#2 Parser

컴퓨터소프트웨어학부

2019054957 이용우

1. Compilation environment and method

- OS: Ubuntu 22.04.2 LTS (Windows 11 wsl2)
- Compiler: gcc 11.3.0, GNU Make 4.3, flex 2.6.4, bison (GNU Bison) 3.8.2
- Method: 주어진 Makefile에서 gcc, -ll 옵션을 변경하여 컴파일

2. How to implement & How it operates

C-Minus 의 BNF Grammar 에 맞춰 파싱 트리를 구현한다. Yacc 소스코드에서 Parsing Rules 에 대하여 트리 노드 생성 로직을 구현하였다. cminus.y 파일에 정의된 savedTree 를 syntax tree 로 정의하여 트리 노드를 추가하는 방식으로 동작한다. 본 보고서는 미리 구현되어 있던 Grammar 는 다루지 않고, 새로 작성한 Grammar Rule 에 대한 구현만을 다룬다.

1) fun_declaration -> type-specifier identifier (params) compound-stmt

```
fun_declaration     : type_specifier identifier LPAREN params RPAREN compound_stmt
                    {
                      $$ = newTreeNode(FunctionDecl);
                      $$->type = $1->type;
                      $$->name = $2->name;
                      $$->child[0] = $4;
                      $$->child[1] = $6;
                      free($1); free($2);
                    }
```

함수 선언문에 대한 grammar rule 이다. FunctionDecl 타입으로 트리 노드를 생성하고, 타입은 반환 타입(type_specifier)으로 설정한다. 함수의 이름은 identifier 의 이름을 가져온다. 함수의 parameter(params)와 내부 구현(compound_stmt)은 child 노드로 추가한다. Child 노드로 추가하지 않은 노드들은 이후에 사용되지 않으므로 free 로 메모리를 해제한다.

2) Params -> param_list | VOID

```
params              : param_list { $$ = $1; }
                    | VOID
                        {
                          $$ = newTreeNode(Params);
                          $$->lineno = lineno;
                          $$->type = Void;
                          $$->flag = TRUE;
                        }
```

함수 파라미터에 대한 grammar rule 이다. param_list 면 list 의 header 를 그대로 assign 한다. VOID 인 경우는 Params 타입으로 트리 노드를 생성하고, 타입은 Void 타입으로 설정한다. 두 Rule 의 출력을 구분하기 위해 void parameter 는 flag 값을 true 로 설정한다.

3) param_list -> param_list COMMA param | param

```
param_list          : param_list COMMA param
                    {
                        YYSTYPE t = $1;
                        if (t != NULL) {
                            while (t->sibling != NULL) {
                                t = t->sibling;
                            }
                            t->sibling = $3;
                            $$ = $1;
                        }
                        else {
                            $$ = $3;
                        }
                    }
                | param { $$ = $1; }
```

param_list 는 COMMA 로 이어진 param 을 linked list 형식으로 sibling node 에 추가한다. param_list 는 left-recursive 한 grammar 로 구현되었다.

4) param -> type_specifier identifier | type_specifier identifier LBRACE RBRACE

```
param               : type_specifier identifier
                        {
                            $$ = newTreeNode(Params);
                            $$->type = $1->type;
                            $$->name = $2->name;
                            free($1); free($2);
                        }
                    | type_specifier identifier LBRACE RBRACE
                        {
                            $$ = newTreeNode(Params);
                            $$->lineno = $2->lineno;
                            if ($1->type == Integer) $$->type = IntegerArray;
                            else if ($1->type == Void) $$->type = VoidArray;
                            else $$->type = None;
                            $$->name = $2->name;
                            free($1); free($2);
                        }
```

함수 parameter 를 선언하는 grammar 이다. Params 타입으로 트리 노드를 생성하고, 배열인지 아닌지의 여부에 따라 type 값을 다르게 설정한다. parameter 의 이름은 identifier 의 이름을 복사하여 가져온다. type_specifier 와 identifier 는 param 트리 노드에 속하지 않으므로 메모리를 free 한다.

