ELE4029 Project Report

3. Semantic

박준영

1. How to build

1.1 Compilation Environment

본 프로젝트는 다음의 환경에서 개발, 테스트하였다.

- Ubuntu Server 20.04.3 LTS
- gcc version 9.3.0
- GNU Make 4.2.1
- flex 2.6.4
- GNU bison 3.5.1

1.2 Compilation Method

우선 아래 명령어를 통해 hconnect에서 repository를 clone한다.

```
git clone https://hconnect.hanyang.ac.kr/2021_ele4029_11216/2021_ele4029_2019064811
```

이후 다음 명령어들을 수행하여 컴파일한다.

```
cd 2021_ele4029_2019064811/3_Semantic
make
```

컴파일이 완료되면 cminus_semantic 실행파일이 생성된다. 프로그램의 실행은 다음과 같은 방식으로 한다.

```
./cminus_semantic <filiename>
```

2. Implementation

1) Symbol Table

본 프로젝트의 목표인 Semantic Analyzer를 구현하기 위해선 parser로부터 생성된 Abstract Syntax Tree(이하 AST)에 등장하는 symbol에 대한 정보를 저장할 필요가 있다. 이러한 symbol의 정보(타입, 이름, 스코프 등)를 symbol table에 저장한다. 그러기 위하여 symbol table의 자료 구조를 만들었다.

기존의 tiny의 symbol table은 global scope만 지원을 하고, 다른 scope를 만드는 것을 지원하지 않는다. 따라서 우선적으로 여러 scope를 가질 수 있도록 아래의 구조체를 이용하여 scope에 대한 자료구조를 만들었다.

```
typedef struct ScopeListRec
   { char * name;
     BucketList bucket[HASH_TBL_SIZE];
     struct ScopeListRec * parent;
     struct ScopeListRec * leftMostChild;
     struct ScopeListRec * rightSibling;
   } * ScopeList;
```

(symtab.h)

위에서 확인할 수 있듯, scope는 tree structure로 구현하였다. 그 이유는 scope 자체가 계층 구조를 이루기 때문이다. 이후 어떤 symbol이 유효한 symbol인지 확인하는 과정에서 상위 scope 방향으로 확인할 필요가 있기에 parent node의 pointer를 담았다. Tree의 같은 level에 여러 scope가 존재할 수 있으므로 이를 표현하기 위하여 우측 sibling node의 pointer도 담았다. 그리고, 해당 scope에 존재하는 symbol을 bucket에 담도록 하였다. 기존 tiny에서는 global scope를 *hashTable* 이란 전역변수로 표현했지만, 본 프로젝트에선 위 구조체를 활용해 *globalScope*란 전역변수로 global scope를 표현하였다.

Symbol의 표현은 *BucketListRec* 구조체를 이용하였다. tiny에 구현된 *BucketListRec* 구조체의 경우, 함수를 제대로 표현하기엔 한계가 있기 때문에 아래와 같이 추가적으로 함수에 대한 정보를 담을 수 있도록 하였다.

```
typedef struct BucketListRec
   { ...
     struct {
       ExpType type;
       int params;
       struct {
        char * name;
        ExpType type;
       } param[MAX_FUNC_PARAMS];
     } func;
     struct ScopeListRec * scope;
   } * BucketList;
```

(symtab.h)

Semantic analyzer에서 함수에 대해 필요한 정보는 반환 타입과 함수 파라미터 정보이다. 위 구조체에서 func.type은 함수 반환타입을, func.param은 함수 파라미터의 이름과 타입을 담는다. 또한, bucket에서 scope를 찾기 수월하게 하기 위해 해당 bucket이 속한 scope의 pointer를 담도록 하였다. 위에서 생략된 부분(...)은 기존 tiny의 코드와 같다.

마지막으로, symbol table의 type을 넣을 때 더 자세하게 하기 위하여(function과 variable을 구분 등) ExpType에 Function과 ErrorExp를 추가하였다. (ErrorExp는 이후에 서술)

2) Symbol Table Building

이 과정은 parser에서 만들어진 AST를 traverse하면서 선언되거나 사용된 symbol을 검출하여 symbol table을 만든다. 이 과정은 analyze.c의 *buildSymtab* 함수를 통해 수행된다.

