G.addr = F.addr;}

16. F -> ( E ) {F.addr = E.addr;} |
        num {F.addr = num.value;} |
        id {F.addr = lookup(id.lexeme); if F.addr == null then error;} | real {F.addr
= real.value;} | string | L {F.addr = L.array + '[' + L.addr']}'

17. B -> B || BM H {backpatch(B1.falseList, BM.instr); B.trueList = merge(B1.trueList,
H.trueList); B.falseList = H.falseList;} |
        H {B.trueList = H.trueList; B.falseList = H.falseList;}

18. H -> H && BM I {backpatch(H1.trueList, BM.instr); H.trueList = I.trueList;
H.falseList = merge(H1.falseList, I.falseList);} |
        I {H.trueList = I.trueList; H.falseList = I.falseList;}

19. I -> ! I {I.trueList = I1.falseList; I.falseList = I1.falseList;} |
        ( B ) {I.trueList = B.trueList; I.falseList = B.falseList;} |
        E Relop E {I.trueList = makeList(nextInstr); I.falseList = makeList(nextInstr
+ 1); gen('if', E1.addr, Relop.op, E2.addr, 'goto'); gen('goto');} |
        true {I.trueList = makeList(nextInstr); gen('goto');} |
        false {I.falseList = makeList(nextInstr); gen('goto');}

```

20. `BM -> ε {BM.instr = nextInstr}`

21. `Relop -> < | <= | > | >= | == | != {Relop.op = op}`

22. `Elist -> Elist , E {Elist.size = Elist1.size + 1;} | E {Elist.size = 1;}`

为了语义动作的设计，以上文法为二义文法，在 Parser 中设计为移进优先于规约，解决了 else 悬空的问题，具体见 `src/parser/Parser.java`。

第 3 章 系统设计

3.1 编译器框架

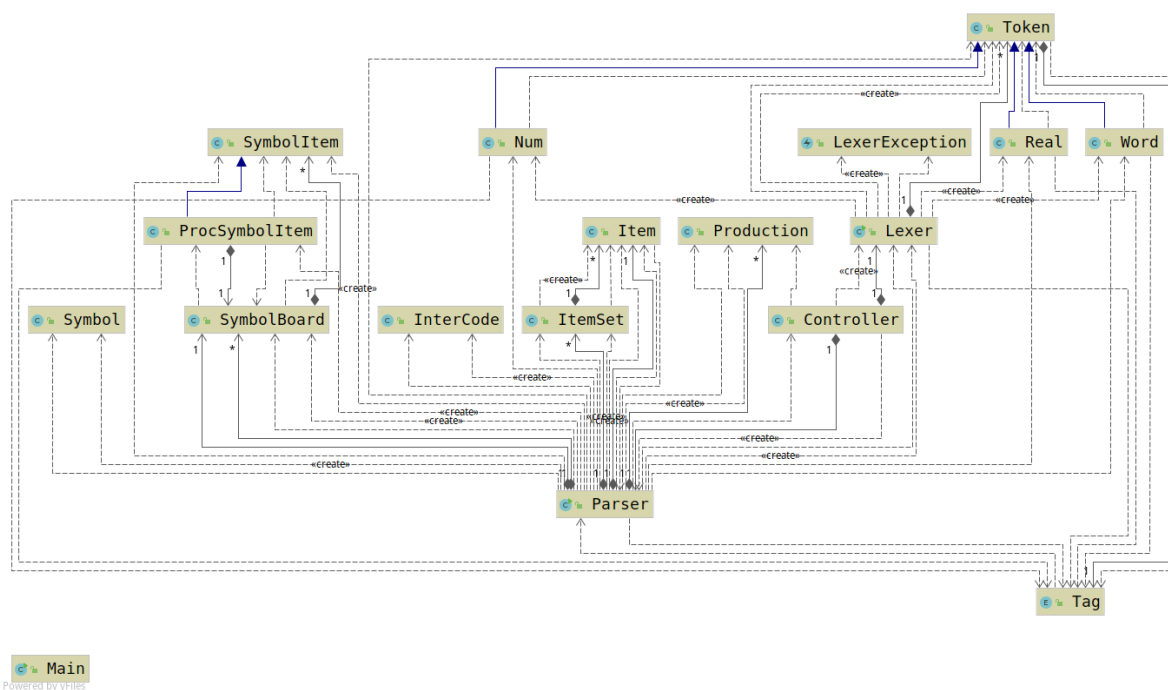


图 3.1: 编译器框架图

对于大作业最终实现的编译器，其 UML 图如3.1所示，下面分别对其中不同的实现进行阐述。

- **Lexer** 为整个编译器的词法分析器部分，其可以将输入的一类 C 语言源文件转化为 Token 序列，并作为下一步语法分析器的输入。

- `Parser` 为整个编译器的语法分析器部分，在本项目中实现了 LR(1) 分析法，并在语法分析的同时进行语义动作，即所有的语义动作作用的属性均为 S 属性 [1]，最终结果为中间代码。
- `Controller` 为整个编译器 GUI 的控制部分，其使用 JavaFX 实现。
- `Main` 为整个编译器的主程序部分，启动 GUI 形式的编译器，调用 `Main` 对应的主函数即可。
- `Token`, `Real`, `Num`, `Word` 分别为 `Lexer` 输出的 `Token` 的父类、浮点型常数类、整数常数类以及标识符和字符串型常数类的父类。
- `Tag` 为标注 `Token` 类别的枚举类型。
- `LexerException` 为发现词法分析错误抛出的异常类。
- `Production` 为语法中的产生式类，分别记录产生式左部和右部。
- `Item`, `ItemSet` 分别为项目类和项目集类，其中项目类即在进行 LR(1) 语法分析时，对应的“项目”概念的类，即文法中的一个产生式和位于它的右部中某处的点组成；项目集类，则是项目的集合。
- `InterCode` 为 `Parser` 最终结果对应的中间代码类。
- `SymbolItem`, `ProcSymbolItem` 分别对应符号表中表项的父类和符号表中方法表项类，其区别为方法表项中包含其对应的局部符号表。
- `SymbolBoard` 为符号表类，其包含的表项均为 `SymbolItem` 类。
- `Symbol` 为产生式中的符号类，或者称为非终结符类，其在产生式中的右部时可能会包含 S 属性，方便语义分析时的语义动作处理。

3.2 核心数据结构

在本项目的所实现的编译器中，所用的数据结构有栈、队列、集合、Map 和多维数组，其中栈为最重要的数据结构。下面分别阐述各个数据结构在编译器中起到的作用。

- **栈 (Stack)**，作为核心数据结构，在 `Parser` 中使用栈记录调用关系，包括记录 offset 和符号表。
- **队列 (Queue)**，在 `Lexer` 中作为输入缓冲实现，其作用是进行一些有二义符号，如 `>`, `>=`。
- **集合 (Set)**，其多次出现在本项目的实现中，分别作为 `Lexer` 中各种符号的记录，`Parser` 中产生式、终结符、非终结符以及项目集族的记录。

- **Map**，其多次出现在本项目实现的 Compiler 中，如记录标识符与其对应的符号表条目、LR(1) 分析法中记录非终结符的 First 集以及每个项目闭包的 GOTO 表。
- **Array**，主要用于记录 LR(1) 分析表。

3.3 主要函数功能

3.3.1 Lexer 主要函数功能