5) compound_stmt -> LCURLY local_declarations statement_list RCURLY

```
compound_stmt       : LCURLY local_declarations statement_list RCURLY
                        {
                            $$ = newTreeNode(CompoundStmt);
                            $$->child[0] = $2;
                            $$->child[1] = $3;
                        }
```

중괄호 block 의 scope 에서 선언되는 stmt 이다. CompoundStmt 타입으로 트리 노드를 생성하고, local_declarations 와 statement_list 를 child 노드로 추가한다.

6) local_declarations -> local_declarations var_declaration | empty

```
local_declarations  : local_declarations var_declaration
                        {
                            YYSTYPE t = $1;
                            if (t != NULL) {
                                while (t->sibling != NULL) {
                                    t = t->sibling;
                                }
                                t->sibling = $2;
                                $$ = $1;
                            }
                            else {
                                $$ = $2;
                            }
                        }
                    | empty { $$ = $1; }
```

compound_stmt 에서의 변수 선언부에 대한 grammar 이다. var_declaration 을 linked list 형식으로 sibling node 에 추가한다. local_declarations 는 left-recursive 한 grammar 로 구현되었다. 변수 선언이 없다면 empty 에 해당하며, empty 노드를 그대로 assign 한다.

7) statement_list -> statement_list statement_list | empty

```
statement_list      : statement_list statement
                    {
                        YYSTYPE t = $1;
                        if (t != NULL) {
                            while (t->sibling != NULL) {
                                t = t->sibling;
                            }
                            t->sibling = $2;
                            $$ = $1;
                        }
                        else {
                            $$ = $2;
                        }
                    }
                  | empty { $$ = $1; }
```

compound_stmt 에서의 statement 들을 linked list 형식으로 sibling node 에 추가한다. statement_list 는 left-recursive 한 grammar 로 구현되었다. statement 가 없다면 empty 에 해당하며, empty 노드를 그대로 assign 한다.

8) selection_stmt -> IF LPAREN expression RPAREN statement ELSE statement
selection_stmt -> IF LPAREN expression RPAREN statement

```
selection_stmt      : IF LPAREN expression RPAREN statement ELSE statement
                    {
                        $$ = newTreeNode(IfStmt);
                        $$->flag = TRUE;
                        $$->child[0] = $3;
                        $$->child[1] = $5;
                        $$->child[2] = $7;
                    }
                  | IF LPAREN expression RPAREN statement
                    {
                        $$ = newTreeNode(IfStmt);
                        $$->child[0] = $3;
                        $$->child[1] = $5;
                    }
```

분기문을 처리하기 위한 grammar 이다. if-else 문은 flag 를 true 로 설정하여 else 가 없는 if 문과 구분한다. IfStmt 타입의 트리 노드를 생성한다. expression 과 statement 들을 child 노드로 추가한다.

9) expression_stmt, iteration_stmt, return_stmt

```
expression_stmt     : expression SEMI { $$ = $1; }
                    | SEMI { $$ = NULL; }
                    ;
iteration_stmt      : WHILE LPAREN expression RPAREN statement
                        {
                            $$ = newTreeNode(WhileStmt);
                            $$->child[0] = $3;
                            $$->child[1] = $5;
                        }
                    ;
return_stmt         : RETURN SEMI
                        {
                            $$ = newTreeNode(ReturnStmt);
                            $$->flag = TRUE;
                        }
                    | RETURN expression SEMI
                        {
                            $$ = newTreeNode(ReturnStmt);
                            $$->child[0] = $2;
                        }
```

expression_stmt 는 SEMI 로 구분하여 expression 을 가진다.

iteration_stmt 는 WhileStmt 타입으로 트리 노드를 생성하고 expression 과 statement 를 child 노드로 추가한다.

return_stmt 는 ReturnStmt 타입으로 트리 노드를 생성한다. 반환하는 데이터가 없다면 flag 값을 true 로 하여 void 타입으로 반환한다. 반환 값이 있다면 expression 을 child node 로 추가한다.

