Semantic Analysis

컴퓨터전공

2013011822

김병조

Modification code (globals.h, main.c, symtab.*, analyze.*)

globals.h -> TreeNode 구조체에 scope 속성 추가

```
typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];
     struct treeNode * sibling;
     int lineno;

     NodeKind nodekind;
     union {
        StmtKind stmt;
        ExpKind exp;
        DecKind dec;
     } kind;

     union {
        TokenType op;
        int val;
        char* name;
     } attr;
     int size;
     ExpType type; /* for type checking of exps */
     struct ScopeListRec *scope;

   } TreeNode;
```

Symbol Table을 만들면서 Compound가 끝낼 때 그 전 scope를 저장하는 용도와, 그것을 이용하여 type checker에 용이할 수 있게 구조체에 scope라는 속서을 추가해 놓는다.

main.c

```
int TraceAnalyze = TRUE;
```

TraceAnalyze 를 TRUE로 바꾸어 Symbol Table이 화면에 나타낼 수 있도록 한다. 이것 뿐만 아니라 NO_ANALYZE 또한 FALSE로 만들어 ANALYZE 기능을 수행 할 수 있도록 한다. 이때 주의 할 점은 NO_PARSE 또한 FALSE로 만들어 AST가 만들어 질 수 있도록 한다.

symtab.h, symtab.c

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

/* The record in the bucket lists for
 * each variable, including name,
 * assigned memory location, and
 * the list of line numbers in which
 * it appears in the source code
 */
typedef struct BucketListRec
   { char * name;
     ExpType type;
     LineList lines;
     int memloc ; /* memory location for variable */
     struct BucketListRec * next;
     TreeNode *treenode;
   } * BucketList;

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

LineList, BucketList, ScopeList 구조체를 만든다.

ScopeList는 scope의 이름, 가지고 있는 bucket의 구조체 배열, 그리고 부모의 scope 정보를 가지고 있다.

BucketList는 각 BucketList의 이름과 타입, LineList로 선언된 줄number, 각 변수가 위치 할 메모리 위치 정보, 그리고 해쉬 테이블을 이용하기 때문에 next BucketList 포인터, 마지막으로 트리구조 속성을 가지고 있다. 이때 트리구조의 속성을 추가한 이유는 함수 call 할 때 parametes와 기존 선언된 함수의 arguments들의 수, 타입 등을 체크 할 때 애초에 각 트리구조를 미리 집어 넣어 놓으면 편하기 때문에 treenode라는 속성을 추가하였다.

```
ScopeList scope_top();
ScopeList scope_create(char *name);
void scope_pop();
void scope_push(ScopeList scope);
void st_insert( char * scope, char * name, ExpType type, int lineno, int loc, TreeNode *t);
void just_add_line(char * name, int lineno);
int addLocation();

//BucketList st_lookup ( char * scope, char * name);
BucketList st_lookup (char * name);
BucketList st_lookup_excluding_parent ( char * scope, char * name);

void printSymTab(FILE * listing);
```

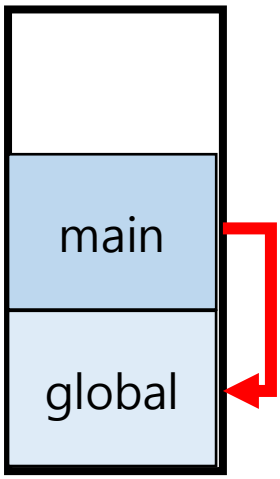
symtab.c 있는 함수를 선언한 모습이다. 과제 pdf 에는 st_lookup 함수에 scope의 이름 정보를 가진 인자가 존재 하였지만 구현하다 보니 필요성을 느끼지 못해서 삭제 하였다. 하지만 st_lookup_excluding_parent 에는 현재 scope를 확인 할 때 쓰여도 괜찮을 것 같아서 삭제하지 않았다.

```
static ScopeList totalScope[1000];
static int ntotalScope = 0;
static ScopeList scopeStack[1000];
static int nScopeStack = 0;
static int location[1000];
```

```
ScopeList scope_top(){
  return scopeStack[nScopeStack - 1];
}

ScopeList scope_create(char *name){
  ScopeList new;

  new = (ScopeList) malloc(sizeof(struct ScopeListRec));
  new->name = name;
  new-> parent = scope_top();

  totalScope[ntotalScope++] = new;
  return new;
}

void scope_pop(){
  nScopeStack = nScopeStack-1;
}

void scope_push(ScopeList scope){
  scopeStack[nScopeStack] = scope;
  location[nScopeStack++] = 0;
}
```

ScopeList들은 Stack으로 관리 한다. 이를 위해 scopeStack과 항상 top을 가리키는 nScopeStack 변수를 선언 하였다.

scope_top, scope_create, scope_pop, scope_push 이 4개의 함수는 스택을 이용하기 위한 함수이다.



scope_create 함수는 함수명 그대로 scope를 새로 만드는 함수이다. scope의 이름이 인자로 들어오면, ScopeList 만큼의 메모리를 할당 한 후 각 속성에 값을 채워 놓는다. 그리고 이 scope가 속해 있는 더 큰 scope를 부모 scope로써 포인터를 설정 한다.

