Compiler

project 3. Semantic

2019027192 김현수

1. ScopeList 구현

```
typedef struct FunctionArgsListRec
{
    char* name; // null possible
    ExpType type;
    int isarray;
    struct FunctionArgsListRec* next;
}* FunctionArgsList;

typedef struct
{
    int args_count;
    FunctionArgsList args;
} FunctionInfo;

typedef struct BucketListRec
{
    char* name;
    ExpType type;
    int isarray;
    SymbolKind kind;
    LineList lines;
    int memloc;
    struct BucketListRec* next;
    FunctionInfo functionInfo
}* BucketList;


typedef struct ScopeListRec
{
    char* name;
    BucketList bucket[SIZE];
    struct ScopeListRec* parent;
    struct ScopeListRec* leftmost;
    struct ScopeListRec* sibling;
}* ScopeList;
```

scope 계층 트리를 생성하기 위해 ScopeList 구조체를 생성했습니다. 해당 구조체는 scope의 이름인 name, scope에 속한 심볼들이 저장된 bucket, 트리 구조를 위한 parent/leftmost/sibling으로 이루어져 있습니다.

그 다음 BucketList 구조체를 조금 수정했습니다. 먼저, Variable과 Function을 구분하기 위해 아래와 같은 SymbolKind Enum을 추가했습니다.

```
typedef enum
{
    FuncSymbol,
    VarSymbol
} SymbolKind;
```

그 다음, function의 parameter 저장해 checkNode에서 argument와 type을 확인하기 위해, FunctionArgsList라는 구조체를 추가했습니다. 그리고 이걸 BucketList에 넣어, SymbolKind가 FuncSymbol인 심볼들에 functionInfo로 가지고 있을 수 있도록 했습니다.

2. st_lookup, st_lookup_excluding_parent 구현

```
BucketList st_lookup(ScopeList scope, char* name)
{
    while (scope)
    {
        int h = hash(name);
        BucketList l = scope->bucket[h];
        while ((l != NULL) && (strcmp(name, l->name) != 0))
            l = l->next;
        if (l)
        {
            return l;
        }
        scope = scope->parent;
    }
    return NULL;
}

BucketList st_lookup_excluding_parent(ScopeList scope, char* name)
{
    int h = hash(name);
    BucketList l = scope->bucket[h];
    while ((l != NULL) && (strcmp(name, l->name) != 0))
        l = l->next;
    if (l)
    {
        return l;
    }

    return NULL;
}
```

기존의 st_lookup을 기반으로 st_lookup과 st_lookup_excluding_parent를 구현했습니다. st_lookup은 scope list tree를 거슬러 올라가며 심볼을 찾고, st_lookup_excluding_parent는 현재 scope에서만 찾습니다. 이렇게 두 버전의 함수가 필요한 이유는 다음 문단에서 설명하도록 하겠습니다.

3. st_insert, st_insert_lineno 구현

```
/* Success: return BucketList, Failure(redefine): return NULL */
BucketList st_insert(ScopeList scope, char* name, ExpType type, int isarray,
SymbolKind kind, int lineno, int loc)
{
    int h = hash(name);
    BucketList l = st_lookup_excluding_parent(scope, name);
    if (l)
    {
        return NULL;
    }

    l = (BucketList)malloc(sizeof(struct BucketListRec));
    l->name = name;
    l->lines = (LineList)malloc(sizeof(struct LineListRec));
    l->lines->lineno = lineno;
    l->memloc = loc;
    l->type = type;
    l->kind = kind;
    l->isarray = isarray;
    l->lines->next = NULL;
    l->next = scope->bucket[h];
    l->functionInfo.args = NULL;
    scope->bucket[h] = l;
    return l;
} /* st_insert */

/* Success: return 0, Failure(undefined): return -1 */
int st_insert_lineno(ScopeList scope, char* name, int lineno)
{
    int h = hash(name);
    BucketList l = st_lookup(scope, name);
    if (!l)
    {
        return -1;
    }

    LineList t = l->lines;
    while (t->next != NULL)
        t = t->next;
    t->next = (LineList)malloc(sizeof(struct LineListRec));
    t->next->lineno = lineno;
    t->next->next = NULL;
    return 0;
}
```

st_insert는 심볼 테이블에 새로운 심볼을 추가할 때 사용되는 함수입니다. 새로운 심볼을 추가할 때는 현재 scope에 해당 심볼이 있는지만 체크해야 하므로, st_lookup_excluding_parent를 사용했습니다.

st_insert_lineno는 심볼이 사용될 시 lineno를 추가해 주는 함수입니다. 이렇게 상위 scope까지 확인해 심볼을 찾는 경우에는 st_lookup을 사용했습니다.