10) Expression, var

```
expression          : var ASSIGN expression
                        {
                           $$ = newTreeNode(AssignExpr);
                           $$->child[0] = $1;
                           $$->child[1] = $3;
                        }
                    | simple_expression { $$ = $1; }
                    ;
var                 : identifier
                        {
                           $$ = newTreeNode(VarAccessExpr);
                           $$->name = $1->name;
                           free($1);
                        }
                    | identifier LBRACE expression RBRACE
                        {
                           $$ = newTreeNode(VarAccessExpr);
                           $$->name = $1->name;
                           $$->child[0] = $3;
                           free($1);
                        }
```

변수에 값을 할당하는 expression 은 AssignExpr 타입으로 트리 노드를 생성한 후, var 와 expression 을 child 노드로 추가한다. simple_expression 은 그대로 사용한다.

변수에 접근하는 expression 은 VarAccessExpr 타입으로 트리 노드를 생성한 후, identifier 의 이름으로 설정한다. 배열에 접근하는 경우는 인덱스에 대한 expression 을 child 노드로 추가한다. Identifier 노드는 트리 노드에 속하지 않으므로 free 로 메모리 할당을 해제한다.

11) simple_expression -> additive_expression relop additive_expression

```
simple_expression   : additive_expression relop additive_expression
                        {
                           $$ = newTreeNode(BinOpExpr);
                           $$->opcode = $2->opcode;
                           $$->child[0] = $1;
                           $$->child[1] = $3;
                           free($2);
                        }
```

simple_expression 은 BinOpExpr 타입으로 트리 노드를 생성한 후 연산의 opcode 를 전달받아 설정한다. 연산의 operand 는 child 노드로 추가한다. Relop 은 트리 노드에 속하지 않으므로 free 한다.

12) additive_expression, addop, term, mulop, factor, call, args

```
additive_expression : additive_expression addop term
                        {
                           $$ = newTreeNode(BinOpExpr);
                           $$->opcode = $2->opcode;
                           $$->child[0] = $1;
                           $$->child[1] = $3;
                           free($2);
                        }
                    | term { $$ = $1; }
addop               : PLUS  { $$ = newTreeNode(Opcode); $$->lineno = lineno; $$->opcode = PLUS; }
                    | MINUS { $$ = newTreeNode(Opcode); $$->lineno = lineno; $$->opcode = MINUS; }
                    ;
term                : term mulop factor
                        {
                           $$ = newTreeNode(BinOpExpr);
                           $$->opcode = $2->opcode;
                           $$->child[0] = $1;
                           $$->child[1] = $3;
mulop               : TIMES { $$ = newTreeNode(Opcode); $$->lineno = lineno; $$->opcode = TIMES; }
                    | OVER  { $$ = newTreeNode(Opcode); $$->lineno = lineno; $$->opcode = OVER; }
                    ;
factor              : LPAREN expression RPAREN { $$ = $2; }
                    | var { $$ = $1; }
                    | call { $$ = $1; }
                    | number { $$ = $1; }
                    ;
call                : identifier LPAREN args RPAREN
                        {
                           $$ = newTreeNode(CallExpr);
                           $$->name = $1->name;
                           $$->child[0] = $3;
                           free($1);
                        }
                    ;
args                : arg_list { $$ = $1; }
                    | empty { $$ = $1; }
```

+, -, *, / 에 대한 연산에 대한 expression(additive_expression, term 은 simple_expression 과 동일하게 구현한다.

함수 호출문은 CallExpr 타입으로 트리 노드를 생성한다. 함수 이름은 identifier 의 이름으로 설정한다.

함수의 인자인 args 는 child 노드에 추가한다. Identifier 는 트리 노드에 속하지 않으므로 free 하여 메모리 할당을 해제한다.

