<non-ambiguous CFG>

START -> CODE
CODE -> VDECL CODE | FDECL CODE | ϵ
VDECL -> vtype id semi | vtype ASSIGN semi
ASSIGN -> id assign RHS
RHS -> EXPR | literal | character | boolstr
**EXPR -> TERM addsub EXPR | TERM
**TERM -> FACTOR multdiv TERM | FACTOR
FACTOR -> lparen EXPR rparen | id | num
FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
ARG -> vtype id MOREARGS | ϵ
MOREARGS -> comma vtype id MOREARGS | ϵ
BLOCK -> STMT BLOCK | ϵ
STMT -> VDECL | ASSIGN semi | if lparen COND rparen lbrace BLOCK rbrace ELSE
STMT -> while lparen COND rparen lbrace BLOCK rbrace
**COND -> COND comp boolstr | boolstr
ELSE -> else lbrace BLOCK rbrace | ϵ
RETURN -> return RHS semi

주어진 CFG G에서 ambiguous가 발생할 수 있는 부분은 두 가지로 나타났다.
1. EXPR → EXPR addsub EXPR | EXPR multdiv EXPR
이 부분에서 모호성이 발생하는 이유는 연산자의 우선순위와 결합법칙이 명확하게 정의되지 않기 때문이다. 이로 인해 동일한 문자열을 파싱할 때 여러 가지 방법으로 구문 분석 트리를 만들 수 있다.

따라서 다음 구문을 새로운 non terminal FACTOR을 추가해서 수정하였다
**EXPR -> TERM addsub EXPR | TERM
**TERM -> FACTOR multdiv TERM | FACTOR

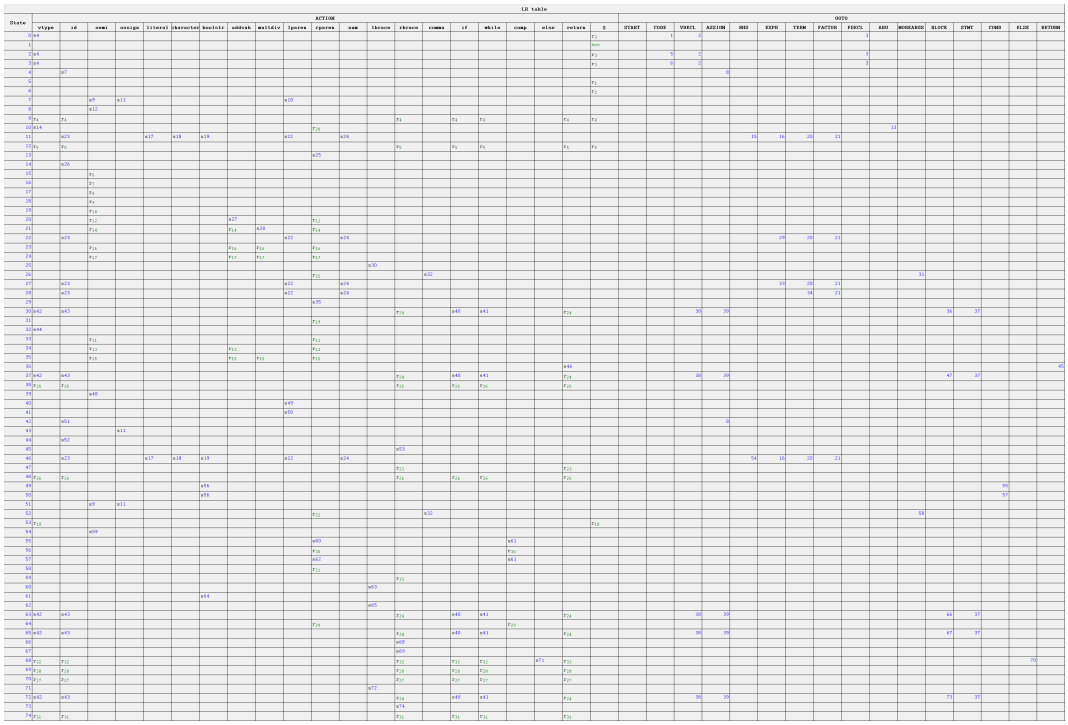
2. COND -> COND comp boolstr
논리 연산자의 우선순위와 결합법칙이 명확하게 정의되지 않기 때문이다. 이로 인해 동일한 문자열을 파싱할 때 여러 가지 방법으로 구문 분석 트리를 만들 수 있다.