```
public void scan();
```

对于 Lexer 中的 scan 函数而言，主要功能为将输入的源文件转化为对应的 Token 序列，负责完成 Lexer 的主要逻辑。

3.3.2 Parser 主要函数功能

1.

```
public List<Production> reduce(List<Token> tokens);
```

对于 Parser 中的 reduce 函数而言，由于在进行语法分析的同时可以进行语义分析，所以 reduce 函数完成了从词法分析器的 Token 序列，到语法分析的产生式序列的转换。并且在函数返回之前，将最终的中间代码进行代码优化并输出。

2.

```
public void items();
```

在 Parser 中另一个核心函数就是 items，其实现的参考算法见 [1] 在 LR(1) 分析法中提到的 items() 算法，主要实现了解析语法文件并生成对应的 LR 分析表的过程。

第 4 章 系统实现

以下内容主要介绍在本项目中实现的 Compiler 里，用于测试完成情况的文件，以及中间结果的展示。

4.1 源文件

```
1    int a;
2    a = 1 + 2;
3    int b;
4    int c;
5    c = 10;
6    b = c + 1;
7    int d;
8    d = c * 2;
```



```
9      int e;
10     e = 0;
11     int x;
12     int y;
13     y = 999;
14     int z;
15     z = 100;
16     while (a < b)
17         if (c < d) x = y + z; else x = a + b;
18
19     a = b + c * (d + e);
20
21     if(a > b)
22         c = d;
23
24     proc float function(float i){
25         i = i + 1;
26         return i;
27     }
28
29     int [2][3] list;
30     int c;
31     int i;
32     int j;
33     int d;
34     float h;
35     d = c + list[i][j];
36     list[i][j] = c;
37
38     d = h * c;
39
40     c[1][2] = d;
41     proc int function(int a, int c){
42         a = c + 10;
43         int d;
44         return a;
45     }
```

在以上的源文件中有变量的声明（包括普通变量和多维数组变量），变量的使用，分支结构（*if-else*），循环结构（*while*）以及函数的声明（使用 *proc* 关键字）等不同的语句。

4.2 语法分析表

	&&		<=	...	A	B	...
0							
1							
2							
3							
...							
58						111	
59			r(F ->real)				
60	r(I ->true)	r(I ->true)					
61		s160					
...							
398							

表 1: LR 分析表（部分）

经过 Parser 的 `items` 函数处理在第2.2节中定义的文法，经过 LR(1) 分析得到的分析表如表1所示。表1中仅仅展示了部分 LR 分析表的内容，共有 398 个状态，对于终结符如左侧的 &&，其对应的状态可能有移进如 `s160` 和规约如 `r(I->true)` 动作；对于非终结符如右侧的 A,B 等，其表示在相应状态和栈顶符号时应该转移的状态，即 GOTO 表。

完整的 LR(1) 分析表见 `src/parser/LRTable.txt` 所示。

4.3 Token 序列

1	<INT>	15	<SEMICOLON>	29	<ID, d>	43	<ID, x>
2	<ID, a>	16	<ID, c>	30	<ASSIGN>	44	<SEMICOLON>
3	<SEMICOLON>	17	<ASSIGN>	31	<ID, c>	45	<INT>
4	<ID, a>	18	<NUM, 10>	32	<MUL>	46	<ID, y>
5	<ASSIGN>	19	<SEMICOLON>	33	<NUM, 2>	47	<SEMICOLON>
6	<NUM, 1>	20	<ID, b>	34	<SEMICOLON>	48	<ID, y>
7	<ADD>	21	<ASSIGN>	35	<INT>	49	<ASSIGN>
8	<NUM, 2>	22	<ID, c>	36	<ID, e>	50	<NUM, 999>
9	<SEMICOLON>	23	<ADD>	37	<SEMICOLON>	51	<SEMICOLON>
10	<INT>	24	<NUM, 1>	38	<ID, e>	52	<INT>
11	<ID, b>	25	<SEMICOLON>	39	<ASSIGN>	53	<ID, z>
12	<SEMICOLON>	26	<INT>	40	<NUM, 0>	54	<SEMICOLON>
13	<INT>	27	<ID, d>	41	<SEMICOLON>	55	<ID, z>
14	<ID, c>	28	<SEMICOLON>	42	<INT>	56	<ASSIGN>

57	<NUM, 100>	96	<IF>	135	<SEMICOLON>	174	<ID, c>
58	<SEMICOLON>	97	<SLP>	136	<INT>	175	<SEMICOLON>
59	<WHILE>	98	<ID, a>	137	<ID, i>	176	<ID, c>
60	<SLP>	99	<G>	138	<SEMICOLON>	177	<MLP>
61	<ID, a>	100	<ID, b>	139	<INT>	178	<NUM, 1>
62	<L>	101	<SRP>	140	<ID, j>	179	<MRP>
63	<ID, b>	102	<ID, c>	141	<SEMICOLON>	180	<MLP>
64	<SRP>	103	<ASSIGN>	142	<INT>	181	<NUM, 2>
65	<IF>	104	<ID, d>	143	<ID, d>	182	<MRP>
66	<SLP>	105	<SEMICOLON>	144	<SEMICOLON>	183	<ASSIGN>
67	<ID, c>	106	<PROC>	145	<FLOAT>	184	<ID, d>
