Paser

2018009125 조성우

1. 컴파일 환경

Ubuntu 18.04.6 Lts에서 동봉된 Makefile을 통해 make명령어를 입력하여 compile하였습니다.

2. 코드구현

수정된 부분들에 대해서만 다뤄보겠습니다. 다음은 cminus.y의 코드들 입니다.

```
74  fun_declaration     : type_specifier identifier LPAREN params RPAREN compound_stmt
75                          {
76                              $$ = newTreeNode(FunctionDecl);
77                              $$->lineno = $2->lineno;
78                              $$->type = $1->type;
79                              $$->name = $2->name;
80                              $$->child[0] = $4;
81                              $$->child[1] = $6;
82                              free($1); free($2);
83                          }
```

먼저 fun_declaration입니다. newTreeNode(FunctionDecl); 새로운 FunctionDecl Treenode를 만들었으며 name과 lineno은 identifier의 것을 그대로 받았고 type은 type_specifier의 것을 그대로 받았습니다. 그리고 child를 두었는데 params를 첫번째 child로, compound_stmt를 두번째 child로 두었습니다. 마지막으로 필요한 정보들을 빼내서 더 이상 필요 없어진 type_specifier와 identifier를 free합니다.

```
85  params              : param_list { $$ = $1; }
86                      | VOID
87                          {
88                              $$ = newTreeNode(Params);
89                              $$->flag = 1;
90                              $$->lineno = lineno;
91
92                          }
```

다음은 params입니다. Param_list일 경우 그대로 param_list를 가리키게 하였고, 만약 파라미터가 존재하지 않는 즉, VOID라면 params를 kind로 갖는 새로운 TreeNode를 만들고 그것의 flag=1로 설정하였습니다. 그리고 tree의 lineno은 현재의 lineno을 갖게 하였습니다. 여기서 flag = 1의 의미는 params가 VOID인 특수한 경우를 나타내는 것이며 다음과 같이 util.c를 보게 되면

```
132             case Params:
133                 if (tree->flag == TRUE) fprintf(listing, "Void Parameter\n");
134                 else
135                     fprintf(listing, "Parameter: name = %s, type = %s\n", tree->name, TYPE2STR(tree->type));
136                 break;
```

Flag = TRUE일 경우 VOID parameter를 출력하는 것을 알 수 있습니다.

```
94   param_list          : param_list COMMA param
95                           {
96                                YYSTYPE t = $1;
97                                if (t != NULL)
98                                {
99                                     while (t->sibling != NULL) t = t->sibling;
100                                    t->sibling = $3;
101                                    $$ = $1;
102                               }
103                               else $$ = $3;
104                          }
105                     | param { $$ = $1; }
106                     ;
```

다음은 param_list입니다. list형식 이기때문에 sibiling들을 이어주는 작업을 하였습니다. 먼저 t = $1로 초기화하였고 t가 존재하지 않을 경우 즉, NULL인 경우는 else문의 $$= $3을 통해 그냥 param을 가리키게 하였으며 만약 t가 존재할 경우, while문을 통해 t의 sibling중 가장 꼬리에 있는 sibling을 찾아낸 다음 그것의 sibling을 $3(param)으로 설정하여 sibling이 연결되도록 하였습니다. 이 코드는 35줄의 기본으로 주어진 declaration_list의 코드를 참고하여 작성하였습니다. param_list에서 parameter가 하나만 존재할 경우 105줄이 실행되도록 하였습니다.

```
107  param               : type_specifier identifier
108                          {
109                               $$ = newTreeNode(Params);
110                               $$->lineno = $2->lineno;
111                               $$->type = $1->type;
112                               $$->name = $2->name;
113                               free($1); free($2);
114                          }
115                     | type_specifier identifier LBRACE RBRACE
116                          {
117                               $$ = newTreeNode(Params);
118                               $$->lineno = $2->lineno;
119                               if ($1->type == Integer) $$->type = IntegerArray;
120                               else if ($1->type == Void) $$->type = VoidArray;
121                               else $$->type = None;
122                               $$->name = $2->name;
123                               free($1); free($2);
124                          }
125                     ;
```

