2022056262_project3

Goal / compilation environment and method

project 1, 2에서 만든 scanner, parser를 기반으로 semantic analyzer 구현

컴파일 환경 및 방법

- Ubuntu 20.04.4 LTS
- 첨부되어 있던 Makefile을 사용 gcc로 컴파일, Makefile로 빌드

parser로 만들어진 syntax tree를 buildSymtab 함수에 넣어서 호출한다. 저 함수에서 traverse 함수를 호출해서 tree를 탐색한다. traverse 함수를 호출할 때 함수 포인터로 받기 때문에 insertNode, nullProc을 넣어서 보낸다. preProc으로 계속 preorder로 탐색하면서 insertNode 함수를 호출하고, 다시 돌아올 땐 nullProc으로 그냥 돌아온다. 그러나 우리는 scope를 기반으로 탐색해야 하기 때문에 exitScope 역할을 해줄 함수를 넣어주어야 한다. insertNode에서 경우에 따라 새로운 scope를 만들기도 하고 재정의, 정의되지 않은 변수 확인도 하고 table에 변수를 추가하는 역할을 한다.
이 때 재정의, 정의되지 않은 변수 확인을 symbol table을 기반으로 수행한다.
symbol table이 다 만들어지면 AST를 순회하면서 symbol table과 node를 기반으로 type을 check한다. scope 기반으로 탐색해야 하기 때문에 preOrder에서 현재 scope를 설정하고 postOrder에서 검사한다.

pre-processing

ppt를 토대로 구조체를 정의하였다. scopelist로 scope들을 관리, 각 scope마다 해당 scope에 정의된 변수들을 관리한다. 변수 중에서도 함수의 경우엔 매개변수들을 관리해주어야 하기 때문에 bucket list에 parameterList를 추가한다.

구조체, parsing 방식 수정

```
typedef struct BucketListRec
   { char * name;
     LineList lines;
     int memloc ; /* memory locat
     struct BucketListRec * next;
     char *type;
     char *kind;
     ParameterList params;
   } * BucketList;
```

```
typedef struct ScopeListRec {
   int childCounter;
     int location;
     char *name;
     int nestedLevel;
     struct ScopeListRec *parent;
     BucketList hashTable[SIZE];
     ParameterList paramlist[SIZE];
     ParameterList *currentParamList;
} *Scope;
```

```
typedef struct LineListRec {
     int lineno;
     struct LineListRec *next;
} *LineList;
```

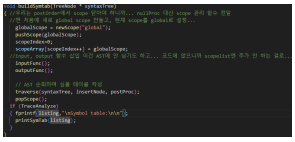
```
iter_stmt : WHILE LPAREN expression RPAREN statement
                { $$ = newStmtNode(whileK,$5->lineno);
                  $$->child[0] = $3;
                  $$->child[1] = $5;
                  $$->lineno=$5->lineno;
                }
          ;
```

```
/* Structure for parameter list */
typedef struct ParamListRec {
     char *name;
     char *type;
     struct ParamListRec *next;
} *ParameterList;
```

lineno를 ppt에 나온 대로 설정해주기 위해 cminus.y 파일도 수정하였다.
그냥 lineno를 원하는 요소의 lineno로 지정해주면 된다.

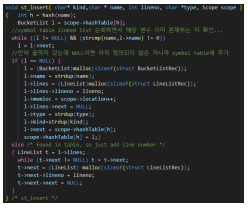
특이 사항으로 scopeListRec의 childCounter는 중첩 scope의 경우
상위 scope의 이름_자식 번호
ex) main_1 이런 식으로 짓기 위해 추가 currentParamList는 AST를 순회하면서 parameter를 받을 때 해당 함수에 한 번에 연결해주기 위한 임시 장소.

symbol table 만들기 전 초기 세팅 + symbol table 추가하는 법 및 scope 관리



초기에 global scope를 추가하고 input, output function을 넣음.
new Scope은 새로운 scope을 만듦. pushScope은 scope linked list에 새로운 scope를 추가하는 역할. scope index와 scopeArray는 추후 semantic 검사를 편하게 하기 위해 설정한 값들. scopeArray에 모든 scope를 저장. scope형 배열이라 단순하게 scope를 탐색하기 쉽다는 장점이 있음.