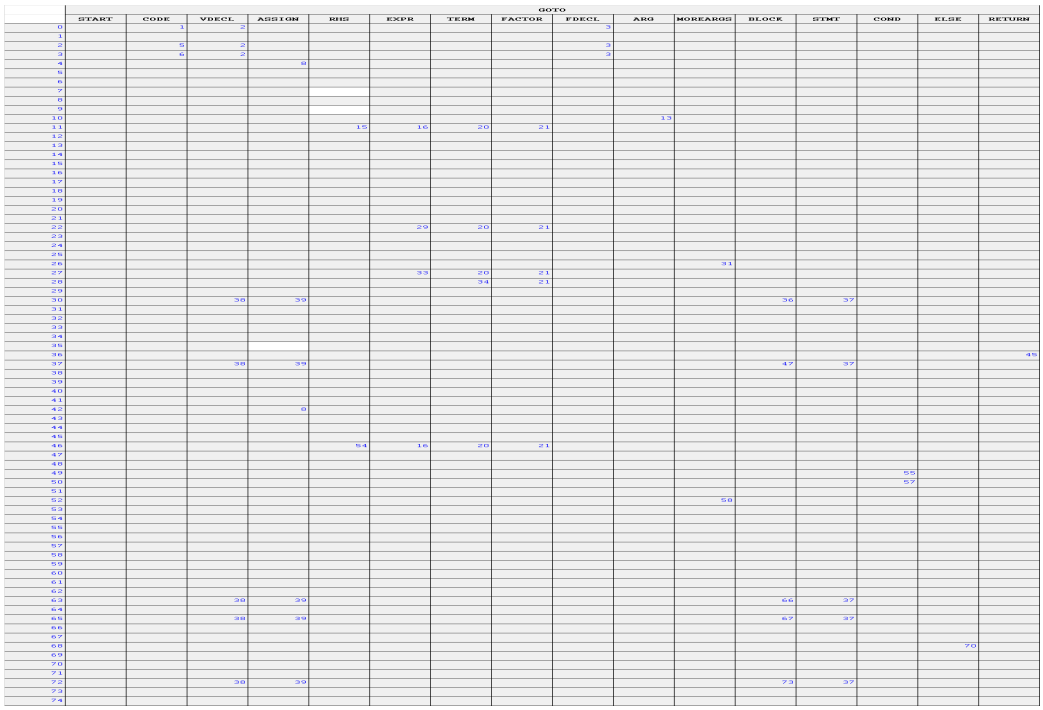
따라서 다음 구문을 c언어 문법에 따라 다음과 같이 수정하였다.
**COND -> COND comp boolstr | boolstr

수정된 CFG G를 바탕으로 사이트에서 SLR parsing table을 구성하였다.

<SLR parsing table>

(제출한 excel 파일에 원본이 있습니다)





단 사이트를 이용할 때, 처음 시작 symbol(non terminal)이 여러 개의 RHS를 가지면 올바르지 않은 parsing table이 형성되기에 임의로 처음 시작 symbol로 START를 추가하여 다음 생성규칙을 추가하였다.

START -> CODE

<TreeNode 구조체>

```
//TreeNode 구조체 선언 : 구조체이름(symbol)과 자식노드(children)들로 정의
typedef struct TreeNode {
    char symbol[MAX_SYMBOL_LENGTH];
    struct TreeNode* children[MAX_CHILDREN];
    int childCount;
} TreeNode;
```

이 구조체에서 각 노드는 symbol과 그 자식 노드들을 가리키는 포인터 배열(children)을 가지며, 자식 노드의 개수를 나타내는 childCount를 가지고 있다.

<createNode 함수>

```
//createNode 함수 선언 : 구조체이름(symbol)을 인자로 넘겨받아 new Node를 생성하는 함수
TreeNode* createNode(const char* symbol) {
    TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
    strncpy(newNode->symbol, symbol, MAX_SYMBOL_LENGTH);
    for (int i = 0; i < MAX_CHILDREN; i++) {
        newNode->children[i] = NULL;
    }
    newNode->childCount = 0;
    return newNode;
}
```

이 함수에서는 주어진 symbol을 이용하여 새로운 노드를 생성하고, 심볼은 인자로 받아 new node를 동적으로 할당하고 초기화한 후 반환한다.

TreeNode* createNode(const char* symbol) 함수는 const char* 형식의 symbol을 인자로 받아 TreeNode 구조체의 포인터를 반환한다.

TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));을 이용해 TreeNode 구조체 크기만큼 메모리를 동적으로 할당하여 newNode 포인터에 저장한다.

strncpy(newNode->symbol, symbol, MAX_SYMBOL_LENGTH);을 통해 symbol 문자열을 newNode의 symbol 멤버에 복사한다. 여기서 strncpy는 최대 MAX_SYMBOL_LENGTH 만큼 복사 가능하다.

for 문에서는 새 노드의 모든 자식 포인터를 NULL로 초기화하고 MAX_CHILDREN은 자식 노드의 최대 개수를 정의한다.

newNode->childCount = 0;를 통해 자식 노드의 개수를 나타내는 childCount를 0으로 초기화하고 return을 통해 새롭게 생성된 노드의 포인터를 반환한다.

<printParseTreePretty 함수>

```
//printParseTree 함수 선언 : parsetree를 시각화하여 출력하는 함수
void printParseTreePretty(TreeNode* root, int depth, int isLast) {
    if (root == NULL) return;
    for (int i = 0; i < depth; i++) {
        printf("    ");
    }
    if (depth > 0) {
        if (isLast) {
            printf(" └──");
        }
        else {
            printf(" ├──");
        }
    }
    printf(" %s\n", root->symbol);
    for (int i = 0; i < root->childCount; i++) {
        printParseTreePretty(root->children[i], depth + 1, i == root->childCount - 1);
    }
}
```

이 함수에서는 파싱 트리를 시각화하여 출력한다. 트리의 root node와 해당 노드의 깊이(depth)를 확인한다. 현재 노드가 해당 깊이에서 마지막 노드인지 여부를 확인하고 깊이와 마지막 여부에 따라 트리를 시각화하여 출력한다.