68	<L>	107	<FLOAT>	146	<ID, h>	185	<SEMICOLON>
69	<ID, d>	108	<ID, function>	147	<SEMICOLON>	186	<PROC>
70	<SRP>	109	<SLP>	148	<ID, d>	187	<INT>
71	<ID, x>	110	<FLOAT>	149	<ASSIGN>	188	<ID, function>
72	<ASSIGN>	111	<ID, i>	150	<ID, c>	189	<SLP>
73	<ID, y>	112	<SRP>	151	<ADD>	190	<INT>
74	<ADD>	113	<LP>	152	<ID, list>	191	<ID, a>
75	<ID, z>	114	<ID, i>	153	<MLP>	192	<COMMA>
76	<SEMICOLON>	115	<ASSIGN>	154	<ID, i>	193	<INT>
77	<ELSE>	116	<ID, i>	155	<MRP>	194	<ID, c>
78	<ID, x>	117	<ADD>	156	<MLP>	195	<SRP>
79	<ASSIGN>	118	<NUM, 1>	157	<ID, j>	196	<LP>
80	<ID, a>	119	<SEMICOLON>	158	<MRP>	197	<ID, a>
81	<ADD>	120	<RETURN>	159	<SEMICOLON>	198	<ASSIGN>
82	<ID, b>	121	<ID, i>	160	<ID, list>	199	<ID, c>
83	<SEMICOLON>	122	<SEMICOLON>	161	<MLP>	200	<ADD>
84	<ID, a>	123	<RP>	162	<ID, i>	201	<NUM, 10>
85	<ASSIGN>	124	<INT>	163	<MRP>	202	<SEMICOLON>
86	<ID, b>	125	<MLP>	164	<MLP>	203	<INT>
87	<ADD>	126	<NUM, 2>	165	<ID, j>	204	<ID, d>
88	<ID, c>	127	<MRP>	166	<MRP>	205	<SEMICOLON>
89	<MUL>	128	<MLP>	167	<ASSIGN>	206	<RETURN>
90	<SLP>	129	<NUM, 3>	168	<ID, c>	207	<ID, a>
91	<ID, d>	130	<MRP>	169	<SEMICOLON>	208	<SEMICOLON>
92	<ADD>	131	<ID, list>	170	<ID, d>	209	<RP>
93	<ID, e>	132	<SEMICOLON>	171	<ASSIGN>	210	<STACK_BOTTOM>
94	<SRP>	133	<INT>	172	<ID, h>		
95	<SEMICOLON>	134	<ID, c>	173	<MUL>		

经过 Lexer 的 `scan` 函数，将第4.1节的源代码转化成对应的 Token 序列，即上面所示。

其中，对于关键字，如 `int` 等，其 Token 对应没有第二维的值，即 `<INT>`；而对于如标识符 `a`，其 Token 表示为 `<ID, a>`。

4.4 产生式序列

1	X -> int	6	F -> num
2	C -> epsilon	7	G -> F
3	T -> X C	8	E -> G
4	A -> epsilon	9	F -> num
5	D -> T id A ;	10	G -> F

11	E -> E + G	64	F -> num
12	S -> id = E ;	65	G -> F
13	X -> int	66	E -> G
14	C -> epsilon	67	S -> id = E ;
15	T -> X C	68	X -> int
16	A -> epsilon	69	C -> epsilon
17	D -> T id A ;	70	T -> X C
18	X -> int	71	A -> epsilon
19	C -> epsilon	72	D -> T id A ;
20	T -> X C	73	F -> num
21	A -> epsilon	74	G -> F
22	D -> T id A ;	75	E -> G
23	F -> num	76	S -> id = E ;
24	G -> F	77	BM -> epsilon
25	E -> G	78	F -> id
26	S -> id = E ;	79	G -> F
27	F -> id	80	E -> G
28	G -> F	81	Relop -> <
29	E -> G	82	F -> id
30	F -> num	83	G -> F
31	G -> F	84	E -> G
32	E -> E + G	85	I -> E Relop E
33	S -> id = E ;	86	H -> I
34	X -> int	87	B -> H
35	C -> epsilon	88	BM -> epsilon
36	T -> X C	89	F -> id
37	A -> epsilon	90	G -> F
38	D -> T id A ;	91	E -> G
39	F -> id	92	Relop -> <
40	G -> F	93	F -> id
41	F -> num	94	G -> F
42	G -> G * F	95	E -> G
43	E -> G	96	I -> E Relop E
44	S -> id = E ;	97	H -> I
45	X -> int	98	B -> H
46	C -> epsilon	99	BM -> epsilon
47	T -> X C	100	F -> id
48	A -> epsilon	101	G -> F
49	D -> T id A ;	102	E -> G
50	F -> num	103	F -> id
51	G -> F	104	G -> F
52	E -> G	105	E -> E + G
53	S -> id = E ;	106	S -> id = E ;
54	X -> int	107	N -> epsilon
55	C -> epsilon	108	BM -> epsilon
56	T -> X C	109	F -> id
57	A -> epsilon	110	G -> F
58	D -> T id A ;	111	E -> G
59	X -> int	112	F -> id
60	C -> epsilon	113	G -> F
61	T -> X C	114	E -> E + G
62	A -> epsilon	115	S -> id = E ;
63	D -> T id A ;	116	S -> if (B) BM S N else BM S