Tiny에서 symbol table에 대한 insert, lookup 연산은 각각 *st_insert*, *st_lookup* 함수로 구현이 되어있다. 하지만 cminus의 scope 지원을 위해 각각 함수 인자를 아래와 같이 수정하였다.

```
BucketList st_insert( ScopeList scope, char * name, ExpType type,
                      int lineno, int loc )
BucketList st_lookup ( ScopeList scope, char * name )
```

st_insert 함수는 기존의 tiny에서 bucket에 scope 정보를 추가하는 루틴만 추가되었다. *st_lookup* 함수의 경우엔 현재 scope에 찾아서 없으면 부모 scope 방향으로 찾아 나가는 루틴을 추가하였다. global scope까지 스캔하여도 찾지 못했다면 그때 NULL을 반환하여 해당 symbol이 없음을 알린다.

우선적으로 global scope를 초기화 하는 symbol table 초기화가 수행된다. 이 과정은 symtab.c의 *init_symtab* 함수에서 수행된다. 자세하게 서술하자면 *globalScope* 전역변수를 초기화 하고, global scope에 cminus의 built-in function인 *input*과 *output* 함수의 symbol 정보를 넣는 작업을 한다.

이후에는 AST를 traverse하며 node의 종류에 따라 아래의 규칙대로 symbol table을 구축한다. 이 과정을 수행하면서 현재 scope 정보는 *currentScope* 전역변수를 이용해 저장한다. 기본값은 *globalScope*로 초기화한다.

<pre-order traversal> - *insertNode* 함수에서 수행 (in analyze.c)

- ParamK 또는 VarDeclK node
  : 같은 scope내에 같은 이름의 symbol이 존재하는지 체크하여 존재하면 오류를 출력하고, 없다면 symbol table에 넣는다.
- FuncDeclK node
  : 같은 scope내에 같은 이름의 symbol이 존재하는지 체크하여 존재하면 오류를 출력하고, 없다면 함수의 파라미터 및 반환타입 정보를 bucket의 func에 넣고 symbol table에 추가한다.

또한 함수 이름으로 된 scope를 만들어 *currentScope*를 새 scope로 교체한다.

- CompoundK node
  : Compound는 기본적으로 scope를 새로 생성하는 statement이다. 그런데 function declaration의 경우 compound statement를 동반하기 때문에 scope가 중복으로 생기는 문제가 있다. 따라서 FuncDeclK를 처리할 때 function이 선언되었다는 flag를 on하여 flag가 on일 때는 새 scope를 만들지 않는 방식으로 이러한 문제를 방지하였다. 그리고 flag를 off하여 그 이후부터는 CompoundK node를 만날 때마다 새로운 scope를 생성하도록 하였다.

- VarAccessK 또는 CallK node
  : 이 두 경우엔 symbol이 선언되었는지 여부를 판단하고 없다면 오류를 출력, 있다면 symbol table에 해당 symbol이 사용된 위치 정보를 추가한다.

<post-order traversal> - *afterInsertNode* 함수에서 수행 (in analyze.c)

- CompoundK node
  : Compound node의 처리가 끝났다면 현재 scope를 벗어나야 함을 의미한다. 따라서 *currentScope*를 현재 scope의 부모 scope로 변경한다.

위에 별도로 기술하지 않은 타입의 node는 처리하지 않고 건너뛰었다.

3) Type Checking

이 과정은 위 과정을 통해 구축된 symbol table을 이용하여 AST를 traverse하며 실질적인 expression의 type을 확인하고 여러 semantic error를 찾는다.

<pre-order traversal> - *beforeCheckNode* 함수에서 수행 (in analyze.c)

- FuncDeclK node
  : 현재 scope의 자식 scope 중에서 함수 이름으로 된 scope를 찾아 *currentScope*를 교체한다.

- CompoundK node
  : 현재 scope의 자식 scope 중에서 해당하는 scope를 찾아 *currentScope*를 교체한다. 하지만, 위와 같은 이슈가 있을 수 있기 때문에 이 과정에서도 FuncDeclK를 처리할 때 function이 선언되었다는 flag를 on하여 flag가 on일 때는 scope를 교체하지 않는 방식으로 문제를 해결하였다. 그리고 flag를 off하여 그 이후부터는 CompoundK node를 만날 때마다 scope를 바꾸도록 하였다.