<Action 구조체>

```
//Action 구조체 선언 : 파싱테이블의 action(s->shift, r->reduce, a->accept)과 state 상태정보를 가짐
typedef struct {
    char action;
    int state;
} Action;
```

Action 구조체를 통해 구문 분석에서 각 state와 input에 대해 수행할 작업과 그 작업에 따라 전환될 상태를 정의한다. 이 구조체에서 각 엔트리는 액션(action)과 상태(state) 정보를 가지고 있다. 액션은 shift, reduce, accept 중 하나를 나타내고 상태는 파싱 과정에서의 상태를 나타낸다.

<Action table 및 goto Table>

Action table에서는 state와 input의 조합에 대해 수행할 작업과 그 작업에 따른 다음 상태를 정의하였다. 이 테이블에서는 parsing 중에 취해야 할 action을 결정한다.
s 는 입력 기호를 스택에 push 하고 지정된 state로 전환한다.
r 은 특정 규칙에 따라 스택에서 기호를 pop하고 non-terminal symbol을 스택에 push 한다.

Goto table에서는 non-terminal symbol을 스택에 push하고 현재 상태와 non-terminal symbol을 바탕으로 상태를 정의한다.
-1은 해당 상태에서 non-terminal symbol이 허용되지 않음을 뜻한다.

<getSymbolIndex 함수>

```
//getSymbolIndex 함수 선언 : parsing table의 인덱스를 구하기 위해 모든 nonterminal을 정수값에 대응
int getSymbolIndex(const char* symbol) {
    if (strcmp(symbol, "vtype") == 0)       return 0;
    if (strcmp(symbol, "id") == 0)          return 1;
    if (strcmp(symbol, "semi") == 0)        return 2;
    if (strcmp(symbol, "assign") == 0)      return 3;
    if (strcmp(symbol, "literal") == 0)     return 4;
    if (strcmp(symbol, "character") == 0)   return 5;
    if (strcmp(symbol, "boolstr") == 0)     return 6;
    if (strcmp(symbol, "addsub") == 0)      return 7;
    if (strcmp(symbol, "multdiv") == 0)     return 8;
    if (strcmp(symbol, "lparen") == 0)      return 9;
    if (strcmp(symbol, "rparen") == 0)      return 10;
    if (strcmp(symbol, "num") == 0)         return 11;
    if (strcmp(symbol, "lbrace") == 0)      return 12;
    if (strcmp(symbol, "rbrace") == 0)      return 13;
    if (strcmp(symbol, "comma") == 0)       return 14;
    if (strcmp(symbol, "if") == 0)          return 15;
    if (strcmp(symbol, "while") == 0)       return 16;
    if (strcmp(symbol, "comp") == 0)        return 17;
    if (strcmp(symbol, "else") == 0)        return 18;
    if (strcmp(symbol, "return") == 0)      return 19;
    if (strcmp(symbol, "$") == 0)           return 20;
    return -1;
}
```

이 함수에서는 parsing table의 인덱스를 구하기 위해 모든 nonterminal을 정수값에 대응시킨다. 이후 이 인덱스를 parser의 action table에서 사용한다. 만약 기호가 정의되지 않으면 -1을 반환한다.

<getSymbolIndex 함수>

```
//getSymbolIndex 함수 선언 : parsing table의 인덱스를 구하기 위해 모든 terminal을 정수값에 대응
int getGotoIndex(const char* nonTerminal) {
    if (strcmp(nonTerminal, "START") == 0)      return 0;
    if (strcmp(nonTerminal, "CODE") == 0)       return 1;
    if (strcmp(nonTerminal, "VDECL") == 0)      return 2;
    if (strcmp(nonTerminal, "ASSIGN") == 0)     return 3;
    if (strcmp(nonTerminal, "RHS") == 0)        return 4;
    if (strcmp(nonTerminal, "EXPR") == 0)       return 5;
    if (strcmp(nonTerminal, "TERM") == 0)       return 6;
    if (strcmp(nonTerminal, "FACTOR") == 0)     return 7;
    if (strcmp(nonTerminal, "FDECL") == 0)      return 8;
    if (strcmp(nonTerminal, "ARG") == 0)        return 9;
    if (strcmp(nonTerminal, "MOREARGS") == 0)   return 10;
    if (strcmp(nonTerminal, "BLOCK") == 0)      return 11;
    if (strcmp(nonTerminal, "STMT") == 0)       return 12;
    if (strcmp(nonTerminal, "COND") == 0)       return 13;
    if (strcmp(nonTerminal, "ELSE") == 0)       return 14;
    if (strcmp(nonTerminal, "RETURN") == 0)     return 15;
    return -1;
}
```

이 함수에서는 parsing table의 인덱스를 구하기 위해 모든 terminal을 정수값에 대응시킨다. CFG g를 넣고 parsing table을 생성하는 경우, 시작 symbol이 CODE -> FDECL CODE , VEDCL CODE , e 세개로 분기되는 경우 잘못된 parsing table이 생성되기 때문에 임의로 START symbol을 추가하였다.
strcmp 함수를 사용해 input 문자열과 non-terminal symbol을 비교한다.
parsing table의 GOTO 부분에서 non terminal symbol에 대한 인덱스를 이용한다.