```
117      S -> while BM ( B ) BM S
118      F -> id
119      G -> F
120      E -> G
121      F -> id
122      G -> F
123      F -> id
124      G -> F
125      E -> G
126      F -> id
127      G -> F
128      E -> E + G
129      F -> ( E )
130      G -> G * F
131      E -> E + G
132      S -> id = E ;
133      F -> id
134      G -> F
135      E -> G
136      Relop -> >
137      F -> id
138      G -> F
139      E -> G
140      I -> E Relop E
141      H -> I
142      B -> H
143      BM -> epsilon
144      F -> id
145      G -> F
146      E -> G
147      S -> id = E ;
148      S -> if ( B ) BM S
149      X -> float
150      DM -> epsilon
151      X -> float
152      M -> X id
153      F -> id
154      G -> F
155      E -> G
156      F -> num
157      G -> F
158      E -> E + G
159      S -> id = E ;
160      F -> id
161      G -> F
162      E -> G
163      S -> return E ;
164      P -> epsilon
165      P -> S P
166      P -> S P
167      D -> proc X id DM ( M ) { P }
168      X -> int
169      C -> epsilon

170      C -> [ num ] C
171      C -> [ num ] C
172      T -> X C
173      A -> epsilon
174      D -> T id A ;
175      X -> int
176      C -> epsilon
177      T -> X C
178      A -> epsilon
179      D -> T id A ;
180      X -> int
181      C -> epsilon
182      T -> X C
183      A -> epsilon
184      D -> T id A ;
185      X -> int
186      C -> epsilon
187      T -> X C
188      A -> epsilon
189      D -> T id A ;
190      X -> int
191      C -> epsilon
192      T -> X C
193      A -> epsilon
194      D -> T id A ;
195      X -> float
196      C -> epsilon
197      T -> X C
198      A -> epsilon
199      D -> T id A ;
200      F -> id
201      G -> F
202      E -> G
203      F -> id
204      G -> F
205      E -> G
206      L -> id [ E ]
207      F -> id
208      G -> F
209      E -> G
210      L -> L [ E ]
211      F -> L
212      G -> F
213      E -> E + G
214      S -> id = E ;
215      F -> id
216      G -> F
217      E -> G
218      L -> id [ E ]
219      F -> id
220      G -> F
221      E -> G
222      L -> L [ E ]
```

223	F -> id	263	F -> id
224	G -> F	264	G -> F
225	E -> G	265	E -> G
226	S -> L = E ;	266	S -> return E ;
227	F -> id	267	P -> epsilon
228	G -> F	268	P -> S P
229	F -> id	269	P -> D P
230	G -> G * F	270	P -> S P
231	E -> G	271	D -> proc X id DM (M) { P }
232	S -> id = E ;	272	P -> epsilon
233	F -> num	273	P -> D P
234	G -> F	274	P -> S P
235	E -> G	275	P -> S P
236	L -> id [E]	276	P -> S P
237	F -> num	277	P -> S P
238	G -> F	278	P -> D P
239	E -> G	279	P -> D P
240	L -> L [E]	280	P -> D P
241	F -> id	281	P -> D P
242	G -> F	282	P -> D P
243	E -> G	283	P -> D P
244	S -> L = E ;	284	P -> D P
245	X -> int	285	P -> S P
246	DM -> epsilon	286	P -> S P
247	X -> int	287	P -> S P
248	M -> X id	288	P -> S P
249	X -> int	289	P -> D P
250	M -> M , X id	290	P -> S P
251	F -> id	291	P -> D P
252	G -> F	292	P -> D P
253	E -> G	293	P -> S P
254	F -> num	294	P -> D P
255	G -> F	295	P -> S P
256	E -> E + G	296	P -> D P
257	S -> id = E ;	297	P -> S P
258	X -> int	298	P -> S P
259	C -> epsilon	299	P -> D P
260	T -> X C	300	P -> D P
261	A -> epsilon	301	P -> S P
262	D -> T id A ;	302	P -> D P

其中 epsilon 表示空串，即 ϵ 。

经过 Parser 中的 reduce 函数，将第4.3节对应的 Token 序列转化为产生式，其最后一个产生式将其规约到原始文法中的 P 即开始符号上，证明完成了语法部分的规约。

4.5 中间代码

1	t1 = 1 + 2	4	t2 = c + 1
2	a = t1	5	b = t2
3	c = 10	6	t3 = c + c

7	d = t3	25	if a > b goto 26
8	e = 0	26	goto 27
9	y = 999	27	c = d
10	z = 100	28	t9 = i + 1
11	if a < b goto 12	29	i = t9
12	goto 20	30	t10 = i * 12
13	if c < d goto 14	31	t11 = j * 4
14	goto 17	32	t12 = t10 + t11
15	t4 = y + z	33	t13 = c + list[t12]
16	x = t4	34	d = t13
17	goto 10	35	t14 = i * 12
18	t5 = a + b	36	t15 = j * 4
19	x = t5	37	t16 = t14 + t15
20	goto 10	38	list [t16] = c
21	t6 = d + e	39	t17 = h * (float) c
22	t7 = c * t6	40	d = t17
23	t8 = b + t7	41	t18 = c + 10
24	a = t8	42	a = t18

在未进行代码优化时，产生的中间代码序列如上所示，其为 Parser 在 reduce 函数返回之前的输出。

4.6 附加：代码优化

1	a = 1 + 2	17	t7 = c * t6
2	c = 10	18	a = b + t7
3	b = c + 1	19	if a > b goto 26
4	d = c + c	20	goto 27
5	e = 0	21	c = d
6	y = 999	22	i = i + 1
7	z = 100	23	t10 = i * 12
8	if a < b goto 12	24	t11 = j * 4
9	goto 20	25	t12 = t10 + t11
10	if c < d goto 14	26	d = c + list[t12]
11	goto 17	27	t14 = i * 12
12	x = y + z	28	t15 = j * 4
13	goto 10	29	t16 = t14 + t15
14	x = a + b	30	list [t16] = c
15	goto 10	31	d = h * (float) c
16	t6 = d + e	32	a = c + 10

以上是经过代码优化之后得到的源文件的中间代码，可以看到其相比第4.5节的 42 行代码，缩减了 10 行。具体使用给定代码优化技术有局部无用中间代码删除和强度削弱。

具体的对比如图4.1所示。

- 可以看到在第 0 行使用到的中间变量 t_1 ，仅仅在第 0 行和第 1 行出现，且并未在之后使用，所以可以将其简单优化为 $a = 1 + 2$ 。
- 在未优化的左侧代码，使用了简单的 $t_3 = c * 2$ ，可以将乘法削弱为加法，如右侧 $d = c + c$ 。