왼쪽 그림은 global이라는 이름을 가진 scope가 존재 하는 중에 main이라는 scope가 새로 생성되어 (선언) 스택에 push 된 모습이다. 이 때 main의 이름을 가진 scope는 부모 scope로 global을 가진다. main scope 안에 또 다른 scope가 생성 될 때는 똑 같은 방식으로 스택에 쌓이게 되고 main scope를 부모로 가진다. 그리고 main scope가 종료가 되면 (compound statement 를 만나면) main scope는 pop이 되어지고 global scope의 남은 TreeNode 들을 살피게 된다.

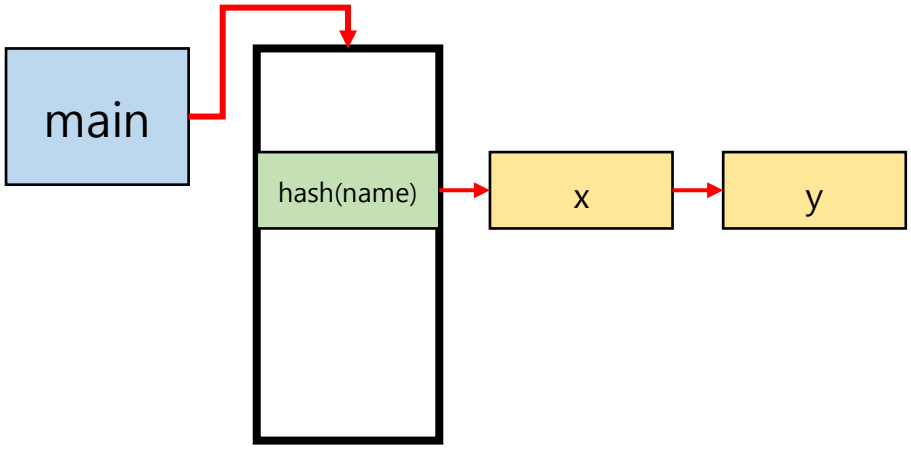
한 scope가 push가 되면 현재 location의 정보 또한 업데이트 되야 한다. Stack에 scope가 push되면 location 배열에 맞는 위치를 0으로 초기화 하고, bucketlist가 생길 때 마다 그것을 높여 줌으로 memloc를 할당 해주는 방식이다.

이 scopestack은 실시간으로 이루어지기 때문에 symbol table을 출력 할 때 쓰이지 않는다. 때문에 totalScope라는 scopelist 배열을 만들었다. 이는 scope가 새로 생성 될 때만 접근 하여 새로운 scope를 집어 넣음으로 써 나중에 출력에 용이하도록 한다.

scope를 관리하는 stack등의 자료구조들은 이러한 방식으로 운영이 된다.

```c
void st_insert(char *scope, char * name, ExpType type, int lineno, int loc, TreeNode *t)
{
  int h = hash(name);
  ScopeList sc = scope_top();


  while(sc){
    if(strcmp(sc->name, scope) == 0){
      break;
    }

    sc = sc->parent;
  }
  BucketList l =  sc->bucket[h];

  while ((l != NULL) && (strcmp(name,l->name) != 0))
    l = l->next;

  if (l == NULL) /* variable not yet in table */
  {

    l = (BucketList) malloc(sizeof(struct BucketListRec));
    l->name = name;

    l->lines = (LineList) malloc(sizeof(struct LineListRec));
    l->lines->lineno = lineno;
    l->lines->next = NULL;

    l->type = type;
    l->memloc = loc;
    l->next = sc->bucket[h];
    l->treenode = t;
    sc->bucket[h] = l;
  }

  else /* found in table, so just add line number */
  {
    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;
  }
} /* st_insert */
```

scope에 bucketList를 추가하는 함수이다. hash함수에 bucketList의 이름을 이용하여 hash값을 얻어 낸다. 인자로 주어진 scope이름을 이용하여 scope_top 부터 맞는 scope가 나올 때 까지 탐색을 한다. 얻어낸 scope의 bucket배열에 hash값 번째에 bucketList를 집어 넣는다. 이 때 그 자리에 동일한 bucketList가 없다면 새로 추가한다. hashtable처럼 기존(다른 이름의) bucketList에 링크드리스트로 이어 붙인다. 하지만 동일한 bucketLlist가 있다면 그 bucketList에 현재 lineno를 집어넣어서 어디서 그 변수나 함수를 사용했는지 기록한다.