문자열이 일치하면 해당 non-terminal symbol의 index를 반환한다.
예를 들어 "CODE" 문자열이 입력으로 주어지면 인덱스 0을 반환하고 주어진 문자열이 미리 정의된 non-terminal symbol 목록에 없는 경우에는 -1을 반환한다.

<StackElement 구조체>

```
// StackElement 구조체 선언 : 스택에는 state와 symbol 값이 저장
typedef struct {
    int state;
    char symbol[MAX_SYMBOL_LENGTH];
} StackElement;

//스택의 크기를 결정
StackElement stack[MAX_STACK_SIZE];
int top = -1;
```

StackElement 구조체는 파서 스택의 각 요소를 정의하고 각 요소는 state와 symbol을 포함한다. state는 현재 스택 요소의 상태를 나타내고 symbol은 현재 스택 요소의 기호를 나타며 최대 길이가 MAX_SYMBOL_LENGTH로 제한된다.

stack은 StackElement 구조체 배열로, 파서의 상태와 기호를 저장하고 top은 스택의 최상위 요소를 가리키며 -1의 초기값을 갖는다.

<push 함수>

```
//push 함수 선언 : stack[top]에 값을 넣기 위한 함수
void push(int state, const char* symbol) {
    if (top < MAX_STACK_SIZE - 1) {
        top++;
        stack[top].state = state;
        strncpy(stack[top].symbol, symbol, MAX_SYMBOL_LENGTH);
    }
    else {
        printf("Stack overflow₩n");
        exit(1);
    }
}
```

push 함수는 스택에 새로운 state와 symbol을 추가한다.
스택의 top이 최대 크기 (MAX_STACK_SIZE - 1)보다 작으면 top을 1 증가시키고, 새 state와 symbol을 스택에 추가한다.
strncpy 함수를 사용해 주어진 기호를 스택 symbol에 복사하고. MAX_SYMBOL_LENGTH 만큼의 길이만큼 복사하여 오버플로우를 방지한다.
스택이 가득 찬 경우, "Stack overflow" 메시지를 출력하고 프로그램을 종료한다.

<pop함수>

```
//pop 함수 선언 : stack[top] 값을 가져오기 위한 함수
void pop(int count) {
    if (top >= count - 1) {
        top -= count;
    }
    else {
        printf("Stack underflow₩n");
        exit(1);
    }
}
```

pop 함수는 스택에서 지정된 수의 요소를 제거한다.
pop의 count는 스택에서 제거할 요소의 수를 나타내고 top이 count - 1보다 크거나 같은지 확인한다. 만약 top이 count - 1보다 크거나 같으면, 스택에 충분한 element가 있다고 판단하고 count만큼 pop 한다. 또한 top에서 count를 제거하여 최상위 요소를 갱신한다.
top이 count - 1보다 작으면, 스택 언더플로우가 발생하므로 오류 메시지를 출력하고 프로그램을 종료한다.

<main 함수>

main 함수에서는 주어진 문자열을 구문 분석하여 유효한지 확인한다. 입력 문자열은 여러 기호로 분할되고, 구문 분석을 통해 적절한 상태로 전환되거나 기호가 reduce된다.

```
int main() {
    char input[MAX_INPUT_SIZE];
    int i = 0;
    int state;
    char* tokens[100];
    int token_count = 0;
    // 각 토큰의 시작 위치를 저장하는 배열 추가
    int tokenPositions[100];

    FILE* file;
    char buffer[256];
```

char input[MAX_INPUT_SIZE]를 통해 입력 문자열을 저장하고 int state를 통해 현재 상태를 저장한다. char* tokens[100]을 통해 입력 문자열을 토큰으로 분할하여 저장한다.

```
// 파일 열기 (읽기 모드)
file = fopen("input.txt", "r");
if (file == NULL) {
    perror("파일을 열 수 없습니다.");
    return 1;
}
```

file = fopen("input.txt", "r");을 이용해 input.txt 파일을 읽기 모드로 열고 파일이 열리지 않으면 경우 오류 메시지를 출력하고 프로그램을 종료한다.
fgets 함수를 사용하여 파일에서 한 줄씩 문자열을 읽어오고 파일을 모두 읽으면 파일을 닫는다.

```
// 파일에서 문자열 읽기
while (fgets(input, sizeof(input), file) != NULL) {
}

fclose(file);

// 개행 문자 제거
input[strcspn(input, "₩n")] = '₩0';
```

while (fgets(input, sizeof(input), file) != NULL) 에서 fgets 함수를 이용해 파일에서 한줄씩 문자열을 읽어온다.
strcspn을 사용해 문자열에서 개행 문자의 위치를 찾고, 그 위치에 NULL 문자를 추가해 개행 문자를 제거한다.

```
// 문자열을 공백으로 분할
// 입력의 끝을 표시하는 심볼 추가
strcat(input, " $");
char* token = strtok(input, " ");
while (token != NULL && token_count < 100) {
    tokens[token_count] = token;
    tokenPositions[token_count] = token - input;  // 토큰의 시작 위치를 저장
    token_count++;
    token = strtok(NULL, " ");
}
```

