**2024\_compiler\_term\_project**

**소프트웨어학부**

**20201044 진용욱**

**0. Develop environment**

* OS: macOS Sonoma 14.4.1
* Compiler: Apple clang version 12.0.0 (clang-1200.0.32.28)

**1. Non-ambiguous CFG**

**1.1 CFG**

1 S -> CODE

2 CODE -> VDECL CODE

3 CODE -> FDECL CODE

4 CODE -> ''

5 VDECL -> vtype id semi

6 VDECL -> vtype ASSIGN semi

7 ASSIGN -> id assign RHS

8 RHS -> EXPR

9 RHS -> literal

10 RHS -> character

11 RHS -> boolstr

12 EXPR -> E

13 E -> E addsub T

14 E -> T

15 T -> T multdiv F

16 T -> F

17 F -> lparen E rparen

18 F -> id

19 F -> num

20 FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace

21 ARG -> vtype id MOREARGS

22 ARG -> ''

23 MOREARGS -> comma vtype id MOREARGS

24 MOREARGS -> ''

25 BLOCK -> STMT BLOCK

26 BLOCK -> ''

27 STMT -> VDECL

28 STMT -> ASSIGN semi

29 STMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE

30 STMT -> while lparen COND rparen lbrace BLOCK rbrace

31 COND -> boolstr \_COND

32 \_COND -> comp COND

33 \_COND -> ''

34 ELSE -> else lbrace BLOCK rbrace

35 ELSE -> ''

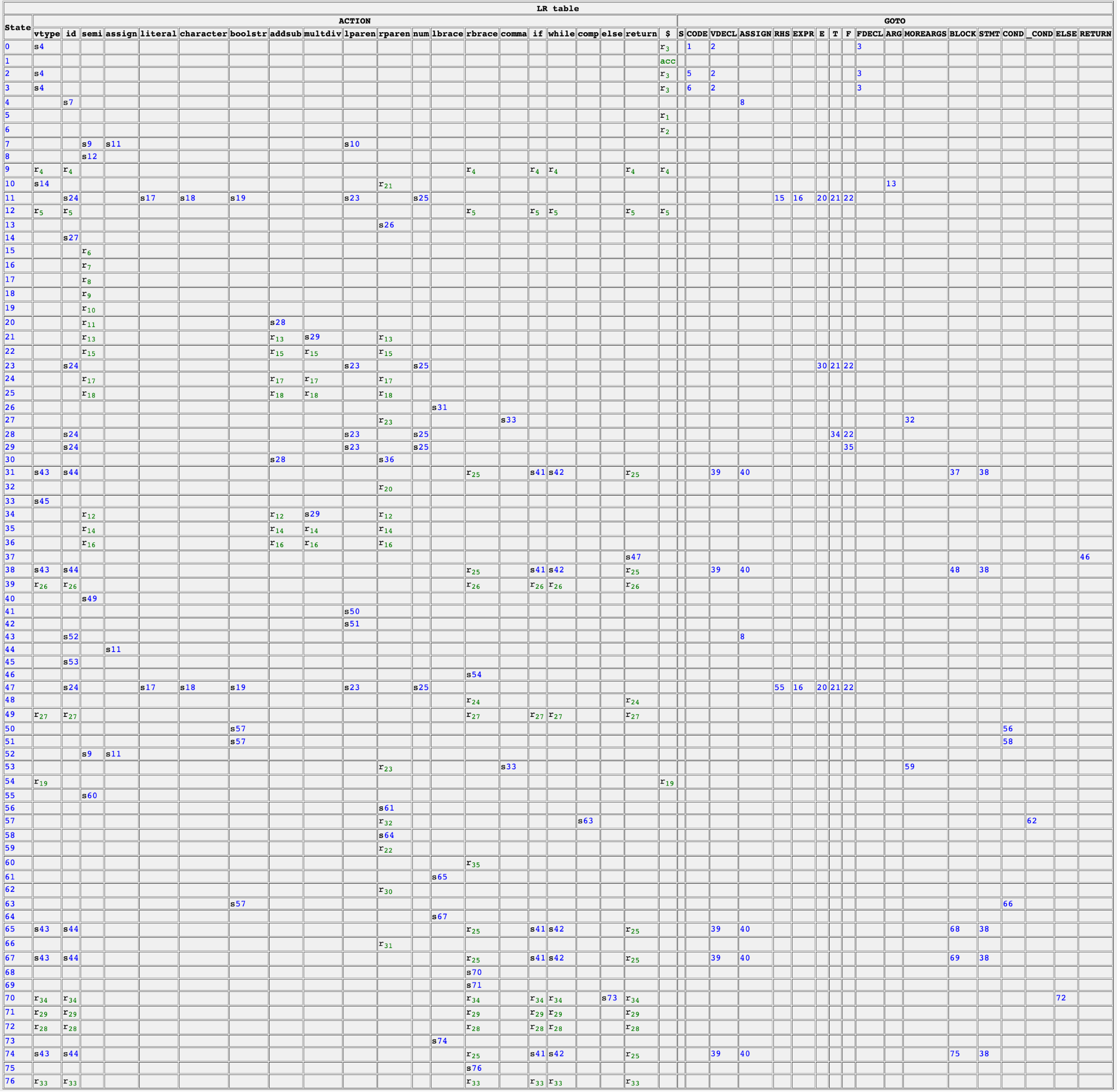
36 RETURN -> return RHS semi

**1.2 Changes**

1. add start symbol S (CFG line 1)
2. Each rule is listed one line at a time.
3. modify EXPR rules (Origin CFG line 05~06) to E, T, F rules to remove ambiguity and assign operation priorities. (CFG line 12~19)
4. modify COND rules (Origin CFG line 14) to COND and \_COND rules to remove ambiguity. (CFG line 31~33)