위의 그림은 main scope안에 x와 y의 hash 값이 같을 경우 bucket의 next 속성을 이용하여 구현된 그림이다. x와 y 같은 각 bucketList에는 위에 언급한 구조체 속성들을 가지고있다.

하지만 이미 같은 이름을 가진 bucketList가 존재하는지에 대한 여부는 집어 넣기 전에 항상 체크가 되어진다. 이때 scope를 또 탐색할 필요가 없고, st_insert 함수 하나로 모든 것을 하려고 하니 이미 선언한 함수를 집어넣을 때 scope의 이름을 가지고 scope를 찾는 부분이 잘 작동 되지 않았다. 따라서 새로운 함수를 만들었다.

```c
void just_add_line(char * name, int lineno){

  BucketList l = st_lookup(name);

  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;
}
```

단순하게 이름과 lineno 정보를 가지고 이미 존재하는 buckeList에 추가하는 함수이다. 이로써 scopeList와 bucketList를 관리하는 함수들은 모두 완성 되었다. 이제 bucketList가 존재하는지 여부를 탐색하는 함수를 소개한다.

```
BucketList st_lookup (char * name){
  int h = hash(name);
  ScopeList sc = scope_top();

  while(sc){
     BucketList l = sc->bucket[h];
     while(l!=NULL){
       if(strcmp(l->name, name) == 0) return l;
       l = l->next;
     }

    sc = sc->parent;
  }

  return NULL;
}

BucketList st_lookup_excluding_parent ( char * scope, char * name){
  int h = hash(name);
  ScopeList sc = scope_top();

  if(strcmp(sc->name, scope)){
    BucketList l = sc->bucket[h];
    while(l!=NULL){
      if(strcmp(l->name, name) == 0) return l;
      l = l->next;
    }
  }

  return NULL;
}
```

이름으로 해당 bucketList를 찾는 함수는 두개로 이루어져 있다. 하나는 현재 스택에 가장 위에 있는 scope로부터 부모 scope로 탐색 하면서 찾는 st_lookoup 함수이다. 이때도 물론 이름의 hash값을 이용하여 각 scope 가 가지고 있는 bucket 배열에 같은 이름이 있는지 찾는다. 찾으면 그 bucketList를 return 하고 못 찾으면 NULL을 return 한다. 이 함수는 변수가 사용 될 때 지역변수부터 시작하여 전역변수에 선언이 되었는지 찾는 기능을 한다.

다른 하나는 해당 scope 안에서만 찾는 st_lookup_excluding_parent 함수 이다. scope 이름을 인자로 받아서 그 scope의 bucket만 검사를 한다. 이 함수가 call 될 때는 그 지역의 scope가 항상 top이기 때문에 scope를 찾는 소스는 추가하지 않았고, 현재 scope가 이 이름이 맞는지 확인만 하였다. 사실 이 함수 또한 scope의 이름이 필요 없지만 추가를 하였다. 이때 역시 찾으면 해당 bucketList를 아니면 NULL를 return 하도록 하였다.

```
void printSymTab(FILE * listing){
  int i;
  int j;
  fprintf(listing,"Variable Name\tType\tLocation\tScope\t\tLine Numbers\n");
  fprintf(listing,"-------------\t-----\t---------\t------\t\t-------------\n");

  for(j = 0 ; j < ntotalScope; j++){
    ScopeList sc = totalScope[j];

    for (i=0;i<SIZE;++i)
    { if (sc->bucket[i] != NULL)
      {
        BucketList l = sc->bucket[i];
        while (l != NULL)
        { LineList t = l->lines;
          fprintf(listing,"%s\t\t",l->name);
          fprintf(listing,"%s\t\t", typeString[l->type]);
          fprintf(listing,"%d\t",l->memloc);
          fprintf(listing,"%s\t\t",sc->name);
          while (t != NULL)
          { fprintf(listing,"%d,",t->lineno);
            t = t->next;
          }
          fprintf(listing,"\n");
          l = l->next;
        }
      }
    }
  }
}
```

마지막으로 만들어진 symbol table을 출력하는 함수 printSymTab 함수가 있다. 위에서 언급 한 totalScope 배열이 여기에 쓰인다. 여태까지 사용한 모든 scopeList를 가지고 있는 totalScope를 가지고 각 scope의 bucket 배열을 차례대로 출력을 해준다. bukcetList의 이름과 타입, 그리고 location 또한 존재하고 있는 scope이름과 위치 하고 있는 line number을 출력을 해줌으로써 이 소스에 어떤 함수와 변수가 어디서 선언되고 쓰였는지 한 눈에 확인 할수 있게 한다.

