#3 Semantic Analysis

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 컴파일러로 gcc를 사용하도록 변경하여 컴파일

2. How to implement & How it operates

1) Symbol Table 자료구조 분석

```c
// Struct: Symbol Table
typedef struct SymbolRec
{
    // Attributes: Name, Error, Type, Kind, Line, Memory Location, Node
    char *name;
    SemanticErrorState state;
    NodeType type;
    SymbolKind kind;
    LineList lineList;
    int memloc;
    TreeNode *node;
    // List Structures
    struct SymbolRec *next;
} SymbolRec, *SymbolList;
```

Symbol Table

```c
// Struct: Scope
typedef struct ScopeRec
{
    // Attributes: Name, Function Node
    char *name;
    SemanticErrorState state;
    TreeNode *func;
    // Symbol Tables in This Scope
    SymbolList symbolList[SIZE];
    int numSymbols;
    // Tree & List Structures
    int numScopes;
    struct ScopeRec *parent;
    struct ScopeRec *next;
} ScopeRec, *ScopeList;
```

Scope

Symbol table은 변수나 함수의 symbol을 저장한다. 새로운 변수나 함수를 선언하면 Symbol table에 insert되며, access할 때마다 해당 symbol을 찾아서 반환해야 한다. 이러한 연산이 빈번하게 일어나기 때문에 hash를 이용하여 접근하도록 한다.

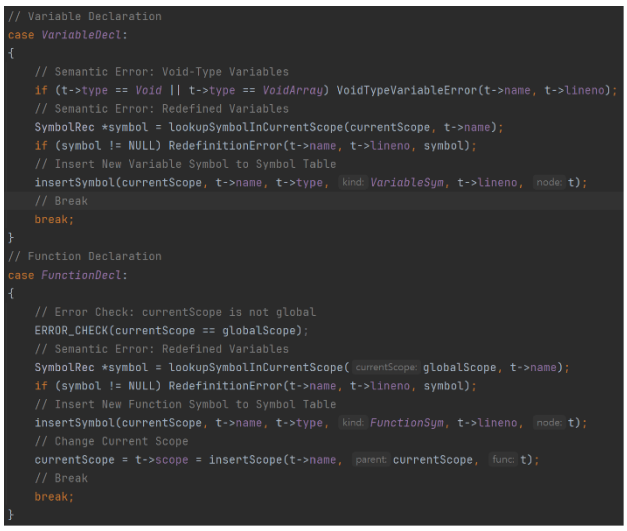
Symbol table은 scope마다 생성되어 Tree와 List의 형태로 구현된다. hash를 이용하여 symbol들을 저장하는 list를 가지며, Current scope에서 symbol을 찾지 못하면 상위 scope으로 search 범위를 확장할 수 있게 tree의 구조를 가진다. 최초에는 Global scope으로 시작하여 built-in function들을 symbol로 가진다. 이후에 compound statement를 만날 때마다 scope을 생성하여 관리한다.

2) Semantic Analysis Process

Parser로부터 얻은 AST(Abstract Syntax Tree)를 여러 차례 순회하며 Semantic analysis를 수행한다. 처음에는 Symbol table을 build하는 과정이다. 트리를 순회하면서 Symbol table에 symbol을 추가하는 작업을 수행한다. 트리의 노드마다 insertNode 함수를 호출하면서 동작한다. 이후에는 Type checking을 하는 과정을 거친다. 트리를 다시 순회하면서 Semantic Rules에 맞는 타입 구조인지 검증하며 타입 정보를 업데이트한다. 트리의 노드마다 checkNode 함수를 호출하면서 동작한다.

