Semantics

2018009125 조성우

1. 컴파일 환경

VM에서 Ubuntu 18.04.6 LTS버전으로 동봉된 Makefile을 통해 컴파일을 하였습니다.

2. 코드구현

먼저 insertNode함수내용부터 말씀드리겠습니다.

```
149         // Variable Declaration
150         case VariableDecl:
151         {
152             // Semantic Error: Void-Type Variables
153             if (t->type == Void || t->type == VoidArray) VoidTypeVariableError(t->name, t->lineno);
154             // Semantic Error: Redefined Variables
155             SymbolRec *symbol = lookupSymbolInCurrentScope(currentScope, t->name);
156             if (symbol != NULL) RedefinitionError(t->name, t->lineno, symbol);
157             // Insert New Variable Symbol to Symbol Table
158             insertSymbol(currentScope, t->name, t->type, VariableSym, t->lineno, t);
159             // Break
160             break;
161         }
162         // Function Declaration
163         case FunctionDecl:
164         {
165             // Error Check: currentScope is not global
166             ERROR_CHECK(currentScope == globalScope);
167             // Semantic Error: Redefined Variables
168             SymbolRec *symbol = lookupSymbolInCurrentScope(globalScope, t->name);
169             if (symbol != NULL) RedefinitionError(t->name, t->lineno, symbol);
170             // Insert New Function Symbol to Symbol Table
171             insertSymbol(currentScope, t->name, t->type, FunctionSym, t->lineno, t);
172             // Change Current Scope
173             currentScope = t->scope = insertScope(t->name, currentScope, t);
174             // Break
175             break;
176         }
```

VariableDecl과 FunctionDecl입니다. 둘 다 lookupSymbolInCurrentScope라는 함수를 각각 currentScope(var)와 globalScope(Func)에서 중복되는 symbol이 있는지 확인하였고 중복되는 symbol이 존재한다면 Error를 발생시켰습니다. 중복되는 symbol이 없다면 둘 다 insertSymbol을 통해 symbol table에 추가해주었습니다. 참고로 FunctionDecl쪽 insertSymbol에서 currentScope를 넣었는데 이 때 어차피 currentScope == globalScope이기 때문에 문제가 발생하지 않았습니다. 그리고 FunctionDecl에서는 추가로 Function scope까지 insertScope를 통해 생성해주었습니다.

```
178         case Params:
179         {
180             // Void Parameters: Do Nothing
181             if (t->flag == TRUE) break;
182 
183             // Semantic Error: Void-Type Parameters
184             if (t->type == Void || t->type == VoidArray) VoidTypeVariableError(t->name, t->lineno);
185 
186             // Semantic Error: Redefined Variables
187             SymbolRec *symbol = lookupSymbolInCurrentScope(currentScope, t->name);
188             if (symbol != NULL) RedefinitionError(t->name, t->lineno, symbol);
189 
190             // Insert New Variable Symbol to Symbol Table
191 
192             insertSymbol(currentScope, t->name, t->type, VariableSym, t->lineno, t);
193             // Break
194             break;
195         }
```

다음은 Params입니다. 먼저 t->flag를 통해 VOID인지 확인하였습니다. 그 다음엔 VOID, VOIDArray타입인지 확인하여 Error를 발생시켰습니다. 그 다음은 마찬가지로 중복 symbol이 있는지 확인하였고 없으면 insertSymbol을 통해 현재 scope에 새로운 symbol을 추가하였습니다.

```
205         case CallExpr:
206         {
207             // Semantic Error: Undeclared Functions
208             SymbolRec *func = lookupSymbolWithKind(globalScope, t->name, FunctionSym);
209             if (func == NULL) func = UndeclaredFunctionError(globalScope, t);
210             // Update Symbol Table Entry
211             else
212                 appendSymbol(globalScope, t->name, t->lineno);
213             // Break
214             break;
215         }
216         // Variable Access
217         case VarAccessExpr:
218         {
219             // Semantic Error: Undeclared Variables
220             SymbolRec *var = lookupSymbolWithKind(currentScope, t->name, VariableSym);
221             if(var == NULL) var = UndeclaredVariableError(currentScope, t);
222             // Update Symbol Table Entry
223 
224 
225             else{
226                 appendSymbol(currentScope, t->name, t->lineno);
227 
228             }
229             // Break
230             break;
```