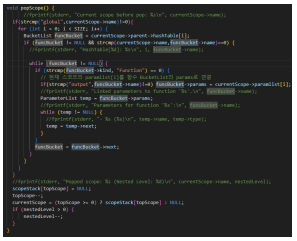
input은 scope도 없고 인자도 없어서 그냥 symbol table에 추가.
output은 parameter가 있어서 parameter를 추가해서 symbol table에 넣고, scope도 존재하기 때문에 새로운 scope를 만들어서 scope stack에 추가. output 함수가 실제로 코드에 존재하는 건 아니라서 semantic 검사할 때는 필요가 없을 거라 판단하여 scopeArray에는 추가하지 않음. symbol table에 인자 추가하고 scope를 탈출.



```
void pushScope(Scope scope) {
    if (currentScope != NULL) {
        scope->parent = currentScope;
    }
    scope->currentParamList = NULL;
    scope->nestedLevel = nestedLevel;
    scopeStack[++topScope] = scope;
    currentScope = scope;
   // fprintf(stderr, "Pushed scope: %
}
```

인자로 받은 scope에서 관리하는 변수들을 순회하면서 인자로 받은 값이 있는 지 확인. 존재한다면 line 번호만 추가하고 없다면 symbol table에 추가. 그냥 해당 scope에서 hash 함수에 의해 만들어진 index에 방금 만든 bucket를 추가하면 됨. location은 해당 scope에서 몇 번째로 선언된 변수인 지를 의미. 이건 scope 구조체의 멤버 변수.
function에 추가하는 방식도 이와 동일하다. 별도로 parameter list에 인자로 받은 param만 추가하면 됨.

function에 추가하는 param list는 addParameter 함수를 이용해서 linked list 형식으로 저장



push scope은 새로운 scope에 들어갔을 때 중첩 수준이나 임시 parameter등을 초기화해주기 위한 함수이다. 중첩 상태라면 새로운 scope의 부모를 이전 scope로 설정해주어야 한다.
pop scope는 현재 scope, 중첩 수준등을 수정해주고 임시 저장 param list를 연결해주는 역할을 수행.

symbol table에서 찾는 행위

```
BucketList st_lookup_bucket(char *name, char * kind) {
    Scope scope = currentScope;
    while (scope != NULL) {
        int h = hash(name);
        BucketList l = scope->hashTable[h];
        while (l != NULL) {

            if (strcmp(name, l->name) == 0 && strcmp(kind,l->kind)==0) {

                return l;
            }
            l = l->next;
        }
        scope = scope->parent;
    }
    return NULL;
}
```

현재 scope부터 global scope까지 돌면서 이름과 타입이 동일한 bucket이 있는 지 검사

```
BucketList st_lookup_top(char *name, char * kind){
   int h = hash(name);
   BucketList l = currentScope->hashTable[h];
   while (l != NULL) {
       if (strcmp(name, l->name) == 0&&strcmp(kind,l->kind)==0) {
           return l;
       }
       l = l->next;
   }
   return NULL;
}
```