```

19-4-28 上午11:36 - interCode.txt
0 : t1 = 1 + 2
1 : a = t1
2 : c = 10
3 : t2 = c + 1
4 : b = t2
5 : t3 = c * 2
6 : d = t3
7 : e = 0
8 : y = 999
9 : z = 100
10 : if a < b goto 12
11 : goto 20
12 : if c < d goto 14
13 : goto 17
14 : t4 = y + z
15 : x = t4
16 : goto 10
17 : t5 = a + b
18 : x = t5
19 : goto 10
20 : t6 = d + e
21 : t7 = c * t6
22 : t8 = b + t7
23 : a = t8
24 : if a > b goto 26
25 : goto 27
26 : c = d
27 : t9 = i + 1
28 : i = t9
29 : t10 = i * 12
30 : t11 = j * 4
31 : t12 = t10 + t11
32 : t13 = c + list[t12]
33 : d = t13
34 : t14 = i * 12
35 : t15 = j * 4
36 : t16 = t14 + t15
37 : list [ t16 ] = c
38 : t17 = h * (float) c
39 : d = t17
40 : t18 = c + 10
41 : a = t18

Current
1 1 0 : a = 1 + 2
2 2 1 : c = 10
3 3 2 : b = c + 1
4 4 3 : d = c + c
5 5 4 : e = 0
6 6 5 : y = 999
7 7 6 : z = 100
8 8 7 : if a < b goto 12
9 9 8 : goto 20
10 10 9 : if c < d goto 14
11 11 10 : goto 17
12 12 11 : x = y + z
13 13 12 : goto 10
14 14 13 : x = a + b
15 15 14 : goto 10
16 16 15 : t6 = d + e
17 17 16 : t7 = c * t6
18 18 17 : a = b + t7
19 19 18 : if a > b goto 26
20 20 19 : goto 27
21 21 20 : c = d
22 22 21 : i = i + 1
23 23 22 : t10 = i * 12
24 24 23 : t11 = j * 4
25 25 24 : t12 = t10 + t11
26 26 25 : d = c + list[t12]
27 27 26 : t14 = i * 12
28 28 27 : t15 = j * 4
29 29 28 : t16 = t14 + t15
30 30 29 : list [ t16 ] = c
31 31 30 : d = h * (float) c
32 32 31 : a = c + 10
33 33
34
35
36
37
38
39
40
41
42
43

```

图 4.1: 中间代码优化前后对比

A 源代码

- src/css/内包含 GUI 使用的 css 文件。
- src/lexer/内包含词法分析部分使用到的源码。
- src/parser/内包含了语法和语义分析部分使用到的源码。
- src/symbols/内包含了符号表部分的源码。
- Main.java 为 GUI 形式的 Compiler 主函数, sample.fxml 为 JavaFX 的配置文件。

B GUI 展示



The screenshot shows a window titled "Compiler" with a menu bar containing "Source", "Lexer", "Grammar", "LR Table", and "Parser". The "Source" tab is active, displaying the following code in a dark-themed editor:

```
int a = 10;
a = x + y;
/*int a = x + 1;*/
int [100][200] b;
a = c;
int d = 0;
if (a == c)
    d = 1;
else
    d = 2;