Variable Name	Type	Location	Scope	Line Numbers
main	void	3	Global	11,
input	int	0	Global	0,14,14,
output	void	1	Global	0,15,
gcd	int	2	Global	4,7,15,
u	int	0	gcd	4,6,7,7,
v	int	1	gcd	4,6,7,7,7,
b	int[]	0	main	11,
x	int	1	main	13,14,15,
y	int	2	main	13,14,15,

위의 그림은 gcd 예제를 가지고 만들어진 symbol table 이다.

analyze.h, analyze.c

```
void buildSymtab(TreeNode *);
void typeCheck(TreeNode *);
```

analyze.c에는 크게 두 함수로 이루어져있다. symbol table을 만들어주는 buildSymtab 함수와 타입을 체크하여 틀린 문법이 있는지 확인하는 typeCheck 함수이다.

```
void buildSymtab(TreeNode * syntaxTree)
{
  global = scope_create(funcName);
  scope_push(global);
  inoutput();
  traverse(syntaxTree,insertNode,forPop);

  scope_pop();

  if (TraceAnalyze)
  { fprintf(listing,"\nSymbol table:\n\n");
    printSymTab(listing);
  }
}
```

buildSymtab 함수에는 우선 기본적인 scope인 global scope가 만들어지고 scope stack에 push 되어야 한다. 이때 현재 scope를 기억하고있는 funcName 이라는 전역 변수를 만들어 놓는다. funcName에는 초기 값으로 "Global" 문자열이 들어가 있다. 이를 통해 global이라는 scope를 만들고 push를 한다. 그리고 int input과 void output 함수는 기본으로 global scope의 bucket에 들어가 있어야한다.

```
static void inoutput(){

  TreeNode *input;

  input = newDecNode(VarK);
  input->type = Integer;

  input->kind.dec = FunK;
  input->attr.name = (char*)malloc(strlen("input")+1);
  strcpy(input->attr.name, "input");
  input->child[0] = NULL;
  input->child[1] = NULL;


  TreeNode *output;

  output = newDecNode(VarK);
  output->type = Void;

  output->kind.dec = FunK;
  output->attr.name = (char*)malloc(strlen("output")+1);
  strcpy(output->attr.name, "output");
  output->child[0] = NULL;
  output->child[1] = NULL;

  st_insert(funcName, input->attr.name, input->type, 0, addLocation(), input);
  st_insert(funcName, output->attr.name, output->type, 0, addLocation(), output);

}
```

이 부분이 input과 output 함수를 만들어서 bucket에 집어 넣는 부분이다. cminus.y의 fun_dec 부분을 이용하여 그대로 하드코딩 해서 집어 넣었다.

다시 돌아가서 buildSymtab 함수에서 traverse 함수를 이용하여 syntaxTree 의 각 TreeNode *t를 재귀적으로 탐색하게 된다. 이때 insertNode 함수와 forPop 함수를 사용하게된다. 모든 탐색이 끝나고 마지막으로 global scope를 pop 하고 symbol table을 출력한다.

```
static void insertNode( TreeNode * t)
{
  switch (t->nodekind)
  {
    case StmtK:
      switch(t->kind.stmt){...
      }
      break;

    case ExpK:
      switch(t->kind.exp){...
      }
      break;

    case DecK:
      switch(t->kind.dec){...
      }
      break;

    default:
      break;
  }
}
```

insertNode라는 함수는 각 TreeNode *t를 분석하여 nodeKind와 종류에 따라 scope를 만들 것 인지, bucketList를 추가할 것 인지 등의 행동을 결정한다.

TreeNode *t가 StmtK 일 때

```
case StmtK:
  switch(t->kind.stmt){
    case IfK:
      if(t->child[2])
        flag = 1;
      funcName = ".if";

      break;

    case WhileK:
      flag = 3;
      break;

    case CompK:

      if(stay == 1){
        stay = 0;
        flag = 0;
      }else{
        if(flag == 1){
          flag = 2;
        }
        else if(flag == 2){
          funcName = ".else";
          flag = 0;
        }
        else if(flag == 3){
          funcName = ".while";
          flag = 0;
        }
        scope_push(scope_create(funcName));
      }
      t->scope = scope_top();

      break;

    default:
      break;
  }
  break;
```

StmtK에는 IfK, WhileK, CompK로 나누어 행동한다. 하지만 IfK와 WhileK와 같은 경우 scope이름을 위한 flag만 바꿔줄 뿐 아무 행동을 하지 않는다.

