**Design and Development of Compiler for C- Language**

**(설계 프로젝트 수행 결과)**

**3.Semantic Analyzer**

과목명: [CSE4120] 기초 컴파일러 구성

담당교수: 서강대학교 컴퓨터공학과 정 성 원

개발자: 20161596 서현규, 20161616 이 동 승

개발기간: 2019. 5. 10 - 2019. 5. 27

Semantic Analyzer 결과 보고서

**프로젝트 제목: Design and Development of Compiler for C-Language:**

**Phase 3: Semantic Analyzer**

**제출일: 2019. 05. 28.**

**개발자: 서현규, 이동승**

**역할 분담:**

**- 서현규: symbol table 구현**

**- 이동승: type checking 구현**

**기여도:**

**서현규 (50%), 이동승(50%)**

# I. 개발 목표

이번 프로젝트에서는C language와 유사하지만 그보다는 덜 복잡한 C- language에 대한 Lexical Analyzer, Parser를 토대로 semantic analyzer를 구현하고 이를 통해서 C- language source code가 semantically 올바른지 검사한다.

# II. 개발 범위 및 내용

## 가. 개발 범위

C- language에 대한 Semantic Analyzer를 설계하고 개발한다.

## 나. 개발 내용

C- language의 Semantic Analyzer을 구현하였고 이를 토대로 source file을 input으로 받아들여서 AST를 바탕으로 semantically 맞는지 검사한다.

# III. 추진 일정 및 개발 방법

## 가. 추진 일정

5월 10일 : 프로젝트 전반적인 이해

5월 12~18일 : symbol table구현, symtab.h symtab.c 자세한 구현을 하였다.

5월 19~24일 : type check를 위한 analyze.h analyze.h를 구현하였다.

5월 25일 : Error검사

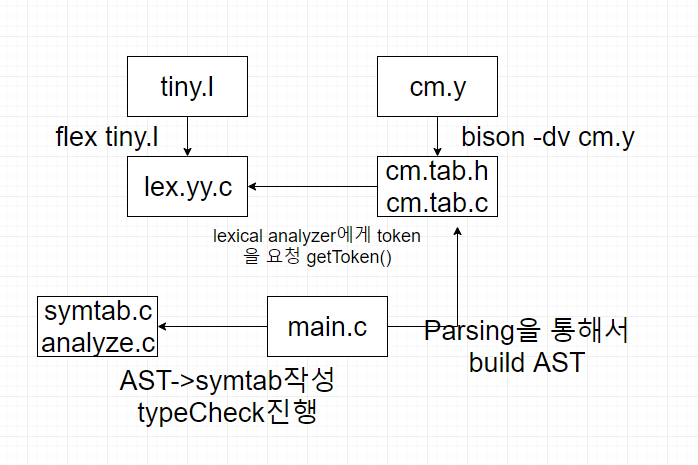
5월 26일 : 보고서 작성

## 나. 개발 방법

C- language에 대한 lexical analyzer를 작성한 것을 토대로 semantic analyzer를 구현한다. 이는 symbol table을 작성하고 이를 type에 대한 검사를 진행한다.