3. Examples & Results

```
timel2ss@DESKTOP-DF5IA3P:~/compiler/parser/2019054957/2_Parser$ ./cminus_parser example/test.1.txt

C-MINUS COMPILATION: example/test.1.txt

Syntax tree:
  Function Declaration: name = gcd, return type = int
    Parameter: name = u, type = int
    Parameter: name = v, type = int
    Compound Statement:
      If-Else Statement:
        Op: ==
          Variable: name = v
          Const: 0
        Return Statement:
          Variable: name = u
        Return Statement:
          Call: function name = gcd
            Variable: name = v
            Op: -
              Variable: name = u
              Op: *
                Op: /
                  Variable: name = u
                  Variable: name = v
                Variable: name = v
  Function Declaration: name = main, return type = void
    Void Parameter
    Compound Statement:
      Variable Declaration: name = x, type = int
      Variable Declaration: name = y, type = int
      Assign:
        Variable: name = x
        Call: function name = input
      Assign:
        Variable: name = y
        Call: function name = input
      Call: function name = output
        Call: function name = gcd
          Variable: name = x
          Variable: name = y
```

Result Screenshot: test.1.txt

```
timel2ss@DESKTOP-DF5IA3P:~/compiler/parser/2019054957/2_Parser$ ./cminus_parser example/test.2.txt

C-MINUS COMPILATION: example/test.2.txt

Syntax tree:
  Function Declaration: name = main, return type = void
    Void Parameter
    Compound Statement:
      Variable Declaration: name = i, type = int
      Variable Declaration: name = x, type = int[]
        Const: 5
      Assign:
        Variable: name = i
        Const: 0
      While Statement:
        Op: <
          Variable: name = i
          Const: 5
        Compound Statement:
          Assign:
            Variable: name = x
              Variable: name = i
            Call: function name = input
          Assign:
            Variable: name = i
            Op: +
              Variable: name = i
              Const: 1
      Assign:
        Variable: name = i
        Const: 0
      While Statement:
        Op: <=
          Variable: name = i
          Const: 4
        Compound Statement:
          If Statement:
            Op: !=
              Variable: name = x
                Variable: name = i
              Const: 0
            Compound Statement:
              Call: function name = output
                Variable: name = x
                  Variable: name = i
```

Result Screenshot: test.2.txt

```
void main(void)
{
    return;
}
int add(int a, int b, void c, void d[],
int e[]) {
    if (c)
        if (a == 3) {
            a = 5;
        }
        else
            b = 5;
    if (a >= 5) {
        void x;
        int y;
        x = d[0];
        y = e[1];
    }
    return a + b;
}
```

test.3.txt

```
timel2ss@DESKTOP-DF5IA3P:~/compiler/parser/2019054957/2_Parser$ ./cminus_parser example/test.3.txt

C-MINUS COMPILATION: example/test.3.txt

Syntax tree:
  Function Declaration: name = main, return type = void
    Void Parameter
    Compound Statement:
      Non-value Return Statement
  Function Declaration: name = add, return type = int
    Parameter: name = a, type = int
    Parameter: name = b, type = int
    Parameter: name = c, type = void
    Parameter: name = d, type = void[]
    Parameter: name = e, type = int[]
    Compound Statement:
      If Statement:
        Variable: name = c
        If-Else Statement:
          Op: ==
            Variable: name = a
            Const: 3
          Compound Statement:
            Assign:
              Variable: name = a
              Const: 5
          Assign:
            Variable: name = b
            Const: 5
      If Statement:
        Op: >=
          Variable: name = a
          Const: 5
        Compound Statement:
          Variable Declaration: name = x, type = void
          Variable Declaration: name = y, type = int
          Assign:
            Variable: name = x
            Variable: name = d
              Const: 0
          Assign:
            Variable: name = y
            Variable: name = e
              Const: 1
      Return Statement:
        Op: +
          Variable: name = a
          Variable: name = b
```

Result Screenshot: test.3.txt