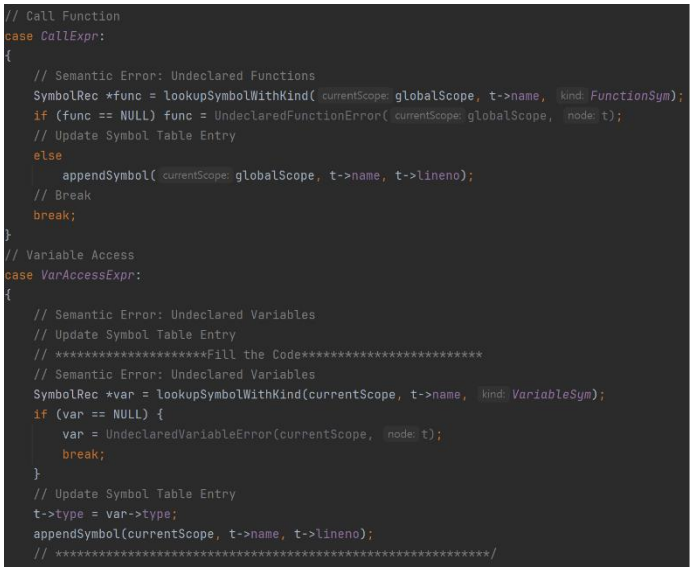
3) insertNode

insertNode는 Symbol table에 변수나 함수의 symbol을 추가하는 작업이다. symbol을 추가하는 작업 이외에도 몇 가지 Type checking을 한다. 예를 들어, void 타입의 변수나 파라미터를 선언한 경우가 이에 해당한다.



```
// Variable Declaration
case VariableDecl:
{
    // Semantic Error: Void-Type Variables
    if (t->type == Void || t->type == VoidArray) VoidTypeVariableError(t->name, t->lineno);
    // Semantic Error: Redefined Variables
    SymbolRec *symbol = lookupSymbolInCurrentScope(currentScope, t->name);
    if (symbol != NULL) RedefinitionError(t->name, t->lineno, symbol);
    // Insert New Variable Symbol to Symbol Table
    insertSymbol(currentScope, t->name, t->type,  kind: VariableSym, t->lineno,  node: t);
    // Break
    break;
}
// Function Declaration
case FunctionDecl:
{
    // Error Check: currentScope is not global
    ERROR_CHECK(currentScope == globalScope);
    // Semantic Error: Redefined Variables
    SymbolRec *symbol = lookupSymbolInCurrentScope( currentScope: globalScope, t->name);
    if (symbol != NULL) RedefinitionError(t->name, t->lineno, symbol);
    // Insert New Function Symbol to Symbol Table
    insertSymbol(currentScope, t->name, t->type,  kind: FunctionSym, t->lineno,  node: t);
    // Change Current Scope
    currentScope = t->scope = insertScope(t->name,  parent: currentScope,  func: t);
    // Break
    break;
}
```

다음은 변수나 함수를 선언하는 부분이다. 우선 Current scope에 이름이 같은 symbol이 있는지 lookup한다. 만약 동일한 symbol이 있다면 redefine하는 것이기 때문에, 에러로 처리해야 한다. Lookup 결과가 null이라면 동일한 이름의 symbol을 찾지 못한 것이기 때문에, insertSymbol로 Current scope에 symbol을 추가한다. 함수의 경우는 함수의 body 부분이 compound statement로 되어 있어서 새로운 scope을 추가한다.



```
// Call Function
case CallExpr:
{
    // Semantic Error: Undeclared Functions
    SymbolRec *func = lookupSymbolWithKind( currentScope: globalScope, t->name,  kind: FunctionSym);
    if (func == NULL) func = UndeclaredFunctionError( currentScope: globalScope,  node: t);
    // Update Symbol Table Entry
    else
        appendSymbol( currentScope: globalScope, t->name, t->lineno);
    // Break
    break;
}
// Variable Access
case VarAccessExpr:
{
    // Semantic Error: Undeclared Variables
    // Update Symbol Table Entry
    // ********************Fill the Code*************************
    // Semantic Error: Undeclared Variables
    SymbolRec *var = lookupSymbolWithKind(currentScope, t->name,  kind: VariableSym);
    if (var == NULL) {
        var = UndeclaredVariableError(currentScope,  node: t);
        break;
    }
    // Update Symbol Table Entry
    t->type = var->type;
    appendSymbol(currentScope, t->name, t->lineno);
    // ************************************************************/
```

다음은 변수나 함수에 access 하는 부분이다. 이름을 기준으로 Symbol table을 lookup하여 symbol을 찾는다. 함수의 경우는 global scope에서만 정의될 수 있으므로 global scope을, 변수는 current scope을 기준으로 lookup한다. 만약 symbol을 발견하지 못하면 undefined로 에러 처리한다.

4) checkNode

checkNode는 Semantic Rule에 맞는 타입을 가지는지 검증하는 작업이다.

```
// If/If-Else, While Statement
case IfStmt:
case WhileStmt:
{
    // Error Check
    ERROR_CHECK(t->child[0] != NULL);
    // Semantic Error: Invalid Condition in If/If-Else, While Statement
    // ********************Fill the Code************************
    TreeNode *condition = t->child[0];
    if (condition->type != Integer) {
        InvalidConditionError(t->lineno);
    }
```

IfStmt와 WhileStmt는 condition 부분이 Integer value이어야 한다. 만약 Integer가 아니라면 InvalidConditionError로 에러 처리한다.

```
// Return Statement
case ReturnStmt:
{
    // Error Check
    ERROR_CHECK(currentScope->func != NULL);
    // Semantic Error: Invalid Return
    // ********************Fill the Code************************
    TreeNode *function = currentScope->func;
    if (t->child[0]) {
        if (function->type != t->child[0]->type) {
            InvalidReturnError(t->lineno);
            break;
        }
    }
    else {
        if (function->type != Void && function->type != VoidArray) {
            InvalidReturnError(t->lineno);
            break;
        }
    }
```