그 다음은 CallExpr과 VarAccessExpr입니다. 둘의 형식은 거의 똑같고 appendSymbol을 통해 어디서 참조되었는지 추가만 해주었습니다.

```
311         case IfStmt:
312         case WhileStmt:
313         {
314             // Error Check
315             ERROR_CHECK(t->child[0] != NULL);
316             // Semantic Error: Invalid Condition in If/If-Else, While Statement
317 
318             TreeNode* conditionNode = t->child[0];
319             //checkNode(conditionNode);
320             if(conditionNode->type != Integer){
321                 InvalidConditionError(t->lineno);
322             }
323             // Break
324             break;
325         }
```

다음은 CheckNode입니다. 319줄에서처럼 처음에는 child의 typecheck가 안되었다고 생각하였었는데 알고 보니 traverse함수에서 DFS형식으로 childNode를 checkNode함수에 넣어 typecheck를 끝마친 상황이었습니다. 따라서 굳이 한 번 더 호출할 필요가 없으므로 주석 처리하게 되었습니다. 320줄에서 child[0]즉 expr부분이 integer가 아니면 오류가 나게끔 처리하였습니다.

```
327         case ReturnStmt:
328         {
329             // Error Check
330             ERROR_CHECK(currentScope->func != NULL);
331             // Semantic Error: Invalid Return
332 
333 
334             if(t->flag == 1){
335                 if(currentScope->func->type != Void) InvalidReturnError(t->lineno);
336             }
337             else{
338                 //checkNode(t->child[0]);
339                 if(t->child[0]->type != currentScope->func->type) InvalidReturnError(t->lineno);
340             }
341 
342             // Break
343             break;
344         }
```

다음은 ReturnStmt입니다. t->flag가 1이라는 것은 empty return을 뜻하므로 함수의 return type이 void가 아닐 경우 error처리를 하였습니다. 그리고 empty return이 아닐 경우 child[0]의 type과 함수의 return type이 같은 지 확인후 다르다면 error를 띄웠습니다.

```
346         case AssignExpr:
347         case BinOpExpr:
348         {
349             // Error Check
350             ERROR_CHECK(t->child[0] != NULL && t->child[1] != NULL);
351             // Semantic Error: Invalid Assignment / Operation
352 
353             if(t->child[0]->type != Integer || t->child[1]->type != Integer){
354                 if(t->kind == AssignExpr) InvalidAssignmentError(t->lineno);
355                 else InvalidOperationError(t->lineno);
356                 break;
357             }
358 
359 
360             // Update Node Type
361             t->type = t->child[0]->type;
362             // Break
363             break;
364         }
```

다음은 AssignExpr과 BinOpExpr입니다. 둘 다 child[0]과 child[1]의 type이 integer가 아니라면 에러를 띄웠고 t->kind에 따라 에러메시지를 다르게 설정하였습니다. 만약 둘 다 integer로 구성돼 있다면 정상적이므로 t->type을 t->child[0]->type, 즉 사실상 integer로 설정하였습니다.

```
366         case CallExpr:
367         {
368             SymbolRec *calleeSymbol = lookupSymbolWithKind(globalScope, t->name, FunctionSym);
369             // Error Check
370             ERROR_CHECK(calleeSymbol != NULL);
371             // Semantic Error: Call Undeclared Function - Already Caused
372             if (calleeSymbol->state == STATE_UNDECLARED)
373             {
374                 t->type = calleeSymbol->type;
375                 break;
376             }
377             // Semantic Error: Invalid Arguments
378             TreeNode *paramNode = calleeSymbol->node->child[0];
379             TreeNode *argNode = t->child[0];
380             while(paramNode != NULL || argNode != NULL){
381                 if(paramNode == NULL && argNode != NULL){
382                     InvalidFunctionCallError(t->name, t->lineno);
383                     break;
384                 }
385                 else if(paramNode != NULL && argNode == NULL){
386                     if(paramNode->flag == 1) break;
387                     else{
388                         InvalidFunctionCallError(t->name, t->lineno);
389                         break;
390                     }
391                 }
392 
393                 //checkNode(paramNode);
394                 //checkNode(argNode);
```

```
393                 //checkNode(paramNode);
394                 //checkNode(argNode);
395                 if(paramNode->type != argNode->type || paramNode->flag == 1){
396 
397                     InvalidFunctionCallError(t->name, t->lineno);
398                     break;
399                 }
400                 else{
401                     paramNode = paramNode->sibling;
402                     argNode = argNode->sibling;
403                 }
404 
405 
406             }
407 
408             // Update Node Type
409             t->type = calleeSymbol->type;
410             // Break
411             break;
412         }
```