strtok 함수를 사용하여 공백을 기준으로 문자열을 토큰화한다. 토큰을 차례대로 배열에 저장하고 동시에 해당 토큰의 시작 위치를 저장한다. 최대 100개의 토큰만 저장할 수 있다.
strcat(input, " $");은 input의 끝에 공백과 $ 기호를 추가하여 파서가 입력의 끝을 인식하도록 돕는다.
char* token = strtok(input, " ");에서는 strtok 함수를 사용하여 입력 문자열을 공백을 기준으로 첫 번째 토큰을 추출한다. strtok는 문자열을 분리하고, 첫 번째 호출에서는 분리할 문자열(input)과 구분자(" ")를 인자로 받는다.
while 문에서는 token이 NULL이 아니고, 토큰의 개수가 100개 미만인 동안 루프를 실행한다. 이는 입력 문자열의 모든 토큰을 처리하고, 최대 100개의 토큰을 저장하기 위한 조건이다.
tokens[token_count] = token;을 이용해 현재 추출된 토큰을 tokens 배열의 token_count 인덱스에 저장한다. tokens 배열은 각 토큰을 문자열로 저장하는 배열이다.
tokenPositions[token_count] = token - input; 을 통해 현재 토큰의 시작 위치를 tokenPositions 배열의 token_count 인덱스에 저장한다.
token = strtok(NULL, " ");을 이용해 다음 토큰을 추출한다. strtok 함수는 이후 호출에서 첫 번째 인자를 NULL로 하여 이전 호출에서 사용된 문자열의 나머지 부분을 계속 처리한다.

```
//가장 먼저 stack에 state = 0과 $를 push
push(0, "$");

TreeNode* parseTree = NULL;
TreeNode* nodeStack[MAX_STACK_SIZE];
int nodeTop = -1;

i = 0;
```

push(0, "$")를 통해 스택에 초기 상태 0과 종료 심볼 $를 푸시한다. 이후 현재 토큰의 인덱스를 추적하기 위한 변수 i를 초기화하고, 노드 스택과 노드의 최상위 위치를 추적하는 변수를 초기화한다.

```
while (i < token_count) { //토큰화한 것을 차례대로 검사
    state = stack[top].state;
    const char* symbol = tokens[i];
    int symbolIndex = getSymbolIndex(symbol);

    if (symbolIndex == -1) {  //잘못된 terminal 값이 input됐을 경우 invaalid symbol 출력
        printf("Invalid symbol '%s'\n", symbol);
        exit(1);
    }
    //action 테이블 값을 가져온다
    Action action = actionTable[state][symbolIndex];
```

while (i < token_count) 에서는 토큰화된 입력을 차례대로 검사한다.

state = stack[top].state;

const char* symbol = tokens[i];

int symbolIndex = getSymbolIndex(symbol);

에서 현재 스택 상태를 확인하고 입력에서 다음 토큰을 가져와 해당 토큰의 인덱스를 확인한다. 잘못된 terminal이 input된 경우(-1)에는 오류 메시지를 출력하고 프로그램을 종료한다. 만약 action table에서 잘못된 값을 참조했다면 error를 발생시킨다. 또한 parsing table에 존재하지 않은 terminal이 iput 되는 경우나 존재하지 않은 terminal이 input되는 경우에도 Error를 발생시킨다.

```
    //action이 s(shift)라면?
    if (action.action == 's') {
        push(action.state, symbol); //state와 symbol값을 stack에 저장
        nodeStack[++nodeTop] = createNode(symbol); //Node 생성
        i++;
    }
```

if (action.action == 's') 부분에서 action이 shift면 state와 symbol값을 stack에 저장하고 Node를 생성한다.

push를 통해 action.state와 현재 토큰을 스택에 푸시한다. 이는 파서의 상태와 입력 심볼을 저장한다.nodeStack[++nodeTop] = createNode(symbol);를 이용해 새로운 트리 노드를 생성하고, nodeStack에 푸시한다. createNode 함수는 주어진 심볼을 가진 새로운 노드를 생성한다.

nodeTop은 노드 스택의 최상위 인덱스를 가리키며, 이를 증가시켜 새 노드를 스택에 추가한다.

if문을 통해 symbolIndex가 -1인 경우, 현재 토큰이 유효하지 않은 터미널 값을 가짐을 의미하기 때문에 오류 메시지를 출력하고 프로그램을 종료한다.

```
//action이 r(reduce)라면?
else if (action.action == 'r') {
    TreeNode* newNode = NULL;
    int numChildren;
    //reduce 할 때 필요한 production rule
    switch (action.state) {
    case 0: // START -> CODE
        newNode = createNode("START");
        numChildren = 1;
        break;
    case 1: // CODE -> VDECL CODE
        newNode = createNode("CODE");
        numChildren = 2;
```