4. 함수 관련 정보

```
typedef struct FunctionArgsListRec
{
    char* name; // null possible
    ExpType type;
    int isarray;
    struct FunctionArgsListRec* next;
}* FunctionArgsList;

typedef struct
{
    int args_count;
    FunctionArgsList args;
} FunctionInfo;
```

함수 관련 정보는 위와 같은 FunctionInfo 구조체에 저장됩니다. FunctionInfo 구조체는 파라미터의 개수인 args_count와, 파라미터들의 linked list인 args를 가집니다. 여기에는 각 파라미터들의 이름과 타입이 저장됩니다.

```
void addFuncArg(BucketList func, TreeNode* param)
{
    FunctionArgsList* arg = &func->functionInfo.args;
    while (*arg)
    {
        arg = &(*arg)->next;
    }
    *arg = (FunctionArgsList)malloc(sizeof(struct FunctionArgsListRec));
    memset(*arg, 0, sizeof(struct FunctionArgsListRec));
    (*arg)->isarray = param->isarray;
    (*arg)->name = param->attr.name;
    (*arg)->type = param->type;
    ++func->functionInfo.args_count;
}
```

addFuncArg 함수를 이용해, AST에 있는 FunctionK 노드를 파싱하고 심볼 노드에 함수 관련 정보를 추가할 수 있습니다.

5. built_in_functions

```
static int built_in_functions(ScopeList scope, int location)
{
    // output
    {
        BucketList output = st_insert(scope, "output", Void, FALSE, FuncSymbol, 0,
location++);
        TreeNode param;
        param.isarray = FALSE;
        param.attr.name = "";
        param.type = Integer;
        addFuncArg(output, &param);
    }

    // input
    {
        st_insert(scope, "input", Integer, FALSE, FuncSymbol, 0, location++);
    }
    return location;
}
```

output과 input과 같이 과제 명세에서 build_in 함수로 제공하라 했던 함수들을 scope에 추가해주는 함수입니다.

output의 경우 addFuncArg로 int 타입 파라미터가 있음을 심볼 테이블에 추가해줍니다.

위 과정들을 마치고 과제 명세와 같이 심볼 테이블을 출력해 보면, test.1.txt 기준으로 아래와 같이 정상적으로 구성되었음을 알 수 있습니다.

```
< Symbol Table >
Variable Name  Variable Type  Scope Name  Location   Line Numbers
-------------  -------------  ----------  --------   ------------
main           Function       global      4          11
input          Function       global      1          0   14   14
output         Function       global      0          0   15
v              Integer        global      2          3
gcd            Function       global      3          4    7   15
u              Integer        gcd         0          4    6    7    7
v              Integer        gcd         1          4    6    7    7    7
x              Integer        main        0          13   14   15
y              Integer        main        1          13   14   15

< Function Table >
Function Name  Scope Name  Return Type  Parameter Name   Parameter Type
-------------  ----------  -----------  --------------   --------------
main           global      Void                          Void
input          global      Integer                       Void
output         global      Void
                                                         Integer
gcd            global      Integer
                                        u                Integer
                                        v                Integer
```

6. insertNode 구현

insertNode에서 고려해야 할 Kind들은 다음과 같습니다

AssignmentK: assignment는 심볼과 관련이 없습니다

OperatorK: operator는 심볼과 관련이 없습니다

ConstantK: Constant는 심볼과 관련이 없습니다

CallK: Call은 symbol을 사용하므로, lineno 추가가 필요합니다 (st_insert_lineno)

VarK: Var은 symbol을 사용하므로, lineno 추가가 필요합니다 (st_insert_lineno)

CompoundK: Compound는 함수 안에서 등장할 경우, 익명 scope를 새롭게 생성하기 때문에, CompoundK가 등장할 시 현재 함수 안인지 확인(is_func_compound)하고, 함수 안쪽이라면 새로운 scope를 추가해줍니다.

SelectionK: Selection은 심볼과 관련이 없습니다

IterationK: Iteration은 심볼과 관련이 없습니다

ReturnK: Return은 심볼과 관련이 없습니다

FuncK: FunK는 Declaration이므로, 심볼의 추가 (st_insert)가 필요합니다. 만약 이미 현재 scope에 존재한다면, redeclaredError를 발생시킵니다.

VarDeclarationK: VarK 역시 Declaration이므로, st_insert가 필요하고, 이미 존재한다면 redeclaredError를 발생시킵니다

ParameterK: ParameterK 역시 Var와 동일하게 Declaration이므로, st_insert가 필요하고, 이미 존재한다면 redeclaredError를 발생시킵니다. 추가로 addFuncArg도 필요합니다

VoidParameterK: Void Parameter는 심볼과 관련이 없습니다.