CompK일 경우 함수가 선언되고 난 후의 CompK는 flag만을 바꿔준다. 하지만 그것이 아닌 while이나 if의 compound의 경우 새로운 scope를 만들어 준다. 이때 flag에 따라 IfK 또는 WhileK의 scope의 이름을 따로 현재 funcName뒤에 ".if"나 ".while"을 붙여서 변경 할려고 하였지만 뜻대로 되지가 않았다. 결국에는 새로 생기는 scope가 단순하게 ".if" 그리고 ".else" 또는 ".while"의 이름을 가지도록 funcName을 바꿔주었다. if else 문을 처리하기위해 단계별로 flag를 변화 시켜 주었다. C언어 문자열 다루는 것이 많이 미숙해서 원하는 기능을 애매하게나마 추가하였다. 마지막으로 현재 TreeNode t가 scope stack 맨 위 scop의 정보를 가지고 있게 함으로 써 나중에 typecheck 할때 또 다른 자료구조를 사용 안해도 되도록 한다.

TreeNode *t가 ExpK 일 때

```
case ExpK:
  switch(t->kind.exp){
    case OpK:
      break;
    case ConstK:
      break;

    case IdK:
    case CallK:
    case ArrIdK:
      if(st_lookup(t->attr.name)){
        just_add_line(t->attr.name, t->lineno);
      }else{
        symbolError(t, "Undeclared");
        break;
      }

    default:
      break;
  }
  break;
```

ExpK에는 OpK, ConstK, IdK, CallK, ArrIDK로 나누어 행동한다. OpK와 ConstK일 경우 bucket에 추가 하지 않기 때문에 아무 행동도 하지않지만 IdK, CallK, ArrIDK와 같이 이미 선언되어져 있는 것을 사용 할 경우에는 st_lookup 함수를 이용하여 현재 scope 부터 시작하여 부모 scope들에 선언되어있는지 확인을 한다. 이때 선언이 되어져있으면 현재 사용되고 있다는 정보를 추가하기 위해 just_add_line 함수를 사용하여 기존 bucketList에 줄 number을 추가한다. 하지만 선언되어있지 않은데 사용하는 경우 에러를 출력한다.

TreeNode *t가 DecK 일 때

```
case DecK:
  switch(t->kind.dec){
    case FunK:
      fprintf(listing, "%d\n", t->lineno);

      if(st_lookup_excluding_parent(funcName, t->attr.name)) {
        symbolError(t, "function already declared in same scope");
        break;
      }

      st_insert( funcName, t->attr.name, t->type, t->lineno, addLocation(), t);
      scope_push(scope_create(t->attr.name));
      funcName = t->attr.name;

      stay = 1;
      break;

    case VarK:

      if(st_lookup_excluding_parent(funcName, t->attr.name)){
        symbolError(t, "Var already declared in same scope");
        break;
      }
      st_insert( funcName, t->attr.name, t->type, t->lineno, addLocation(), t);
      break;

    case ParamK:
      if(t->type != Void){
        st_insert(funcName, t->attr.name, t->type, t->lineno, addLocation(), t);
      }
      break;
  }
  break;
```

저번 Parsing 과제를 할 때 DecKind에는 VarK, FunK, ParamK 총 3가지 종류를 분리해 놓았다. 이에 따라 행동이 달라진다.

우선 함수를 선언할 때(FunK) 같은 scope에 똑 같은 이름을 가진 함수가 이미 선언되어져있는지 확인을 한다. 따라서 st_lookup 함수보다 st_lookup_excluding_parent 함수가 유용하다. 함수가 이미 선언되어있는지 확인하고 선언이 되어져있으면 error을 출력한다. 반면에 선언이 되어져있지 않은 경우, 현재 funcName과 TreeNode *t의 정보들을 st_insert 함수를 이용하여 현재 scope의 bucket에 집어 넣는다. 함수가 선언되어 또다른 scope를 stack에 넣어야 하기 때문에 t의 이름을 가진 scope를 생성하고 stack에 넣는다. 그리고 실시간의 scope이름을 바꿔주기 위해 funcName 또한 바꿔준다. 함수가 선언되면 다음에 compound statement가 나올 것이기 때문에 scope의 재생성을 방지하기위해 flag를 1로 둔다. 이때 flag의 이름은 stay로 전역변수로 선언하였다.