else if (action.action == 'r') 부분에서는 action이 reduce면 Reduce할 때 필요한 case에 따라 새로운 노드를 생성한다. TreeNode* newNode = NULL; int numChildren;을 통해 새로 생성될 트리노드를 위한 포인터와 자식노드의 수를 저장할 변수를 선언한다. switch (action.state) { 를 이용해 reduce 때 필요한 production rule을 정의한다. 잘못된 생성규칙이면 invalid reduction 을 출력하고 종료한다.

```
case 0: // START -> CODE
    newNode = createNode("START");
    numChildren = 1;
    break;
case 1: // CODE -> VDECL CODE
    newNode = createNode("CODE");
    numChildren = 2;
    break;
case 2: // CODE -> FDECL CODE
    newNode = createNode("CODE");
    numChildren = 2;
    break;
case 3: // CODE -> e
    newNode = createNode("CODE");
    numChildren = 0;
    break;
case 4: // VDECL -> vtype id semi
    newNode = createNode("VDECL");
    numChildren = 3;
    break;
case 5: // VDECL -> vtype ASSIGN semi
    newNode = createNode("VDECL");
    numChildren = 3;
    break;
case 6: // ASSIGN -> id assign RHS
    newNode = createNode("ASSIGN");
    numChildren = 3;
    break;
```

```
case 7: // RHS -> EXPR
    newNode = createNode("RHS");
    numChildren = 1;
    break;
case 8: // RHS -> literal
    newNode = createNode("RHS");
    numChildren = 1;
    break;
case 9: // RHS -> character
    newNode = createNode("RHS");
    numChildren = 1;
    break;
case 10: // RHS -> boolstr
    newNode = createNode("RHS");
    numChildren = 1;
    break;
case 11: // EXPR -> TERM addsub EXPR
    newNode = createNode("EXPR");
    numChildren = 3;
    break;
case 12: // EXPR -> TERM
    newNode = createNode("EXPR");
    numChildren = 1;
    break;
case 13: // TERM -> FACTOR multdiv TERM
    newNode = createNode("TERM");
    numChildren = 3;
    break;
```

```
case 14: // TERM -> FACTOR
    newNode = createNode("TERM");
    numChildren = 1;
    break;
case 15: // FACTOR -> lparen EXPR rparen
    newNode = createNode("FACTOR");
    numChildren = 3;
    break;
case 16: // FACTOR -> id
    newNode = createNode("FACTOR");
    numChildren = 1;
    break;
case 17: // FACTOR -> num
    newNode = createNode("FACTOR");
    numChildren = 1;
    break;
case 18: // FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
    newNode = createNode("FDECL");
    numChildren = 9;
    break;
case 19: // ARG -> vtype id MOREARGS
    newNode = createNode("ARG");
    numChildren = 3;
    break;
case 20: // ARG -> e
    newNode = createNode("ARG");
    numChildren = 0;
    break;
case 21: // MOREARGS -> comma vtype id MOREARGS
    newNode = createNode("MOREARGS");
    numChildren = 4;
    break;
```

```
case 22: // MOREARGS -> e
    newNode = createNode("MOREARGS");
    numChildren = 0;
    break;
case 23: // BLOCK -> STMT BLOCK
    newNode = createNode("BLOCK");
    numChildren = 2;
    break;
case 24: // BLOCK -> e
    newNode = createNode("BLOCK");
    numChildren = 0;
    break;
case 25: // STMT -> VDECL
    newNode = createNode("STMT");
    numChildren = 1;
    break;
case 26: // STMT -> ASSIGN semi
    newNode = createNode("STMT");
    numChildren = 2;
    break;
case 27: // STMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE
    newNode = createNode("STMT");
    numChildren = 8;
    break;
case 28: // STMT -> while lparen COND rparen lbrace BLOCK rbrace
    newNode = createNode("STMT");
    numChildren = 7;
    break;
```

```
case 29: // COND -> COND comp boolstr
    newNode = createNode("COND");
    numChildren = 3;
    break;
case 30: // COND -> boolstr
    newNode = createNode("COND");
    numChildren = 1;
    break;
case 31: // ELSE -> else lbrace BLOCK rbrace
    newNode = createNode("ELSE");
    numChildren = 4;
    break;
case 32: // ELSE -> e
    newNode = createNode("ELSE");
    numChildren = 0;
    break;
case 33: // RETURN -> return RHS semi
    newNode = createNode("RETURN");
    numChildren = 3;
    break;
default: //잘못된 생성규칙
    printf("Invalid reduction₩n");
    exit(1);
}
```

각 상태마다 자식 노드의 수를 설정하고 parse tree를 구성하기 위한 “(non terminal)”을 이름으로 갖는 노드를 생성하도록 하였다.

```
    //production rule의 right side의 nonterminal과 terminal의 개수의 합만큼 자식 노드생성
    for (int j = numChildren - 1; j >= 0; j--) {
        newNode->children[newNode->childCount++] = nodeStack[nodeTop--];
    }
    //production rule의 right side의 nonterminal과 terminal의 개수의 합만큼 pop진행
    pop(numChildren);
    //pop한 결과가 gotoTable의 state와 Nonterminal로 갔을 때 -1이 아닌 값이 i값이 있으면 i 상태로 이동!
    push(gotoTable[stack[top].state][getGotoIndex(newNode->symbol)], newNode->symbol);
    nodeStack[++nodeTop] = newNode;
}
```

production rule의 right side nonterminal과 terminal의 개수의 합만큼 자식 노드를 생성한다. numChildren 변수에 저장된 자식 노드의 수만큼 루프를 돌면서, nodeStack에서 자식 노드를 하나씩 꺼내어 newNode의 자식 노드 배열에 추가한다. nodeTop—을 이용해 자식 노드를 꺼내면서 스택 포인터를 감소시키고, newNode->children[newNode->childCount++]를 통해 자식 노드 배열에 추가하고 자식 노드의 수를 증가시킨다.