ReturnStmt는 함수의 반환 타입과 Return value의 타입이 일치하는지 검증해야 한다. 만약 함수가 return하는 값이 있는데, 그 타입을 함수의 반환 타입과 일치하지 않는다면 InvalidReturnError로 에러 처리한다. 만약 return하는 값이 없는데, 함수의 반환 타입이 Void가 아니라면 InvalidReturnError로 에러 처리한다.

```
// Assignment, Binary Operator Expression
case AssignExpr:
case BinOpExpr:
{
    // Error Check
    ERROR_CHECK(t->child[0] != NULL && t->child[1] != NULL);
    // Semantic Error: Invalid Assignment / Operation
    // ********************Fill the Code************************
    TreeNode *lhs = t->child[0];
    TreeNode *rhs = t->child[1];
    if (lhs->type != rhs->type) {
        if (t->kind == AssignExpr) {
            InvalidAssignmentError(t->lineno);
            break;
        }
        if (t->kind == BinOpExpr) {
            InvalidOperationError(t->lineno);
            break;
        }
    }
```

AssignExpr 과 BinOpExpr 은 연산의 좌우 operand 의 타입이 일치해야 한다. 좌측의 operand 를 lhs, 우측의 operand 를 rhs 라고 선언하였다. 만약 둘의 타입이 다르다면 AssignExpr 의 경우는 InvalidAssignmentError 로, BinOpExpr 의 경우는 InvalidOperationError 로 에러처리한다.

```
// Semantic Error: Invalid Arguments
TreeNode *paramNode = calleeSymbol->node->child[0];
TreeNode *argNode = t->child[0];
// ********************Fill the Code*******************
int flag = TRUE;
while (paramNode && paramNode->type != Void) {
    if (!argNode) {
        InvalidFunctionCallError(t->name, t->lineno);
        flag = FALSE;
        break;
    }
    if (paramNode->type != argNode->type) {
        InvalidFunctionCallError(t->name, t->lineno);
        flag = FALSE;
        break;
    }
    paramNode = paramNode->sibling;
    argNode = argNode->sibling;
}

if (flag && argNode) {
    if (argNode->type != Void || argNode->sibling) {
        InvalidFunctionCallError(t->name, t->lineno);
    }
}
```

CallExpr 은 함수의 파라미터와 호출할 때 넘겨주는 argument 의 타입과 개수가 일치해야 한다. 타입을 검증하여 일치하면 sibling 으로 넘어가서 모두 검증을 마칠 때까지 반복한다. 만약 argument 의 수가 다르거나 타입이 다르다면 InvalidFunctionCallError 로 에러처리한다.

```
// Array Access or Not
if (t->child[0] != NULL)
{
    // Semantic Error: Index to Not Array
    // Semantic Error: Index is not Integer in Array Indexing
    // ********************Fill the Code************************
    if (t->type != IntegerArray) {
        ArrayIndexingError2(t->name, t->lineno);
        break;
    }
    TreeNode *index = t->child[0];
    if (index->type != Integer) {
        ArrayIndexingError(t->name, t->lineno);
        break;
    }
    // ************************************************************/
    // Update Node Type
    t->type = Integer;
}
```

VarAccessExpr 은 배열이 아닌 변수에 index 를 주는 경우와 배열의 index 가 Integer 타입이 아닌 경우 두 가지를 검증해야 한다. 만약 두 가지 경우 중에 하나라도 속한다면, ArrayIndexingError 로 에러처리한다.

3. Examples & Results

1) undeclared.1.cm

```
timel2ss@DESKTOP-DF5IA3P:~/compiler/semantic/2019054957/3_Semantic$ cat testcase/01_undeclared/undeclared.1.cm
void main(void)
{
        x;
}
```

```
timel2ss@DESKTOP-DF5IA3P:~/compiler/semantic/2019054957/3_Semantic$ ./cminus_semantic testcase/01_undeclared/undeclared.1.cm

C-MINUS COMPILATION: testcase/01_undeclared/undeclared.1.cm
Error: undeclared variable "x" is used at line 3
```

2) redefined.0.cm

```
timel2ss@DESKTOP-DF5IA3P:~/compiler/semantic/2019054957/3_Semantic$ cat testcase/02_redefined/redefined.0.cm
void func( void )
{
        int x;
        int x;
}
```

```
timel2ss@DESKTOP-DF5IA3P:~/compiler/semantic/2019054957/3_Semantic$ ./cminus_semantic testcase/02_redefined/redefined.0.cm

C-MINUS COMPILATION: testcase/02_redefined/redefined.0.cm
Error: Symbol "x" is redefined at line 4 (already defined at line 3)
```

3) array.2.cm

```
timel2ss@DESKTOP-DF5IA3P:~/compiler/semantic/2019054957/3_Semantic$ cat testcase/03_array/array.2.cm
void func(void) { }

void main(void)
{
        int x[3];

        x[func()];
}
```

```
timel2ss@DESKTOP-DF5IA3P:~/compiler/semantic/2019054957/3_Semantic$ ./cminus_semantic testcase/03_array/array.2.cm

C-MINUS COMPILATION: testcase/03_array/array.2.cm
Error: Invalid array indexing at line 7 (name : "x"). indicies should be integer
```