Vark 또한 마찬가지로 이미 선언 되어져 있으면 에러를 아니면 bucket에 추가를 한다. ParamK의 경우에는 타입이 void가 아닐경우에만 bucket에 추가를 한다.

```
static void forPop( TreeNode * t){
  if(t->nodekind == StmtK){
    if(t->kind.stmt == CompK){
      scope_pop();
      ScopeList sc = scope_top();
      funcName = sc->name;
    }
  }
}
```

scope stack에 집어넣거나, bucketList에 추가를 한 후 에 scope가 끝나면 (CompK가 들어오면) scope_pop 함수를 사용하여 scope stack에 있는 젤 위의 scope를 정리한다. 그리고 funcName 또한 변화된 scope stack의 top의 이름을 넣어 아직 끝나지 않은 부모 scope의 처리를 할 수 있게 한다.

```
void typeCheck(TreeNode * syntaxTree)
{
  scope_push(global);
  traverse(syntaxTree,forPush,checkNode);
  scope_pop();
}
```

typeCheck 함수 또한 같은 프로세스로 진행이 된다. global scope 를 우선 scope sctack 에 push 를 한 후 traverse 함수로 ast 를 순회하면서 각 TreeNode *t 의 타입을 체크하게 된다.

```
static void forPush (TreeNode *t){
  switch (t->nodekind)
  {
    case StmtK:
      switch(t->kind.stmt){
        case IfK:
          //funcName = strcat(funcName, ".IF");
          break;

        case WhileK:
          //funcName = strcat(funcName, ".WHILE");
          break;

        case CompK:
          scope_push(t->scope);

          break;

        default:
          break;
      }
      break;

    case DecK:
      switch(t->kind.dec){
        case FunK:
          funcName = t->attr.name;
          break;
      }
      break;

    default:
      break;
  }
}
```

buildSymtab 과 다르게 우선 Compound statement 가 들어오면 scope 를 push 하기만 한다. 이때 전에 설정해놓은 TreeNode *t 의 scope 속성을 이용하여 간편하게 scope stack 에 push 되도록 할 수 있다.

checkNode 함수 또한 TreeNode *t 의 nodekind 를 가지고 행하는 행동들이 달라진다.

TreeNode *t 가 StmtK 일 때

```
case StmtK:
  switch (t->kind.stmt)
  {
    case IfK:
      if(t->child[0]->type == Void)
        typeError(t->child[0], "void is only available for function");
      break;

    case WhileK:
      if(t->child[0]->type == Void)
        typeError(t->child[0], "void is only available for function");
      break;

    case CompK:
      scope_pop();
      funcName = scope_top()->name;
      break;

    case RetK:
    {
      ExpType FuncType = st_lookup(funcName)->type;

      if(FuncType == Void && (t->child[0] != NULL || t->child[0]->type != Void))
        typeError(t, "void Function should return void");
      else if(FuncType == Integer && (t->child[0] == NULL || t->child[0]->type != Integer))
        typeError(t, "integer Function should return integer");
      break;
    }
    default:
      break;
  }
  break;
```

StmtK 는 IfK, WhileK, CompK, RetK 로 나누어 행동한다. IfK 와 WhileK 의 조건은 Void 타입이면 안되기 때문에 Void 타입체크를 하였다. CompK 에 경우 scope 를 pop 하면서 funcName 을 변화시킨다.

RetK 가 들어올 경우 그 함수의 type 을 우선 변수에 저장해 놓는다. 이때 함수의 타입이 Void 일 경우 return 되는 값이 없어야 하고, 함수의 타입이 Integer 인 경우 return 되는 값이 void 가 되지 않도록 타입을 체크한다.

TreeNode *t 가 ExpK 일 때

TreeNode *t 가 ExpK 일 경우 type 에 가장 신경을 써야한다. 전체적인 프로세스는 AST 를 순회하는 순서대로 각 변수가 선언되어져있는 BucketList 를 찾고 그 BucketList 의 type 을 현재 TreeNode *t 의 type 에 집어 넣는다. 그 후 OpK 에서 사용되는 각 변수들의 타입을 체크한다.

```
case ConstK:
  t->type = Integer;
  break;
```

ConstK 인 경우 TreeNode *t 의 type 에 Integer 를 집어 넣는다.

```
case IdK:{

  BucketList b = st_lookup(t->attr.name);
  if(b == NULL){
    break;
  }

  t->type = b->type;
  break;
  }

case ArrIdK:{
  BucketList b = st_lookup(t->attr.name);

  if(b == NULL){
    break;
  }

  if(t->child[0]->type != Integer)
    typeError(t->child[0], "exp should be Integer");
  else{
    t->type = Integer;
  }

  break;
}
```