# IV. 연구 결과

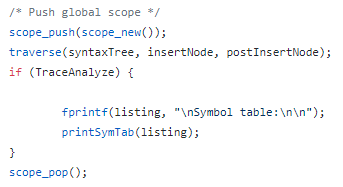
## 1. 합성 내용



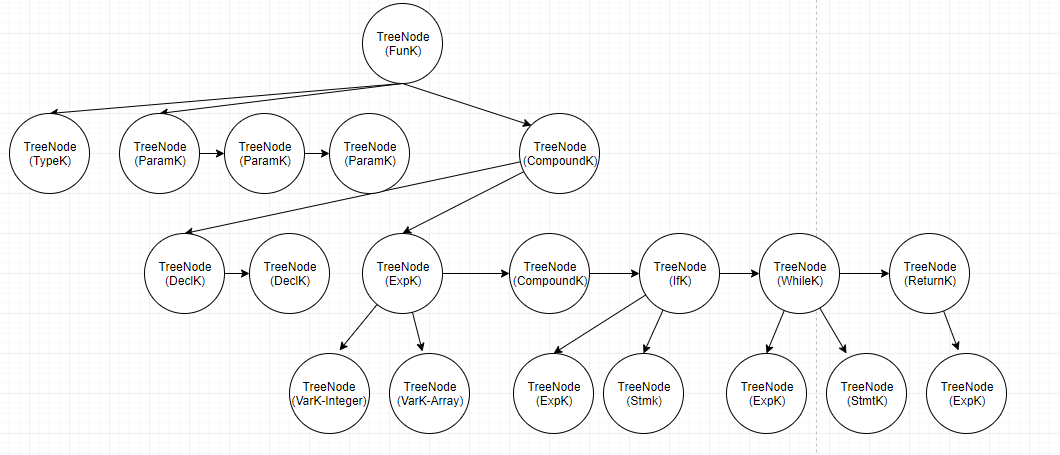
Project3에 대한 전체 program flow는 위의 그림과 같다. main.c에서 cm.tab.c에 parsing을 통한 AST를 요청한다. 이러한 요구를 cm.tab.c는 tiny.l을 flex한 lex.yy.c의 getToken으로 input source file을 tokenize한다 뒤 cm.y에 만들어진 rule에 따라서 LALR(1)으로 parsing한다. 그리고 만들어진 AST를 통해서 symbol table을 작성하고 이를 바탕으로 typeCheck를 진행한다. symbol table에는 variable, function에 대한 이름, Scope, memory location, V/P/F, Array여부 Arrsize 및 type, 사용된 line number에 대한 정보를 담고 있다.

## 2. 분석 내용

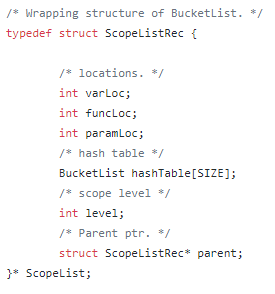
프로젝트를 수행하는 데 있어서 주요한 점은 크게 3가지다. 첫째는 symbol table을 만들기 위한 과정, typeCheck를 하기 위한 과정, 마지막으로 memory location 계산 및 사용된 lineno를 기록하는 것이 그것이다. memory location에 관한 설명은 첫번째 두번째를 진행하면서 설명하겠다. symbol table을 만들기 위해 void buildSymtab (TreeNode \*syntaxTree)으로 lexical analyzer에서 만든 AST를 활용한다. 이때 첫번째 줄에 scope\_push(scope\_new())를 하는 이유는 global variable과 function을 저장하기 위함이다.



먼저 첫 두가지를 구현하기 위해서는 아래의 그림과 같이 C- language가 가질 수 있는 가장 formal한 형태를 바탕으로 preorder, postorder를 적절히 섞어서 구현할 수 있다.



먼저 symbol table을 만드는 것은 symtab.c에 정의 되어 있는 함수들을 먼저 설명하고 analyze.c에서 구현하였다. symtab.h에는 아래와 같이 선언된 structure가 있고 이를 활용하여 scope를 stack으로 구현하였다. varLoc, funLoc, paramLoc은 variable및 function에 대한 location을 기록하기 위한 변수이고 hashTable은 variable및 function name을 저장하기 위한 hash table이다. level은 현재 scope의 level을 뜻하며 parent는 현재 scope의 바로 전 scope를 가리키는 pointer형 변수이다.



struct ScopeListRec\* scope\_top (void)은 scope stack의 가장 top element를 return한다.

void scope\_push (struct ScopeListRec\* sc)는 scope stack에 element를 push한다.

void scope\_pop (void)은 scope stack의 가장 위의 element를 return한다.

struct ScopeListRec\* scope\_new (void)은 scope element를 새로이 만드는 데 이때 memory location을 저장하기 위한 initializing을 하는데 이는 추후에 설명하겠다.

static int hash (char\* key)는 char\* key를 hashing value를 계산한다.

void st\_insert (char\* name, int lineno, int loc, char VPF, int type, int len, TreeNode\* params)현재 scope stack의 가장 위의 element의 hashTable에 symbol table을 위한 정보를 저장한다.