**2. SLR parsing table (can see bigger in**

* you can see detail (bigger) table in parsing\_table.png or in <http://jsmachines.sourceforge.net/machines/slr.html>

using CFG in SLR/resource/CFG.txt

**3. Syntax analyzer**

1. parsing input file, /resource/CFG.txt, /resource/table.txt -> init tokens, CFG rules, action table. The information in the file (CFG.txt, table.txt) is assumed to be valid. - src/init.ccp, src/parsefile.cpp
2. add init state to state\_stack.
3. to represent tree, using Node class. Node class include string token, Node\* parent, vector<Node\*> childs.
4. while accept or reject, do action in table[state][symbol], state = state\_stack.top(), symbol = tokens[splitter]
5. if action is shift (sn), shift splitter to right (increase) and add state n to state\_stack, add Node(tokens[splitter]) to parsing\_stack
6. if action is reduce (rn), pop state\_stack and parsing\_stack n times. and add new Node(cfg[n].LHS), add child to this Node which popped in parsing\_stack. and goto(state\_stack.top()) (add GOTO state using changed state\_stack)
7. if action is acc, accept and print parsing tree.
8. else, there is no rule in action table, print error using state\_stack, parsing\_stack, tokens. and print uncomplete parsing tree
9. if you want to see parsing detail step by step, please uncommenting include/SLR.h 4 line and make re. (#define DEBUG)

**4. Example**

1. **accept example**

**test2\_in.txt**

vtype id assign id semi

**test2\_out.txt**

Accept!!!

└── S

└── CODE

├── VDECL

│ ├── vtype

│ ├── ASSIGN

│ │ ├── id

│ │ ├── assign

│ │ └── RHS

│ │ └── EXPR

│ │ └── E

│ │ └── T

│ │ └── F

│ │ └── id

│ └── semi

└── CODE

1. **reject example**

**test5\_in.txt (no lbrace in FDECL before if)**

vtype id semi vtype id lparen rparen if lparen boolstr comp boolstr rparen lbrace vtype id semi rbrace return id semi rbrace

**test5\_out.txt**

Error occured at step 11, No action in table[26][if]

State Stack: 0 2 4 7 10 13 26

Token Stack: VDECL vtype id lparen ARG rparen if | lparen boolstr comp boolstr rparen lbrace vtype id semi rbrace return id semi rbrace $

└── S

├── VDECL

│ ├── vtype

│ ├── id

│ └── semi

├── vtype

├── id

├── lparen

├── ARG

└── rparen

1. **Input File Error example**

**test6\_in.txt (6th token is lpare, invalid)**

vtype id semi vtype id lpare rparen lbrace if lparen boolstr comp boolstr rparen lbrace rbrace

**test6\_out.txt**

Input file has invalid token: lpare!!

1. **debug example (uncommenting SLR.h //#define DEBUG)**

**test2\_in.txt**

vtype id assign id semi

**test2\_debug\_out.txt**

===== Step0 action:s4 =====

State Stack: 0

Token Stack: | vtype id assign id semi $

S4

===== Step1 action:s7 =====

State Stack: 0 4

Token Stack: vtype | id assign id semi $

S7

===== Step2 action:s11 =====

State Stack: 0 4 7

Token Stack: vtype id | assign id semi $

S11

===== Step3 action:s24 =====

State Stack: 0 4 7 11

Token Stack: vtype id assign | id semi $

S24

===== Step4 action:r17 =====

State Stack: 0 4 7 11 24

Token Stack: vtype id assign id | semi $

R17

GOTO22

===== Step6 action:r15 =====

State Stack: 0 4 7 11 22

Token Stack: vtype id assign F | semi $

R15

GOTO21

===== Step8 action:r13 =====

State Stack: 0 4 7 11 21

Token Stack: vtype id assign T | semi $

R13

GOTO20

===== Step10 action:r11 =====

State Stack: 0 4 7 11 20

Token Stack: vtype id assign E | semi $

R11

GOTO16

===== Step12 action:r7 =====

State Stack: 0 4 7 11 16

Token Stack: vtype id assign EXPR | semi $

R7

GOTO15

===== Step14 action:r6 =====

State Stack: 0 4 7 11 15

Token Stack: vtype id assign RHS | semi $

R6

GOTO8

===== Step16 action:s12 =====

State Stack: 0 4 8

Token Stack: vtype ASSIGN | semi $

S12

===== Step17 action:r5 =====

State Stack: 0 4 8 12

Token Stack: vtype ASSIGN semi | $

R5

GOTO2

===== Step19 action:r3 =====

State Stack: 0 2

Token Stack: VDECL | $

R3

GOTO5

===== Step21 action:r1 =====

State Stack: 0 2 5

Token Stack: VDECL CODE | $

R1

GOTO1

===== Step23 action:acc =====

State Stack: 0 1

Token Stack: CODE | $

Accept!!!

└── S

└── CODE

├── VDECL

│ ├── vtype

│ ├── ASSIGN

│ │ ├── id

│ │ ├── assign

│ │ └── RHS

│ │ └── EXPR

│ │ └── E

│ │ └── T

│ │ └── F

│ │ └── id

│ └── semi

└── CODE