다음은 param의 관한 코드입니다. param에는 두 종류가 있는데, 하나는 일반적인 파라미터이고 나머지 하나는 배열형식의 파라미터입니다. 따라서 두개를 구분하였고 이 두개의 정의는 이미 작성 되어있던 var_declaration을 참조하였습니다. 다음은 compound_stmt와 local_declarations입니다.

```
126  compound_stmt       : LCURLY local_declarations statement_list RCURLY
127                          {
128                               $$ = newTreeNode(CompoundStmt);
129                               $$->lineno = lineno;
130                               $$->child[0] = $2;
131                               $$->child[1] = $3;
132                          }
133                     ;
134  local_declarations  : local_declarations var_declaration
135                          {
136                               YYSTYPE t = $1;
137                               if (t != NULL)
138                               {
139                                    while (t->sibling != NULL) t = t->sibling;
140                                    t->sibling = $2;
141                                    $$ = $1;
142                               }
143                               else $$ = $2;
144                          }
145                     | empty { $$ = $1;}
```

Compound_stmt 부터 보면 CompoundStmt 의 kind 를 가지고 있는 새로운 TreeNode 를 만들었고 lineno 은 현재의 lineno 을 가리켰으며 첫번째 child 로 local_declarations 를, 두번째 child 로 statement_list 를 두었습니다. 다음은 local_declarations 입니다. Param_list 와 마찬가지로 sibling 을 이어주는 작업을 진행하였으며 local_declarations 가 공란일 경우, 즉 empty 일 경우 그대로 empty 를 받도록 하였습니다. (Empty 는 NULL 을 가리키고 있습니다.) 다음은 statement_list 입니다.

```
147 statement_list      : statement_list statement
148                         {
149                             YYSTYPE t = $1;
150                             if (t != NULL)
151                             {
152                                 while (t->sibling != NULL) t = t->sibling;
153                                 t->sibling = $2;
154                                 $$ = $1;
155                             }
156                             else $$ = $2;
157                         }
158                     | empty { $$ = $1; }
```

Local_declarations와 똑같이 작성하였습니다.

```
185 expression_stmt     : expression SEMI { $$ = $1; }
186                     | SEMI { $$=NULL; }
187                     ;
188 iteration_stmt     : WHILE LPAREN expression RPAREN statement
189                         {
190                             $$ = newTreeNode(WhileStmt);
191                             $$->lineno = lineno;
192                             $$->child[0] = $3;
193                             $$->child[1] = $5;
194
195                         }
196                     ;
197 return_stmt        : RETURN SEMI
198                         {
199                             $$ = newTreeNode(ReturnStmt);
200                             $$->flag = 1;
201                             $$->lineno = lineno;
202                         }
203                     | RETURN expression SEMI
204                         {
205                             $$ = newTreeNode(ReturnStmt);
206                             $$->lineno = lineno;
207                             $$->child[0] = $2;
208                         }
209                     ;
```

```
166 selection_stmt     :     IF LPAREN expression RPAREN statement ELSE statement
167                         {
168                             $$ = newTreeNode(IfStmt);
169                             $$->lineno = lineno;
170                             $$->flag = 1;
171                             $$->child[0] = $3;
172                             $$->child[1] = $5;
173                             $$->child[2] = $7;
174
175                         }
176                     | IF LPAREN expression RPAREN statement
177                         {
178                             $$ = newTreeNode(IfStmt);
179                             $$->lineno = lineno;
180                             $$->child[0] = $3;
181                             $$->child[1] = $5;
182
183                         }
184                 ;
```

먼저 왼쪽은 expression_stmt와 iteration_stmt, return_stmt입니다. Expression_stmt에서 expression SEMI라면 그대로 expression을 가리키도록 하였으며 SEMIT는 아무런 의미가 없기 때문에 NULL로 설정하였습니다. Iteration_stmt는 kind를 Whilestmt로 설정한 새로운 TreeNode를 만들고 자식들을 설정해주었습니다. Return_stmt는 두가지 종류가 있는데 return값이 있는 경우와 없는 경우입니다. return값이 존재하지 않는 경우나 존재하는 경우 둘다 kind가 Returnstmt인 새로운 TreeNode를 만들었습니다. 이 중 전자는 param의 VOID와 같이 특수한 케이스이기 때문에 flag값을 1로 설정함으로써 나타낼 수 있습니다. 후자의 경우 child하나를 설정하였습니다. 그리고 오른

쪽은 selection_stmt로 if 문과 if else문을 나타내도록 하였습니다. 둘의 형식은 거의 동일하며 차이점은 if else문을 알아볼 수 있도록 if else문에는 flag를 1로 설정하였습니다.