위 조건들에 따라 insertNode 함수를 구현했습니다.

semantic error 중 redeclaredError는 그 특성상 insertNode에서도 충분히 검사가 가능하므로, insertNode에서 검사하도록 했습니다.

관련이 없는 Kind들을 제외한 Kind (Call, Var 등)는 redeclaredError를 검사함과 동시에 symbol table에 insert 하고, type 정보를 treeNode에 반영하도록 했습니다.

7. scope stack 구현

insertNode 와 checkNode 에서, 현재 scope 가 어디인지 나타낼 방법이 필요합니다

이는 stack 으로 구현하는 것이 편하므로, scope stack 을 구현했습니다

```
typedef struct
{
    ScopeList scope;
    int location;
} ScopeStackPair;

static ScopeStackPair scope_stack[MAXSCOPEDEPTH];
static int scope_stack_top_index = -1;
int scope_stack_push(ScopeList scope, int location)
{
    if (scope_stack_top_index >= MAXSCOPEDEPTH - 1)
    {
        return -1;
    }
    scope_stack[++scope_stack_top_index].scope = scope;
    scope_stack[scope_stack_top_index].location = location;
    return 0;
}

ScopeStackPair* scope_stack_top()
{
    return &scope_stack[scope_stack_top_index];
}

int scope_stack_pop()
{
    if (scope_stack_top_index < 0)
    {
        return -1;
    }
    scope_stack[scope_stack_top_index].scope = NULL;
    scope_stack[scope_stack_top_index--].location = 0;
    return 0;
}
```

위와 같이 scope 와 location 을 합한 pair 구조체를 만들고, stack 을 구현했습니다.

```
void buildSymtab(TreeNode* syntaxTree)
{
    ScopeList global_scope = init_global_scope();
    traverse(syntaxTree, insertNode, afterInsertNode);
    if (TraceAnalyze)
    {
        printSymTab(listing, global_scope);
        printFuncTab(listing, global_scope);
    }
}
```

```
void typeCheck(TreeNode* syntaxTree)
{
    traverse(syntaxTree, beforeCheckNode, checkNode);
}
```

scope 에 들어오면 해당 scope 를 push 하고, scope 를 나가면 해당 scope 를 pop 해야 합니다.

구조상 push 는 preorder, pop 은 postorder 로 이뤄지는게 맞으므로, afterInsertNode 와 beforeCheckNode 를 추가했습니다.

```
            case CompoundK:
                if (!is_func_compound)
                {
                    char buf[101];
                    snprintf(buf, 100, "%s_%d", pair->scope->name, t->lineno);
                    ScopeList newScope =
                        create_ScopeList(pair->scope, copyString(buf));
                    scope_stack_push(newScope, 0);
                    t->scope = newScope;
                    pair->location++;
                }
                is_func_compound = FALSE;
                break;
```

```
// DeclarationK
        ScopeList newScope =
                    create_ScopeList(pair->scope, t->attr.name);
                scope_stack_push(newScope, 0);
                t->scope = newScope;
                is_func_compound = TRUE;
```

위와 같이 insertNode 에서 함수에 들어가거나, 함수 안쪽의 임시 scope 에 들어갈 시 scope stack 에 push 합니다.

임시 scope 의 이름은 (부모 scope 의 이름)_(lineno)로 했습니다.

```
static void afterInsertNode(TreeNode* t)
{
    if (t->nodekind == StmtK && t->kind.stmt == CompoundK)
    {
        scope_stack_pop();
    }
}
```

node 가 Compound 일 경우, scope 를 벗어난 것 이므로 scope stack 에서 pop 을 한 번 해주면 됩니다.

CheckNode 역시, preorder 인 beforeCheckNode 에서 현재 노드가 CompoundK 와 FuncK 일 시 treeNode 에 저장해 놓은 scope 를 이용해 복구하고, postorder 인 checkNode 에서는 pop 하도록 구현했습니다.

8. checkNode 구현

expression의 타입은 아래 단계를 통해 결정됩니다

1. type: 타입이 무엇인가? (void, int)
2. isarray: 배열인가?
3. redeclaredError와 같은 문제로 인해 타입이 결정되지 않았는가?

checkNode는 이 디자인을 고려하여 작성되었습니다.

AssignmentK: lhs와 rhs의 타입은 동일해야 하고, int 여야 합니다. 그리고 노드의 타입을 결정합니다.

OperatorK: operand의 타입은 동일해야 하고, int여야 합니다 그리고 노드의, 타입을 결정합니다.