다음은 CallExpr입니다. 368줄에서 table lookup을 통해 알맞은 symbol을 가져옵니다. 여러 error처리를 거친 후 380줄입니다. 여기서부터 함수 인자들을 비교합니다. 둘 중에 하나라도 null이 아닐 경우 계속 while문을 진행합니다. 381줄에서 선언된 인자 수보다 더 많은 인자를 넣어 call을 진행한 경우의 error를 띄웁니다. 386줄에서는 예를 들어 int a(VOID)로 선언되었고 a(); 로 호출한 경우 올바른 상황이므로 break를 해주었습니다. 이와 같은 경우가 아닌 상황은 call했을 때 인자가 필요한 것보다 부족한 경우이므로 error를 띄웠습니다. 395줄에서는 둘 다 NULL이 아니지만 type이 다른 경우와 함수 선언당시 VOID로 인자를 선언하였는데 다른 인자가 들어온 경우의 error를 처리하였습니다. 모든 경우에 해당되지 않았다면 정상적인 진행이므로 각자 sibling을 통해 다음 인자를 검사할 수 있도록 하였습니다. 둘 다 NULL이 되어 while문을 나왔다면 t->type을 함수의 return type으로 선언하였습니다.

```
414         case VarAccessExpr:
415         {
416             SymbolRec *symbol = lookupSymbolWithKind(currentScope, t->name, VariableSym);
417             // Error Check
418             ERROR_CHECK(symbol != NULL);
419             // Semantic Error: Access Undeclared Variable - Already Caused
420             if (symbol->state == STATE_UNDECLARED)
421             {
422                 t->type = symbol->type;
423                 break;
424             }
425             // Array Access or Not
426             if (t->child[0] != NULL)
427             {
428                 //checkNode(t->child[0]);
429 
430                 // Semantic Error: Index to Not Array
431                 if(symbol->type != IntegerArray) {
432                     ArrayIndexingError2(t->name, t->lineno);
433                 }
434                 // Semantic Error: Index is not Integer in Array Indexing
435                 if(t->child[0]->type != Integer){
436                     ArrayIndexingError(t->name, t->lineno);
437                 }
438 
439                 // Update Node Type
440                 t->type = Integer;
441             }
442             // Update Node Type
443             else
444                 t->type = symbol->type;
445             // Break
446             break;
447         }
```

다음은 마지막으로 varAccessExpr입니다. 다른case들과 마찬가지로 여러 error처리를 거친 후 426줄부터 보겠습니다. 426줄에서부터는 array인 상황을 처리하는 과정을 나타냅니다. 먼저 431줄에서 IntegerArray가 아닌 경우 error를 처리하였고 435줄에서 index가 integer가 아닌 경우를 처리하였습니다. 정상 진행될 경우 t->type을 Integer로 설정하였습니다. 444줄에서는 array가 아니라 일반 변수인 상황을 처리하였습니다.

3. test결과

pdf에서 설명하신대로 ./testcase_result.sh을 실행하여 모든 result들을 비교하였습니다. 다행히 myresult와 예시로 나온 result의 모든 결과가 동일하였습니다. 과제 편의 많이 봐주셔서 감사드리며 한학기동안 고생하셨습니다. 감사합니다.