Project #3. Semantic Analysis Report

2018009234

1. 컴파일 환경

Windows Subsystem for Linux를 이용하여 Ubuntu 20.04.5 LTS에서 작성된 프로젝트입니다. gcc를 이용하여 컴파일합니다. make all 명령어를 통해 실행 파일 cminus_semanitc을 생성합니다.

2. 과제 분석

Project 2에서 구현한 parser로 생성된 AST를 이용하여 Symbol Table을 생성한 뒤 Type Check를 진행합니다. TINY와는 다르게 Symbol Table을 생성할 때에는 scope를 고려해야하며, Type Check도 C-Minus Spec에 맞게끔 수정해야 합니다.

3. Project 2 관련 수정 사항

코드의 가독성과 구현의 용이함을 위하여, 변수에 접근하는 경우를 판단하기 위한 VarAccessK를 추가하였습니다. 이에 따라 cminus.y에서 변수에 접근하는 경우 kind가 VarAccessK 되도로 수정하였습니다. Type Check 시 선언되지 않은 변수를 처리하기 위해 Undeclared라는 타입을 추가하였습니다. 마지막으로 변수와 함수가 어떠한 scope에서 사용되었는 지를 저장하는 멤버인 curScope를 TreeNode 구조체에 추가하였습니다.

4. Symbol Table 구현

1) 전체적인 디자인

Scope를 구현하기 위하여 전체적인 구조부터 구현하였습니다. Scope를 구현하기 위하여 Symbol Table을 여러 개 생성하고 이를 계층 구조로 연결하는 방식을 채택하였습니다. Symbol Table은 기존 코드와 유사하게 Linked List를 이용하였지만 Hash를 계산하는 부분은 생략하였습니다. 각각의 Symbol Table은 하나의 Scope를 의미합니다.

2) 자료 구조

```
struct SymRec_{
  char * name;
  ExpType type;
  int loc;
  // If var, a scope where the var is defined,
  // If func, a scope of func
  SymTable* scope;
  LineNode* lines;

  int paramNum;

  SymRec* next;
};
```

```
struct LineNode_{
  int lineno;
  LineNode* next;
};
```

```
struct SymTable_{
  char* name;
  int loc;
  SymRec* head;
  SymTable* parent;
  SymTable* next; // for using when printing
};
```

```
struct StackNode_{
  SymTable* symTab;
  StackNode* next;
};
```

SymRec 구조체는 Symbol Table에서의 레코드에 대한 정보를 저장하는 구조체입니다. Symbol의 이름, 타입, Location, 사용된 Line 등 기본적인 정보를 저장합니다. Line에 관련된 정보는 기존 코드와 유사하게 LineNode 구조체 이용해 Linked List로 구현하였습니다. paramNum은 함수인 경우 parameter의 개수를, 변수인 경우 -1을 저장하여 함수 호출에 관련된 Type Check, 함수와 변수의 구별 등에 사용합니다. 마지막으로 변수인 경우 define되어 있는 Symbol Table(Scope)을, 함수인 경우 그 함수의 Scope를 저장하고 있습니다.

SymTable 구조체는 Symbol Table(Scope)에 대한 정보를 저장하는 구조체입니다. SymRec Linked List의 head를 통해 레코드에 접근할 수 있게 하였습니다. 변수 접근에서 상위 Scope까지 탐색하기 위하여 계층 구조에서의 부모 Scope의 주소를 저장하고 있습니다. Symbol Table 역시 Linked List로 연결되어 있긴 하지만 이것은 단지 Debugging을 위한 용도입니다.

StackNode 구조체는 Scope 처리를 위해 Symbol Table을 한번 감싸고 있는 Stack입니다. 이에 대한 자세한 설명은 구현에 작성되었습니다.