production rule의 right side의 nonterminal과 terminal의 개수의 합만큼 pop을 진행한다. pop의 결과가 gotoTable 의 state와 Nonterminal로 간 경우, -1이 아닌 i가 존재하면 i 상태로 이동한다. [pop 후 남은 스택의 최상위 상태(stack[top].state)와 새로 생성된 노드의 심볼(newNode->symbol)을 사용하여 gotoTable에서 다음 상태를 찾고 gotoTable에서 찾은 다음 상태로 push한다. 이때 푸시되는 상태는 gotoTable에서 찾은 값이고 이 값이 유효한 경우(-1이 아닌 값)에 다음 상태로 이동한다.nodeStack[++nodeTop] = newNode;을 이용해 생성된 새 노드를 nodeStack에 추가한다. nodeTop을 증가시키고, 증가된 위치에 newNode를 할당한다.]

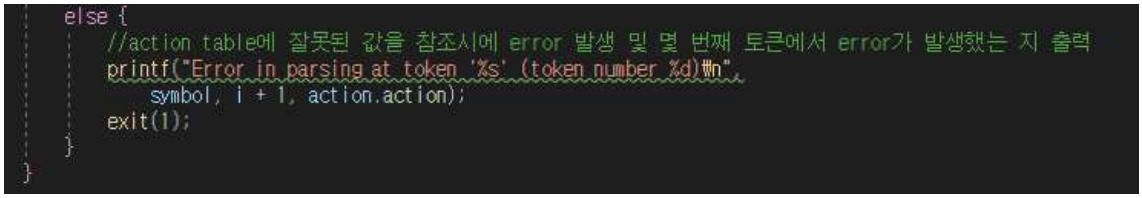
```
//action이 a(accept)라면?
else if (action.action == 'a') {
    //현재 nodeStack에 top요소를 parseTree에 저장
    parseTree = nodeStack[nodeTop];
    printf("Input string is accepted.₩n");
    //parseTree를 출력
    printParseTreePretty(parseTree, 0, -1);
    break;
}
```

action이 accept이면 현재 node stack의 top 요소를 parsetree에 저장하고 parstree를 출력한다.

else if (action.action == 'a') 을 통해 현재 파싱 액션(action.action)이 'a'(accept)를 나타내는 경우를 처리한다. 'a'는 입력 문자열이 파싱 규칙에 부합하여 파싱이 성공했음을 의미한다. parseTree = nodeStack[nodeTop]를 이용해 현재 nodeStack에 top요소를 parseTree에 저장한다. printf("Input string is accepted.\n");를 이용해 입력 문자열이 성공적으로 파싱되었음을 사용자에게 알리는 메시지를 출력한다.

printParseTreePretty(parseTree, 0, -1);을 이용해 parseTree를 출력한다. parseTree는 트리의 루트 노드, 0은 초기 들여쓰기 레벨, -1은 추가적인 스타일링 또는 트리의 깊이를 나타낸다.

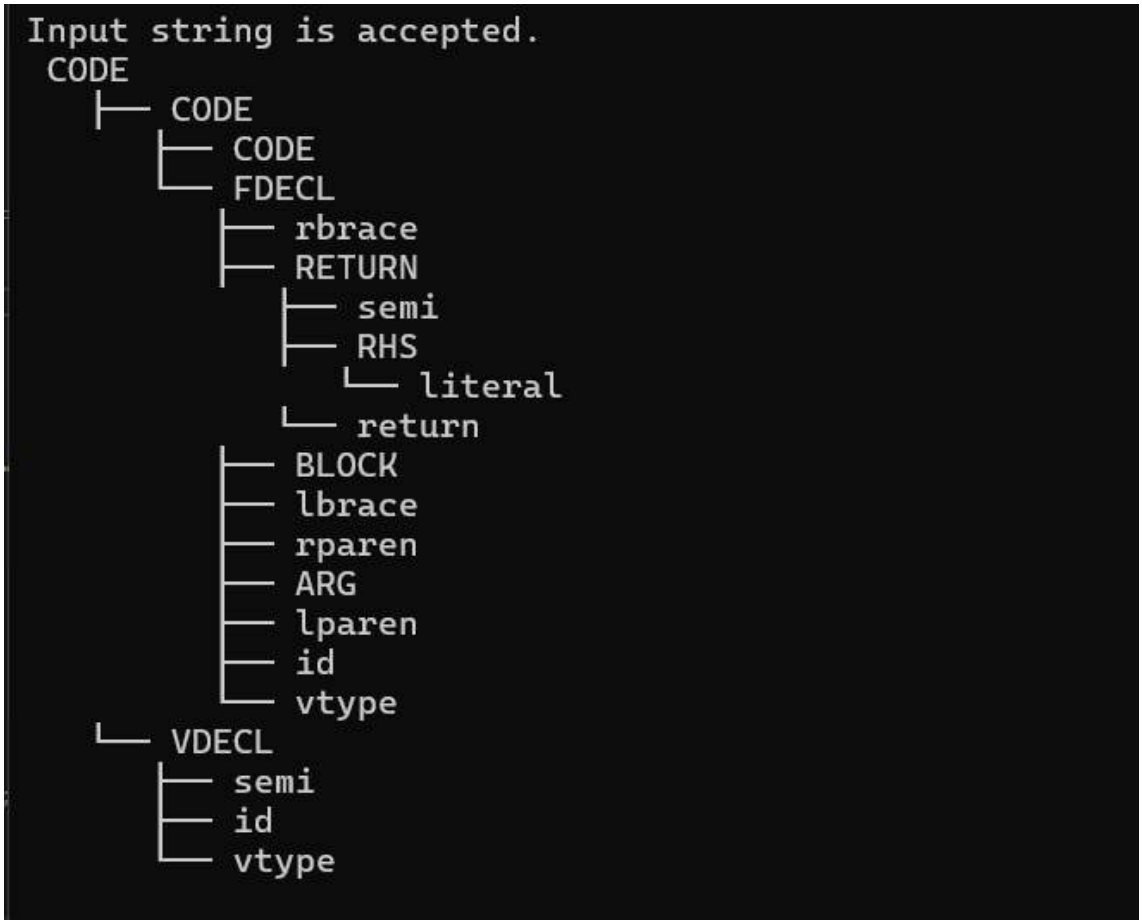
break;를 통해 입력 문자열이 성공적으로 파싱되었기 때문에 while 루프를 종료한다.