IdK 또는 ArrIdK 인 경우 st_loookup 으로 선언되어 저장된 BukcetList 를 찾고, 그 buckList 의 type 을 TreeNode *t 의 type 에 집어 넣는다. 이때 ArrIdK 인 경우 array[0]에서의 0 의 부분이 Integer 인지 확인도 한다. 그리고 array 는 배열이지만 int 로 선언된 배열의 한 부분 array[0]은 int 형이기 때문에 cminus 특성상 Integer 를 t 의 type 에 집어 넣는다. cminus 가 아니고 C 언어 일 경우 배열을 선언 할 때 배열인지에 대한 타입 뿐만 아니라 구체적으로 어떤 타입인지에 대한 속성을 하나 더 추가해놓아야 겠다라는 생각이 들었다.

```
 case CallK:{
   BucketList b = st_lookup(t->attr.name);

   if(b == NULL)
     break;

   TreeNode *args = t->child[0];
   TreeNode *params = b->treenode->child[0];

   while(params != NULL){
     if(args == NULL){
       typeError(args, "num(args) and num(params) should be same");
       break;
     }
     else if(args->type == Void){
       typeError(args, "void is only available for function");
       break;
     }
     else if(args->type != params->type){
       typeError(args, "args and params should have same type");
       break;
     }
     else{
       args = args->sibling;
       params = params->sibling;
     }
   }

   t->type = b->type;
   break;
 }
```

CallK 일 때 st_lookup 으로 해당 선언된 BucketList 를 가져온 후 그것의 parameter 들과 현재 TreeNode *t 의 argument 부분을 비교한다. 두 변수들은 개수와 타입이 같아야 하고, void 이면 안된다. 이것을 각 sibling 을 각각 순회하면서 비교하게한다. 그리고 마지막으로 Call 한 함수의 type 을 선언된 함수의 type 으로 바꾼다.

```
case OpK:{

  ExpType left = t->child[0]->type;
  ExpType right = t->child[1]->type;
  TokenType op = t->attr.op;

  if( left == Void ){
     typeError(t->child[0], "void is only available for function");
   }
  if( right == Void ){
    typeError(t->child[1], "void is only available for function");
  }

  if(op == ASSIGN){
    if( left != right ){
      typeError(t->child[0], "two operands should be same type when assign");
    }
    else
      t->type = t->child[0]->type;
  }else{
    if( left != right ){
      typeError(t->child[0], "two operands should be same type");
    }
    else if(left == Array && right == Array)
      typeError(t, "two operands shoud not be array");

    else if(op == MINUS && left == Integer && right == Array)
      typeError(t, "minus no int - array");

    else if( (op == TIMES || op == OVER) && (left == Array || right == Array) )
      typeError(t, "no times or over in array");

    else
      t->type = Integer;
  }
  break;
}
```

OpK 인 경우 왼쪽 변수의 type, 오른쪽 변수의 tpye 그리고 op 에 대한 정보를 변수로 저장을 한 후 비교를 하게 된다. 우선 양쪽의 변수는 void 이면 안된다. 그리고 Assign 일 경우 같은 type 을 가져야 하고, array 끼리는 계산을 할 수 없도록 한다.

Makefile

```
symtab.o: symtab.c symtab.h globals.h
    $(CC) $(CFLAGS) -c symtab.c

analyze.o: analyze.c globals.h symtab.h analyze.h util.h
    $(CC) $(CFLAGS) -c analyze.c
```

깃에 올린 소스를 보면 알 수 있듯이 cgen.*과 code.* 부분을 모두 제외 시켰다. 이로써 semantic analysis가 완성 되었다.

Compilation environment

Os : Ubuntu 16.04.3 LTS

gcc : gcc (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609

Example and Result

Example :

test.cm

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

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

void main(int a [])
{
    int x;int y;
    x = input(); y = input();
    output(gcd(x,y));
}

```

result

```
Building Symbol Table...

Symbol table:

Variable Name   Type    Location        Scope           Line Numbers
-------------   -----   ---------       ------          -------------
main            void            3       Global          12,
input           int             0       Global          0,15,15,
output          void            1       Global          0,16,
gcd             int             2       Global          4,8,16,
u               int             0       gcd             4,6,6,6,7,8,8,
v               int             1       gcd             4,7,8,8,8,
a               int[]           0       main            12,
x               int             1       main            14,15,16,
y               int             2       main            14,15,16,

Checking Types...

Type Checking Finished
```

test2.cm

```
int x[20];

int test(int a[], int x)
{
    int y;
    y = x + 3;
    return y;
}

void main(void)
{
    int a;
    int b;
    int y[10];
    a = 10;

    y[test(y,4)] = a;
}
```

result

```
Building Symbol Table...

Symbol table:

Variable Name   Type    Location        Scope           Line Numbers
-------------   -----   ---------       ------          ------------
main            void            4       Global          10,
input           int             0       Global          0,
output          void            1       Global          0,
x               int[]           2       Global          1,
test            int             3       Global          3,17,
a               int[]           0       test            3,
x               int             1       test            3,6,
y               int             2       test            5,6,7,
a               int             0       main            12,15,17,
b               int             1       main            13,
y               int[]           2       main            14,17,17,

Checking Types...