현재 scope에서 이름과 타입(함수, 변수)이 동일한 bucket이 있는 지 검사하는 함수

AST 순회하면서 symbol table을 만들기 위한 행위

Decl→Funck

Decl→param

Stmt→comound

Exp→call, var, assign

- **Funck**
  AST의 node 하나를 받아서 declaration의 function이라면 제일 먼저 현재 scope에 동일한 이름의 변수가 선언된 적이 있는 지 탐색. st_look_up 함수는 currnetScope만 검사한다. 부모로 타고 가지 않음.
  c-minus의 경우 동일한 scope 내에서 변수와 함수 이름이 동일한 것을 허용하지 않기 때문에 그냥 이름이 동일한 값이 있는 지 확인하고, 존재한다면 redefine 에러를 띄운다. 그리고 사용 혹은 선언된 줄이기 때문에 line linked list에 해당 line no을 추가한다.
  선언된 적이 없었다면 symbol table에 해당 함수를 추가하고 새로운 scope 추가. 함수 내부에 여러 중첩문이 생기는 상황을 고려해서 scopeDepth 전역 변수를 설정해서 함수가 시작되었다는 의미로 1로 정의. 그리고 현재 scope의 paramlist에서 이 함수에 해당하는 주소를 currentParameterList에 저장.
- **Params**
  parameter를 받으면 똑같이 현재 scope에 존재하는 지 확인하고 없다면 symbol table에 해당 변수 추가, addParameter를 통해서 아까 받은 주소에 해당 인자 추가. void인 경우도 처리해준다. 배열 형식의 인자를 받을 때도 동일하다.
  만약 아예 parameter가 없다면 symbol table에 추가하지도 않고, param이 NULL이면 알아서 void로 인식하기 때문에 별도로 수행할 일 없음.
- **Compound**
  만약 {가 온다면 이게 함수 바로 뒤에 오는 건지 중첩인 지 확인해야 함. 따라서 scopeDepth가 1이라면 함수 바로 뒤에 오는 것이기 때문에 scope를 추가하지 않고 depth만 증가시킴. 만약 중첩된 상태라면 새로운 scope를 만들어야 하는데 이 때 이름은 고유한 이름을 만들기 위해 아까 선언한 childCounter를 이용해서 만들어준다. 중첩이기 때문에 중첩 level을 update해주어야 한다. current Scope은 pushScope에 의해 변경.
- **Exp**
  call이라면 현재 scope에 해당 이름의 함수가 있는지 확인하고 없다면 undeclare 에러, 후 undetermined 상태로 table에 추가. 그 외 변수 사용할 때도 동일하게 처리.
  assign의 type check는 추후에 하기 때문에 symbol table을 만들 때는 따로 처리하지 않았다.

Preorder로 모든 node를 탐색했다면 postorder로 postProc 함수 수행

```
void postProc(TreeNode *t) {
    if (t->nodekind == StmtK && t->kind.stmt == CompoundK) {
        popScope();
        scopeDepth--;
    }
}
```

compound라면 scope이 달라지기 때문에 조건문을 걸어서 Scope를 나갈 수 있게 한다.
postOrder는 이렇게 scope을 처리하고 끝냄. symbol table이 다 만들어지면 type check 수행

type check

```
void typeCheck(TreeNode *syntaxTree) {

    //printf("%d\n",scopeIndex);
    for(int i=0; i<scopeIndex; i++){
      //printf("%s ",scopeArray[i]->name);
    }
    checkIndex=0;
    currentScope=scopeArray[checkIndex++];
    traverse(syntaxTree, checkScope, checkNode);
}
```

type check 시작

```
static void checkScope(TreeNode *t) {
  switch (t->nodekind) {
    case DeclK:
      switch (t->kind.decl) {
        case FuncK:
          //printf("check funck %s, %d\n",t->attr.name,t->lineno);
          funcName=t->attr.name;
          if(checkIndex==scopeIndex){
            printf("Index out of range!\n");
          }else{
            //printf("check scope funck curscope: %s\n",currentScope->name);
            currentScope=scopeArray[checkIndex++];
            scopeDepth=1;
            //printf("after check scope curscope: %s\n",currentScope->name);
          }break;
        default:
          break;
      }
    break;
```

checkScope Decl

```
case StmtK:
  switch (t->kind.stmt){
    case CompoundK:
      //printf("compound %s, %d\n",t->attr.name,t->lineno);
      if(scopeDepth==0){
        //printf("첫 번째 compound %d\n",t->lineno);
        scopeDepth++;
      }
      else{
        //printf("check scope compound curscope: %s\n",currentScope->name);
        currentScope=scopeArray[checkIndex];
        scopeDepth--;
        //printf("after check scope curscope: %s\n",currentScope->name);
      }
      break;
    default:
      break;
  }
break;
case ExpK:
  break;
```

checkScope Stmt

- **type check**
  scopeindex는 symbol table을 만들 때 삽입된 총 scope의 수, checkIndex는 type check할 때 배열을 탐색하기 위한 index. type check도 scope 기반으로 해야 하기 때문에 preorder에서 scope을 조종하고, postorder에서 검사를 수행. scope는 symbol table 만들 때 array에 저장했던 scopes를 기반으로 검사.
- **checkScope Decl**
  function이면 새로운 scope가 들어온 것.
  currentScope를 scopeArray에서 한 칸 뒤로 이동. 한 칸 뒤로 갈 수록 늦게 나온 scope(중첩 수준 아니고 코드 상에서 늦게 나온) 여기서도 함수 내부에 중첩이 생길 수 있기 때문에 scopeDepth 변수 사용
- **checkScope Stmt**
  compound라면 새로운 scope인지 함수의 scope인지 판단 위해 scopeDepth로 판별. 새 scope이라면 역시 한 칸 뒤로 이동.