else에서는 if와 else if 조건을 모두 만족하지 않는 경우를 처리한다. 액션이 's'(shift)나 'r'(reduce), 'a'(accept)가 아닌 경우에 해당한다.

<Accept 되는 경우>

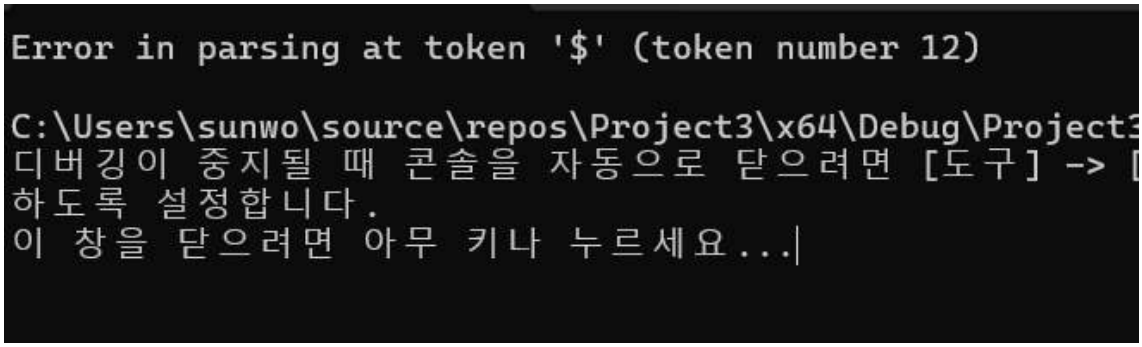
예시 : vtype id semi vtype id lparen rparen lbrace return literal semi rbrace



<Reject 되는 경우>

1)

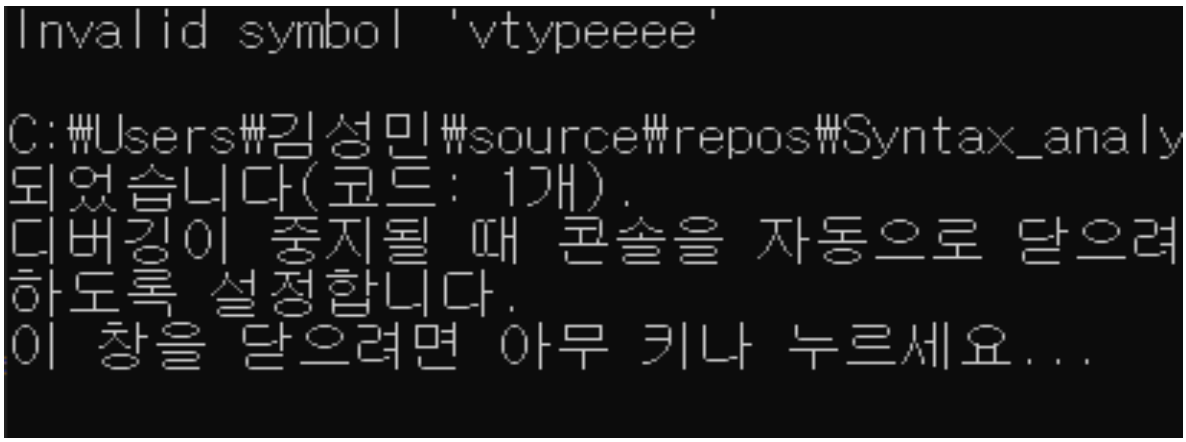
예시 : vtype id semi vtype id lparen rparen lbrace return literal semi



parsing 과정에서 오류가 날 시에는 몇 번째 token에서 오류가 발생하였는 지 출력해준다.

2)

예시 : vtypeeee id semi vtype id lparen rparen lbrace return literal semi rbrace



토큰 인식 과정에서 잘못된 token, 즉 인식할 수 없는 토큰이 입력된다면 다음과 같이 오류 메시지를 출력해준다.