/**Hello world!
Hello Java! **/

do
    a = a + 1;
while (a < 20);

float f = 10.0;

char [10] s = "Hello";

proc int function(int x, int y){
    y = x + 1;
    return y;
}

call function(a, d);
record stu {
    int z = 1;
}
int xx
int yy;
```

图 B.1: 源码部分

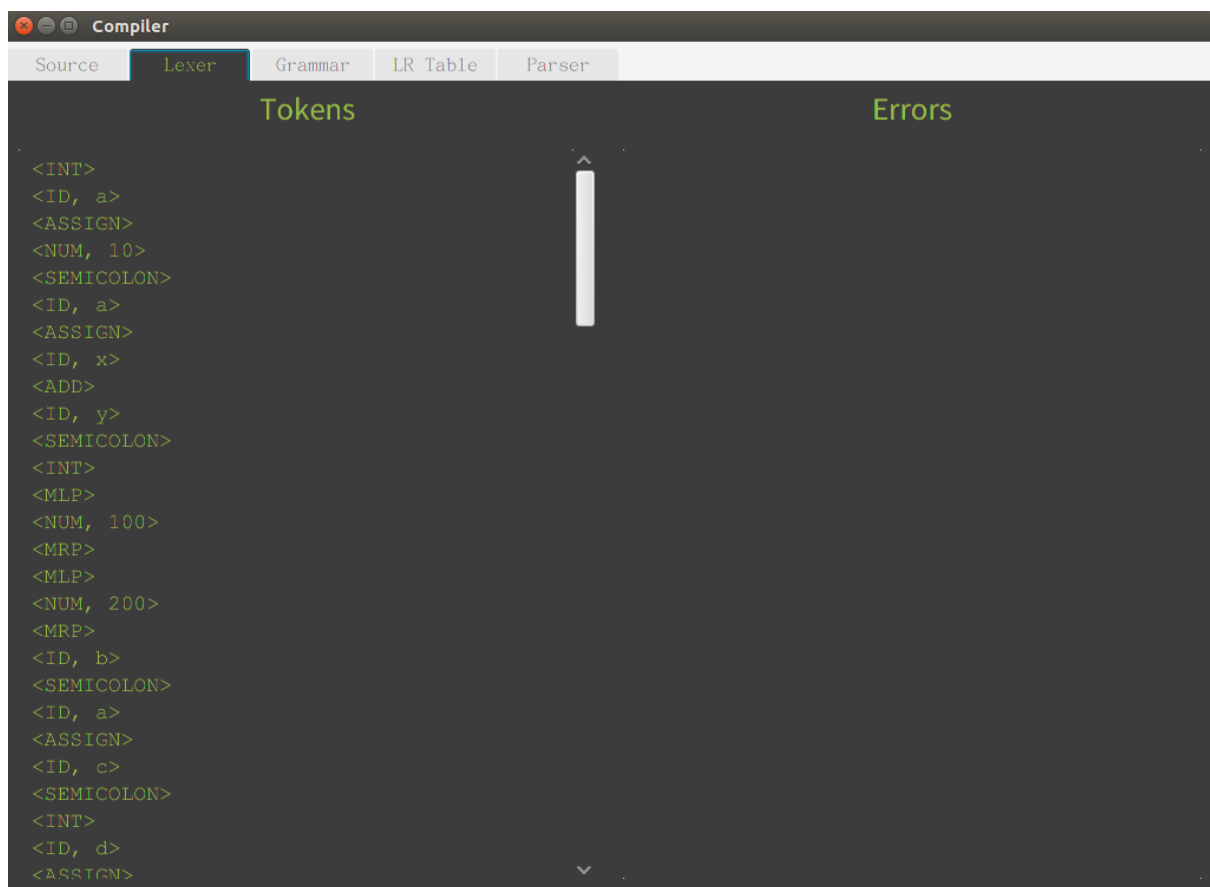
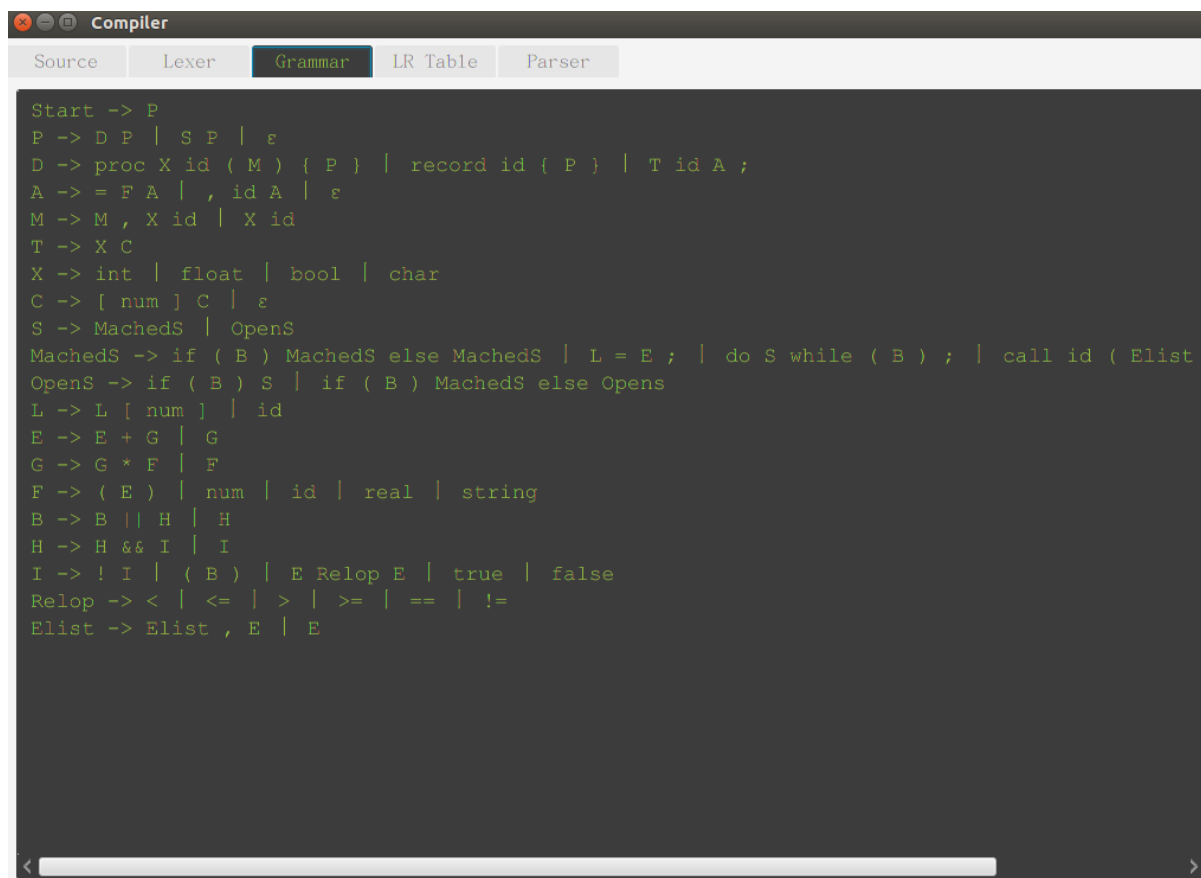


图 B.2: Lexer 部分

A screenshot of a software application titled "Compiler". It has four tabs: "Source", "Lexer", "Grammar" (which is selected and highlighted), "LR Table", and "Parser". The main area displays a list of grammar rules in a monospaced font. The rules are: Start -> P; P -> D P | S P | ε; D -> proc X id (M) { P } | record id { P } | T id A ;; A -> = F A | , id A | ε; M -> M , X id | X id; T -> X C; X -> int | float | bool | char; C -> [num] C | ε; S -> MachedS | OpenS; MachedS -> if (B) MachedS else MachedS | L = E ; | do S while (B) ; | call id (Elist; OpenS -> if (B) S | if (B) MachedS else Opens; L -> L [num] | id; E -> E + G | G; G -> G * F | F; F -> (E) | num | id | real | string; B -> B || H | H; H -> H && I | I; I -> ! I | (B) | E Relop E | true | false; Relop -> < | <= | > | >= | == | !=; Elist -> Elist , E | E.