3) 구현

```
case CompK:
  if(!alreadyCreated)  {
    SymTable* newSymTab = st_build(peekSymStack(), NULL);
    pushSymStack(newSymTab);
  }
  alreadyCreated = FALSE;
```

구현은 크게 Scope 생성과 레코드 삽입으로 나뉩니다. Scope의 생성은 Compound statement, 즉 CompK에서 발생합니다. 새로 생성되는 Scope는 현재의 Scope를 부모 Scope로 가져야 하므로 현재 Scope가 무엇인지를 파악하기 위해 Stack을 이용하였습니다.

traverse()는 AST를 재귀적으로 순회하면서 Preorder와 Postorder로 실행할 수 있는 함수를 정할 수 있습니다. 따라서 Preorder로 CompK Node를 만나면 Symbol Table을 생성하여 Stack에 push하고, Postorder로 CompK Node를 만나면 Stack에서 pop합니다.

이때 FuncK Node를 만난 경우 함수의 Scope를 만든 후 그곳에 Parameter도 넣어야 합니다. 따라서 FuncK Node에서 Scope를 만든 뒤 플래그(alreadyCreated)를 세워 FuncK Node의 child인 CompK Node에서는 Scope를 생성하지 않도록 하였습니다.

레코드 삽입의 경우, AST를 Preorder로 순회하기 때문에 현재 Stack에서 관리 중인 Top Scope에만 넣을 수 있습니다. 따라서 변수의 경우 Top Scope에 삽입을 하고, 함수의 경우 Top Scope에 삽입한 뒤 위에 서술한 바와 같이 자신의 Scope를 만들고 Stack에 push합니다. Parameter의 경우 FuncK Node에서 생성된 Scope가 이미 Top Scope가 되었기 때문에 Variable과 유사한 방식으로 삽입합니다. Parameter는 항상 함수의 Symbol Table(Scope)에 맨 위부터 삽입됩니다.

Symbol Table을 생성하면서 추가적으로 해야만 하는 것은 Type Check 시 어느 Scope부터 확인해야 하는지를 알아야 하기 때문에 변수를 접근하거나, 함수를 호출하는 Scope가 무엇인지를 저장해야 합니다. 따라서 "2. Project 2 관련 수정 사항"에 작성한 것과 같이 TreeNode의 curScope에 Top Scope를 저장합니다. 그리고 변수나 함수가 Undeclared 되어 있다면 TreeNode의 type을 Undeclared로 수정한 뒤 Type Check에서 검출되도록 합니다.

Built-In function인 input과 output은 buildSymTab()에서 하드 코딩으로 구현하였습니다.

5. Type Check 구현

1. 전체적인 디자인

Type Check는 이미 Symbol Table이 정상적으로 생성되었다는 가정 하에 진행합니다. AST를 Postorder로 순회하면서 진행합니다. Postorder로 순회하기 때문에 어떠한 TreeNode를 처리할 때에 이미 자신의 child들의 타입은 모두 확정되었다고 생각할 수 있습니다.

2. Variable / Parameter Declaration

변수와 parameter 선언의 경우 2가지 오류가 발생할 수 있습니다. 첫 번째로 변수가 void로 선언된 경우 오류를 검출합니다. 두 번째로 변수가 Redefine되는 경우인데 이는 Symbol Table을 생성하면서 확인해주었습니다.

3. Undeclared Variable / Function

선언되지 않은 변수와 함수의 경우, Symbol Table을 생성하는 과정에서 TreeNode의 type을 Undeclared로 바꿨기 때문에 이를 확인하여 오류를 검출합니다.

4. Accessing Variable

변수를 사용하고자 하는 경우, 먼저 해당 TreeNode의 child가 존재하는지를 확인합니다. 만약 child가 존재하는 경우에는 indexing을 진행하려고 하는 것입니다. 만약 indexing을 진행하지 않는다면, 변수가 사용되고 있는 Scope부터 Global Scope까지 Symbol Table을 확인하여서 얻은 레코드의 type으로 TreeNode의 type을 변경합니다.

Indexing을 진행하고 있는 경우에는 2가지를 체크해야 합니다. 먼저 위와 같은 방식으로 Symbol Table을 확인하여서 사용하고자 하는 변수가 Integer Array인지 확인합니다. 만약 Integer Array가 맞다면 현재 TreeNode의 type가 Integer인지 확인합니다. 만약 두 경우 모두 아니라면 오류를 검출합니다. 두 경우 모두 문제가 없다면 정상적으로 Integer Array에 접근한 것이므로, 현재 TreeNode의 type을 Integer로 설정합니다.