다음은 expression과 var입니다.

```
expression          : var ASSIGN expression
                        {
                            $$ = newTreeNode(AssignExpr);
                            $$->lineno = lineno;
                            $$->child[0] = $1;
                            $$->child[1] = $3;

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

Expression은 a=3 처럼 나타낼 수 있는 경우와 그렇지 않은 경우를 나타내었으며 expression에서 kind의 종류만 바뀐 것을 빼면 두 경우 모두 특이점 없이 여타 다른 코드들과 동일한 형태로 만들었습니다. 여기서 var의 경우 위의 예시처럼 a인 경우가 있을 수 있고 a[2] = 3에서 a[2]처럼 배열의 형식을 띌 수 가 있기 때문에 두 가지 경우로 나누었으며 마찬가지로 kind만 빼면 특이점은 딱히 없습니다.

```
simple_expression   : additive_expression relop additive_expression
                    {
                        $$ = $2;
                        $$->child[0] = $1;
                        $$->child[1] = $3;

                    }
                    | additive_expression { $$ = $1; }
                    ;
relop               : LE { $$ = newTreeNode(BinOpExpr); $$->lineno = lineno; $$->opcode = LE; }
                    | LT { $$ = newTreeNode(BinOpExpr); $$->lineno = lineno; $$->opcode = LT; }
                    | GT { $$ = newTreeNode(BinOpExpr); $$->lineno = lineno; $$->opcode = GT; }
                    | GE { $$ = newTreeNode(BinOpExpr); $$->lineno = lineno; $$->opcode = GE; }
                    | EQ { $$ = newTreeNode(BinOpExpr); $$->lineno = lineno; $$->opcode = EQ; }
                    | NE { $$ = newTreeNode(BinOpExpr); $$->lineno = lineno; $$->opcode = NE; }
                    ;
additive_expression : additive_expression addop term
                        {
                            $$ = $2;
                            $$->child[0] = $1;
                            $$->child[1] = $3;
                        }
                        | term { $$ = $1; }
addop               : PLUS  { $$ = newTreeNode(BinOpExpr); $$->lineno = lineno; $$->opcode = PLUS;  }
                        | MINUS { $$ = newTreeNode(BinOpExpr); $$->lineno = lineno; $$->opcode = MINUS; }
                        ;
```

simple_expression과 relop, additive_expression, addop부분입니다. Simple_expression은 일반적인 additive_expression으로 구성된 경우와 중간에 relop이 낀 형태인 경우, 이렇게 두가지로 나눌 수가 있습니다. 차이점이 있다면 relop이 낀 경우 $$는 relop을 가리키고 좌우의 것을 자식으로 받고 relop이 없는 경우는 그대로 additive_expression을 가리킵니다. 다음은 relop입니다. 특이한점은 여기서 opcode를 사용하게 되며 이것은 operator들 간의 식별자 역할을 하게 됩니다. operator들을 kind가 BinOpExpr인 TreeNode로 설정을 하였으며 각각에 맞는 opcode들을 설정해 주었습니다. Addop는 +와 *의 operator들을 나타내도록 opcode를 설정하였습니다.

Additive_expression은 list형식이 아니기 때문에 sibling을 이어주는 작업을 하지 않았으며 addop의 node를 그대로 가리켰으며 양쪽의 식을 child로 받도록 하였습니다. 혹은 term을 그대로 받도

록 하였습니다.

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

마지막으로 term, mulop, factor, call입니다. Term은 additive_expression의 형식과 동일하며 mulop 도 addop와 마찬가지로 opcode부분만 빼면 동일한 형식입니다. factor에는 여러가지 형식이 있는데 이 중 call을 살펴보면 CallExpr를 kind로 갖는 TreeNode를 생성하여 identifier의 name, lineno을 참조하였고 자식으로 인수를 가리키는 args를 설정하였습니다. 그리고 쓸모 없어진 identifier를 free하였습니다.

3. Test 결과

다음은 각각 test.1.txt와 test.2.txt의 결과입니다.

```
chosungwoo@chosungwoo:~/Desktop/Paser$ ./cminus_parser test.1.txt

C-MINUS COMPILATION: 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
chosungwoo@chosungwoo:~/Desktop/Paser$
```

```
chosungwoo@chosungwoo:~/Desktop/Paser$ ./cminus_parser test.2.txt

C-MINUS COMPILATION: 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
chosungwoo@chosungwoo:~/Desktop/Paser$
```