```
Start -> P
P -> D P | S P | ε
D -> proc X id ( M ) { P } | record id { P } | T id A ;
A -> = F A | , id A | ε
M -> M , X id | X id
T -> X C
X -> int | float | bool | char
C -> [ num ] C | ε
S -> MachedS | OpenS
MachedS -> if ( B ) MachedS else MachedS | L = E ; | do S while ( B ) ; | call id ( Elist
OpenS -> if ( B ) S | if ( B ) MachedS else Opens
L -> L [ num ] | id
E -> E + G | G
G -> G * F | F
F -> ( E ) | num | id | real | string
B -> B || H | H
H -> H && I | I
I -> ! I | ( B ) | E Relop E | true | false
Relop -> < | <= | > | >= | == | !=
Elist -> Elist , E | E
```

图 B.3: 文法部分

		&&	<=	Opens	string	bool
0						s9
1						
2						
3						
4						
5						
6						r(S ->
7						r(S ->
8						s9
9						
10						
11						
12						s9
13						
14						
15						s40
16						
17						
18						
19					s25	
20					s70	
21						
22						
23					s56	
24						

图 B.4: LR 分析表部分

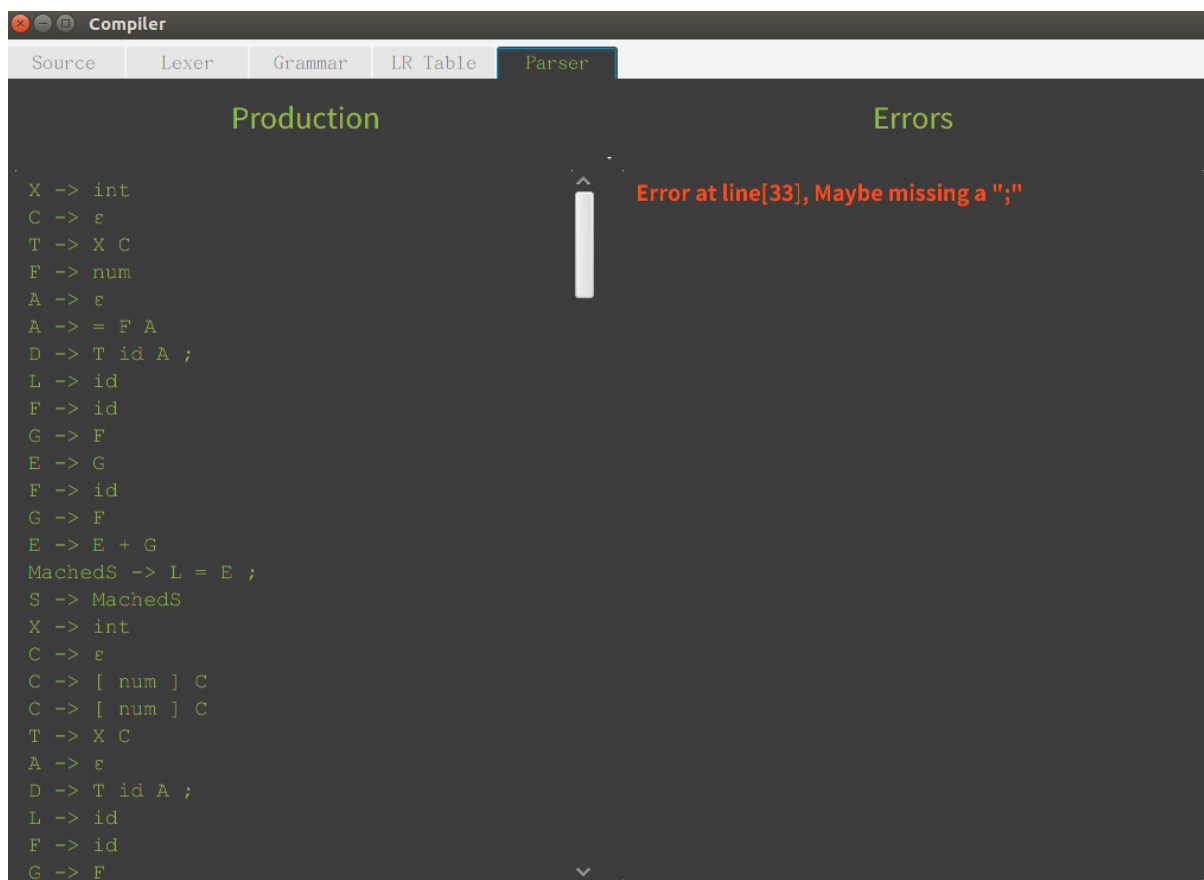


图 B.5: Parser 部分

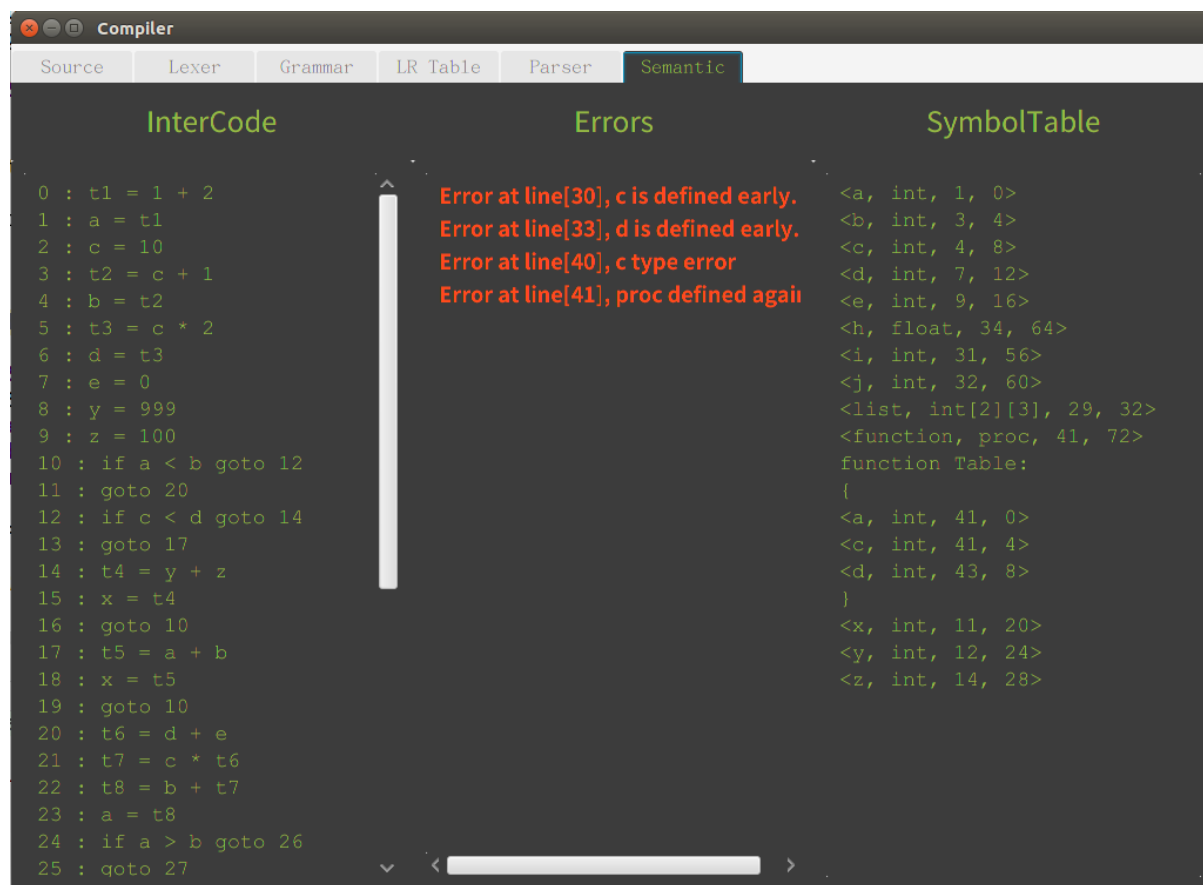


图 B.6: 语义分析部分

C 参考文献

参考文献

- [1] Aho A V, Sethi R, Ullman J D. Compilers, principles, techniques[J]. Addison wesley, 1986, 7(8): 9.