5. Condition Check

If, If-Else, While의 경우 조건문의 TreeNode의 type이 Integer여야 합니다. 따라서 현재 TreeNode의 첫 번째 child의 type이 Integer가 아닌 경우 오류를 검출합니다.

6. Operation

Operation의 경우, 두 개의 Operand가 모두 Integer인지 확인합니다. 따라서 현재 TreeNode의 첫 번째, 두 번째 child의 type을 확인하고 모두 Integer 라면 현재 TreeNode의 type을 Integer로 수정합니다.

7. Assignment

할당문의 경우, LHS와 RHS의 타입이 일치해야 합니다. 따라서 첫 번째, 두 번째 child의 type이 같은 지를 확인하고 만약 다르다면 오류를 검출합니다.

8. Return Type Check

Return의 경우 현재 Return Statement가 작성되어 있는 함수의 return type과 Return Statement가 하고자 하는 Expression의 type이 일치해야 합니다. 이를 위해 Symbol Table을 생성하면서

ReturnK Node 혹은 ReturnNonK Node를 만나는 경우, 그 Node의 attr.name을 함수의 이름으로 설정합니다. 이후 Type Check를 할 때 함수의 이름을 Global Scope에서 찾은 후 return type이 무엇인지 확인합니다. 이후 ReturnK인 경우 함수의 return type과 현재 TreeNode의 첫 번째 child의 타입이 같은 지 확인하고, ReturnNonK인 경우 함수의 return type이 Void인지 확인합니다.

9. Function Call

함수 호출의 경우 parameter와 argument가 일치하는 지 여부를 확인해야 합니다. 먼저 CallK TreeNode에는 함수의 이름이 저장되어 있으므로, Global Scope에서 함수에 대한 레코드를 찾습니다. 해당 레코드의 멤버 변수를 통해 호출하고자 하는 함수의 Scope와 Parameter의 개수를 알 수 있습니다.

저의 구현에서는 함수의 parameter를 무조건 함수로부터 생성된 Symbol Table에 맨 위부터 순서대로 저장됩니다. 함수의 Parameter의 개수와 Symbol Table을 알기 때문에 Global Scope에서 함수의 대한 레코드를 얻는다면 Parameter에 대한 정보를 알 수 있습니다.

함수의 argument는 CallK Node의 첫 번째 child에 list로 연결되어 있습니다. 먼저 이 list를 순회하면서 argument의 개수를 파악한 뒤, Parameter의 개수와 다르다면 오류를 검출합니다.

만약 argument의 개수와, parameter의 개수가 일치한다면, argument list와 함수의 Symbol Table을 순서대로 순회하면서 argument의 타입과 parameter의 타입이 일치하는지 확인합니다.

모두 문제가 없다면 CallK Node의 type을 함수의 return type으로 설정합니다.

6. 테스트 케이스

```
hgh@DESKTOP-DPO7V4P:~/Compilers/2022_ele4029_2018009234/3_Semantic$ ./cminus_semantic testcase/test_1.cm

C-MINUS COMPILATION: testcase/test_1.cm
hgh@DESKTOP-DPO7V4P:~/Compilers/2022_ele4029_2018009234/3_Semantic$ ./cminus_semantic testcase/test_2.cm

C-MINUS COMPILATION: testcase/test_2.cm
hgh@DESKTOP-DPO7V4P:~/Compilers/2022_ele4029_2018009234/3_Semantic$ ./cminus_semantic testcase/test_3.cm

C-MINUS COMPILATION: testcase/test_3.cm
Error: Invalid function call at line 12 (name : "x")
hgh@DESKTOP-DPO7V4P:~/Compilers/2022_ele4029_2018009234/3_Semantic$ ./cminus_semantic testcase/test_4.cm

C-MINUS COMPILATION: testcase/test_4.cm
Error: Invalid array indexing at line 4 (name : "x"). Indices should be integer
```

제공된 테스트 케이스에 대한 결과입니다.