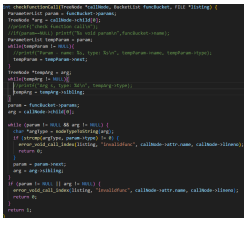
preorder로 scope를 다 설정했다면 postorder로 node를 검사한다.

checkNode assign

```
case CompoundK:
  //printf("compound cur scope: %s, ch
  checkIndex-=2;
  currentScope=scopeArray[checkIndex];
  scopeDepth--;
  //printf("compound after curScope: %s
  break;
```

checknode comound

- **Assign**
  c-minus의 문법에선 LHS에 함수가 올 수 없다. 따라서 Assign의 LHS에 함수랑 동일한 이름을 가진 임의의 문자가 올 순 있어도 kind가 함수일 순 없기 때문에 symbol table에서 해당 이름의 변수를 찾고, 좌변과 우변의 type 비교.
  둘 다 int거나 둘 다 intarray일 때만 허용. 그 외는 invalid로 설정
- **Compound**
  post order에서 compound를 만났다면 scope 밖으로 나가는 것이기 때문에 index와 currentScope 수정 필요. index는 바로 한 칸 전이 이전 scope인데 들어올 때 미리 ++을 해놨기 때문에 2를 빼야 함



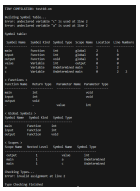
checkFunctionCall

checkNode call

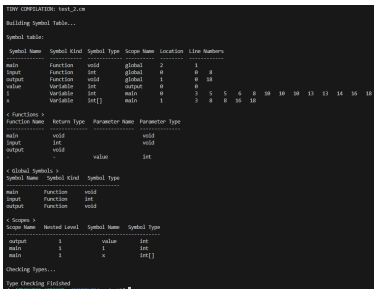
- **Call**
  호출하는 함수와 인자가 일치하는 지 확인해야 함. 일단 symbol table에서 해당 이름의 함수를 찾고, 해당 node의 인자와 symbol table에 저장되어 있던 매개변수 비교.
  node의 인자는 child[0]부터 sibling으로 연결, symbol table은 linked list이므로 next로 연결되어 있음. 둘 중 하나가 NULL이 될 때까지 다음 인자로 넘어가면서 둘의 type이 일치하는 지 확인.
  만약 하나라도 일치하지 않는다면 에러 문구 출력, 둘 중 하나가 NULL이 되어서 끝났는데 둘 중 하나라도 NULL이 아닌 게 있다면 인자 수가 안 맞는 거니 에러 문구 출력.

위와 같은 방식으로 모든 case를 처리해주면 된다.

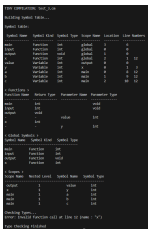
Result



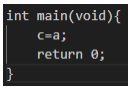
result 1



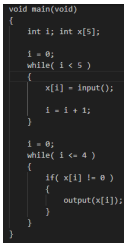
result 2



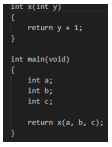
result3



test1



test2



test3

Trouble shooting

```
typedef enum {Void,Integer,IntArray,
VoidArray,Undetermined} ExpType;
```

원래 type check를 할 때 undetermined 상태가 되면 node 자체를 undetermined로 설정하고 추후에 나오는 코드들에서도 undetermined 상태로 검사한다고 생각하여 node의 type에 Undetermined를 추가했었다. 그러나 test file과 result file을 토대로 undetermined가 되면 그 순간에만 에러 문구를 출력하고, 그 뒤엔 정상적인 자료형으로 판단하여 검사한다는 사실을 알게 되었고, 해당 type을 사용하진 않았다.

- st_insert하고 st_insert_function에서 또 삽입하려고 해서 redefined error 발생.
  function은 st_insert_function만 호출하도록 수정

- location을 전역변수로 설정했었는데 중첩되어 있다가 나오니까 0으로 초기화되는 문제 생겨서 아예 member 변수로 설정함
- 재정의되는 경우만 고려해서 선언했던 변수가 사용되거나 호출될 때는 그냥 undeclared 검사만 했었는데 그러니까 line num에 선언된 순간만 나옴 그래서 insert line 함수 추가해서 linked list에 연결하도록 수정