Type Checking Finished
```

test3.cm

```
int x[10];

int minloc (int a[], int low, int high) {
    int i; int x; int k;
    k = low;
    x = a[low];
    i = low + 1;

    while (i < high) {
        if (a[i] < x) {
            x = a[i];
            k = i;
        }
        i = i + 1;
    }
    return k;
}

void sort (int a[], int low, int high) {
    int i; int k;
    i = low;

    while( i < high - 1) {
        int t;
        k = minloc(a, i, high);
        t = a[k];
        a[k] = a[i];
        a[i] = t;
        i = i + 1;
    }
}


void main(void) {
    int i;
    i = 0;

    while ( i < 10 ) {
        x[i] = input();
        i = i + 1;
    }

    sort(x,0,10);
    i = 0;

    while( i < 10 ) {
        output(x[i]);
        i = i + 1;
    }
}
```

result

```
Building Symbol Table...

Symbol table:

Variable Name   Type    Location        Scope           Line Numbers
-------------   -----   ---------       ------          -------------
main            void            5       Global          35,
sort            void            4       Global          19,44,
input           int             0       Global          0,40,
minloc          int             3       Global          3,25,
output          void            1       Global          0,48,
x               int[]           2       Global          1,40,44,48,
low             int             1       minloc          3,5,6,7,
a               int[]           0       minloc          3,6,10,11,
i               int             3       minloc          4,7,9,10,11,12,14,14,
k               int             5       minloc          4,5,12,16,
x               int             4       minloc          4,6,10,11,
high            int             2       minloc          3,9,
low             int             1       sort            19,21,
a               int[]           0       sort            19,25,26,27,27,28,
i               int             3       sort            20,21,23,25,27,28,29,29,
k               int             4       sort            20,25,26,27,
high            int             2       sort            19,23,25,
t               int             0       .while          24,26,28,
i               int             0       main            36,37,39,40,41,41,45,47,48,49,49,

Checking Types...

Type Checking Finished
```

check_if_else.cm

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

int gcd(int u, int v)
{
    u = u + u;
    if (v==0) {
        int b;
        b = 1;
        return u;
    }
    else{
        int c;
        c =2;
        return gcd(v,u-u/v*v);
    }
    /*sad*/
}

void main(int a [])
{
    int x;int y;
    x = input(); y = input();
    output(gcd(x,y));
}

```

result

```
Building Symbol Table...

Symbol table:

Variable Name   Type    Location        Scope           Line Numbers
-------------   -----   ---------       ------          -------------
main            void            3       Global          20,
input           int             0       Global          0,23,23,
output          void            1       Global          0,24,
gcd             int             2       Global          4,15,24,
u               int             0       gcd             4,6,6,6,10,15,15,
v               int             1       gcd             4,7,15,15,15,
b               int             0       .if             8,9,
c               int             0       .else           13,14,
a               int[]           0       main            20,
x               int             1       main            22,23,24,
y               int             2       main            22,23,24,

Checking Types...
```

for_type_check.cm

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

int gcd(int u, int v)
{
    int a[3];
    u = u + u;
    u = u + a;
    a = a[3] + a[2];
    if (v==0) return u;
    else return gcd(v,u-u/v*v);
    /*sad*/
}

void main(int a [])
{
    int x;int y;
    x = output();
    x = input(); y = input();
    output(gcd(x,y));
    return 1;
}

```

result

```
Checking Types...
Type error at line 8: two operands should be same type
Type error at line 8: void is only available for function
Type error at line 8: two operands should be same type when assign
Type error at line 9: two operands should be same type when assign
Type error at line 18: void is only available for function
Type error at line 18: two operands should be same type when assign
Type error at line 21: void Function should return void
```

for_type check2.cm

```
int x;
int a[5];


void funcA(void) {
    int a;
    a = input();
    output(a);
    return a;
}


void funcB(int a) {
    int x[4];
    output(funcA(a,x));
    x = a;
}


int funcC(int a, int b[]) {
    b = 2;
    a = funcB(a);
    funcB(b);
    return b;
}


int funcD(int a, int b, int c[]) {
    funcB(a,b);
    funcA(a);
    funcB(funcD(a,c));
    return a;
}

```

result

```
Checking Types...
Type error at line 9: void Function should return void
Type error at line 15: args and params should have same type
Type error at line 16: two operands should be same type when assign
Type error at line 21: two operands should be same type when assign
Type error at line 22: void is only available for function
Type error at line 22: two operands should be same type when assign
Type error at line 23: args and params should have same type
Type error at line 24: integer Function should return integer
Type error at line 30: args and params should have same type
Type error at line 31: args and params should have same type

Type Checking Finished
```