ConstantK: 해당 노드의 타입은 int 입니다.

CallK: 해당 심볼이 존재하는지, arguments의 개수가 함수의 parameter 개수와 동일한지, 각 parameter에 맞는 타입의 argument를 주었는지 등을 확인합니다

VarK: 해당 심볼이 존재하는지 확인하고, array라면 index가 정상적인지 확인합니다. 그리고 현재 노드의 타입을 결정합니다

CompoundK: scope_stack을 pop 합니다. (postorder)

SelectionK: 조건문의 타입이 int 인지 검사합니다.

IterationK: 조건문의 타입이 int 인지 검사합니다.

ReturnK: return 문의 타입이 함수의 타입과 동일한지 검사합니다.

FuncK: 함수의 선언은 type checking이 필요하지 않습니다 (int[] main 과 같은 array return은 lexical error가 발생하므로 고려할 필요가 없음)

VarDeclarationK: 변수의 타입은 int, int array여야만 합니다

ParameterK: 변수의 타입은 int, int array여야만 합니다

VoidParameterK: Void Parameter 선언은 type checking이 필요하지 않습니다

9. 테스트 진행

테스트 프로그램 (sematic_test.py)를 작성해 테스트를 진행했습니다.

test_case_s 폴더에 있는 테스트 파일은 모두 정상적으로 analyze 되어야 하고, test_case_e 폴더에 있는 테스트 파일은 모두 semantic error가 출력되어야 합니다. 이 때 error at line (숫자) 부분을 파싱해 테스트 파일에 주석으로 입력된 정답을 비교해, semantic error가 발생한 코드 라인까지 테스트했습니다.

테스트 성공: 00_함수리턴타입추론.txt

테스트 성공: 01_배열참조타입추론.txt

테스트 성공: 02_비교연산타입추론.txt

테스트 성공: 03_파라미터10개함수.txt

테스트 성공: 04_배열타입파라미터.txt

테스트 성공: 05_같은스코프.txt

테스트 성공: 06_전역변수.txt

테스트 성공: 07_상위스코프.txt

테스트 성공: 00_void에서int반환.txt

Type error at line 3: Function of type 'void' cannot return a value.

테스트 성공: 01_int에서반환값없음.txt

Type error at line 3: The return statement of int type function must contain a value.

테스트 성공: 02_void에int인자주고호출 copy.txt

function call error at line 7: The a function has 0 parameters, but only 1 entered.

테스트 성공: 03_인자하나부족.txt

function call error at line 7: The a function has 2 parameters, but only 1 entered.

테스트 성공: 04_인자하나많음.txt

function call error at line 7: The a function has 1 parameters, but only 2 entered.

테스트 성공: 05_파라미터타입이void copy.txt

Type error at line 1: Variable type must be integer or integer array

테스트 성공: 06_변수타입이void.txt

Type error at line 3: Variable type must be integer or integer array

테스트 성공: 07_if에배열.txt

Type error at line 4: The type of if condition can only be integer.

테스트 성공: 08_while에배열.txt

Type error at line 4: The type of loop condition can only be integer.

테스트 성공: 09_배열끼리연산.txt

Type error at line 5: operations between array names are not possible.

테스트 성공: 10_배열index에배열.txt

Type error at line 5: The index of the array must be integer.

테스트 성공: 11_배열index에void.txt

Type error at line 10: The index of the array must be integer.

테스트 성공: 12_void타입배열.txt

Type error at line 3: Variable type must be integer or integer array

테스트 성공: 13_파라미터타입이void배열.txt

Type error at line 1: Variable type must be integer or integer array

테스트 성공: 14_선언안됨.txt

Undeclared error at line 11: 'z' undeclared

테스트 성공: 15_스코프겹침.txt

Redeclared error at line 4: 'a' redeclared

테스트 성공: 16_하위스코프.txt

Undeclared error at line 6: 'y' undeclared

테스트 성공: 17_선언안된함수사용.txt

Undeclared error at line 3: 'a' undeclared

테스트 성공: 18_while조건빔.txt

Syntax error at line 3: syntax error

테스트 성공: 19_if조건빔.txt

Syntax error at line 3: syntax error

테스트 성공: 20_line_undeclared리턴.txt

Undeclared error at line 5: 'c' undeclared

테스트 성공: 21_line_while조건.txt

Undeclared error at line 6: 'c' undeclared

테스트 성공: 22_line_if조건.txt

Undeclared error at line 6: 'c' undeclared

테스트 성공: 23_var선언.txt

Redeclared error at line 9: 'a' redeclared

고려된 테스트 케이스 모두에서 적절한 결과가 출력되는 모습을 확인할 수 있었습니다.