<post-order traversal> - *checkNode* 함수에서 수행 (in analyze.c)

- ParamK 또는 VarDeclK node
  : 타입이 Void이거나 VoidArr인 경우 오류가 나도록 하였다.

- CompoundK node
  : Compound node의 처리가 끝났다면 현재 scope를 벗어나야 함을 의미한다. 따라서 *currentScope*를 현재 scope의 부모 scope로 변경한다.

- IfK, IfElseK 또는 WhileK node
  : 조건(child[0])의 타입이 Integer인지 체크하여 그렇지 않다면 오류가 나도록 하였다.

- ReturnK node
  : 함수는 global scope에서만 선언될 수 있고, 또 함수의 scope는 함수의 이름과 같은 점을 이용하여 함수의 bucket 정보를 우선 얻었다. 그 후 return하는 자료의 타입과 함수 반환 타입이 올바른지를 체크하여 올바르지 않다면 오류가 나도록 하였다.

- OpK node
  : 우선 연산할 두 operand가 ErrorExp면(오류가 존재하면) OpK node의 타입도 ErrorExp로 하여 계속 오류라 표기하도록 하였다. 그 후 두 operand가 모두 Integer인지 체크하여 그렇다면 OpK node의 type을 Integer로, 아니라면 오류를 출력하였다.

- ConstK node
  : cminus에서 constant는 모두 Integer이므로 ConstK node의 type을 Integer로 하였다.

- AssignK node
  : Assignment operator의 lhs 또는 rhs가 ErrorExp이면 AssignK node의 type도 ErrorExp로 하였다. 그 후 lhs가 lvalue가 아니라면 오류가 출력하게 하였고, lhs와 rhs가 타입이 같지 않아도 오류가 뜨도록 하였다. 문제가 없다면 AssignK node의 type을 lhs의 type으로 하였다.

- VarAccessK node
  : 배열이라면 배열의 index가 올바른지 체크하는 등의 처리를 하고 문법적으로 올바르지 않다면 해당 node의 type을 ErrorExp로 하였다.

- CallK node
  : 우선적으로 함수의 인자 개수를 체크하였다. 만약 인자 개수가 맞다면 함수의 인자 타입들이 모두 올바른지 체크한다. 문제가 없다면 해당 node의 type을 함수 반환형으로, 문제가 있다면 ErrorExp로 하였다.

위에 별도로 기술하지 않은 타입의 node는 처리하지 않고 건너뛰었다. 만약 node가 ErrorExp라면 이미 오류가 발생한 것을 의미하므로, 더 이상 오류를 출력하지 않고 처리하지 않도록 하였다.

4) Error Producing

오류는 gcc에서 C를 이용해 오류를 발생시켜 나온 메시지를 출력하도록 하였다. 기존의 tiny 에러 출력 함수는 자유도가 떨어지는 문제가 있다. 그렇기에 format string을 지원하는 새로운 오류 출력 함수를 다음과 같이 만들었다.

```
static void semanticError(TreeNode* t, const char* message, ...)
```

(analyze.c)

아래는 사용 예시다.

```
semanticError(t, "redefined function '%s'", t->attr.name);
```

3. Screenshots

<Test Source>

```
/* A program to perform Euclid's
   Algorithm to compute gcd */

int gcd (int u, int v)
{
    if (v == 0) return u;
    else return gcd(v,u-u/v*v);
    /* u-u/v*v == u mod v */
}
int gcd(int x) { return x; }

void main(void)
{
    int x; int y;
    x = input(); y = input();
    output(gcd(x,y));
    z = input();
}
```

<Output>

```
(base) jun@deepjun:~/Univ/ELE4029/3_Semantic$ ./cminus_semantic testcase/test.0.txt

C-MINUS COMPILATION: testcase/test.0.txt
Semantic Error: redefined function 'gcd' at line 10
Semantic Error: undefined identifier 'z' at line 17
```