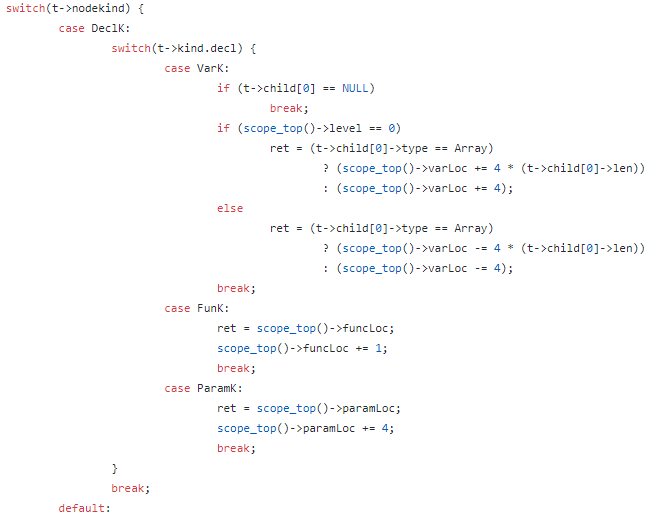
void st\_insert\_global (char\* name, int lineno) lineno를 저장하는 함수이다.

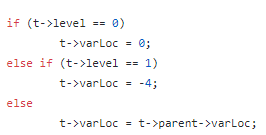
BucketList st\_lookup (char\* name) name을 가진 것을 scope\_top()에서 시작하여 모든 scope의 hashTable을 traverse하면서 찾고 hashTable의 값을 return한다.

BucketList st\_lookup\_local (char\* name) 위와 비슷하지만 scope\_top()에서만 찾는다.

void printSymTab(FILE\* listing) symbol table을 출력하는 함수이다.

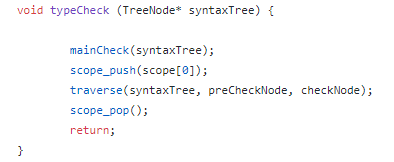
symbol table을 만들기 위해서는 TreeNode를 preorder로 방문하면서 크게 3가지로 나눌 수 있다. 먼저 nodekind가 StmtK일 때 CompoundK라면 새로이 scope를 만들어야 하는데 이때 만약 Compound안의 Compound가 있을 수 있고 function이 시작하는 compound가 있을 수 있다. 이를 구별하기 위해서 scope\_cont를 global하게 설정하여 구현하였다. Exp일 때는 IdK이면 st\_lookup을 통해서 그전에 Declaration이 되었는지 파악해야 하며 CallK일 때도 마찬가지다. 마지막으로 DeclK일 때는 Redeclaration이 있는지 st\_lookup을 통해서 파악하며 그렇지 않을 경우 각각의 경우에 따라서 variable,array,function을 구분해서 st\_insert하면 된다. 이때 memory location을 계산해주는데 variable과 function만을 계산한다. 이는 preorder로 해당 TreeNode를 방문할 때마다 static int calcLoc (TreeNode\* t)를 통해서 계산한다.

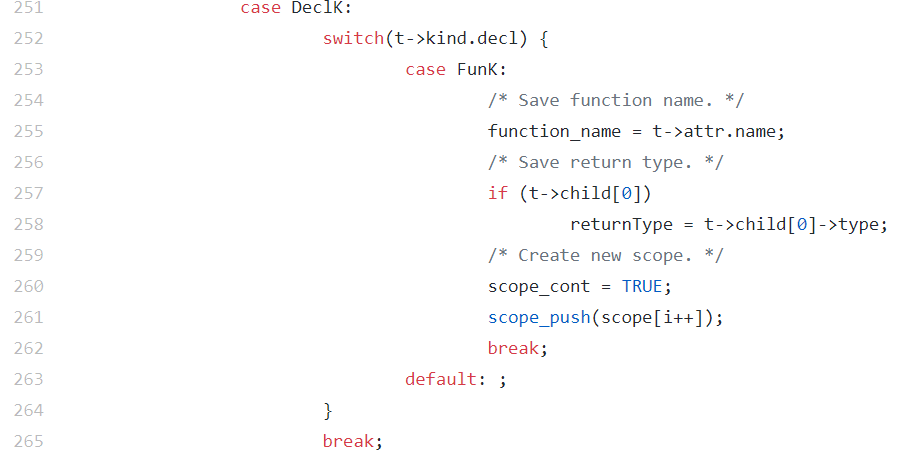
 각각의 member variable인 . varLoc, funLoc, paramLoc 은 initializing은 scope에 따라서 달리 해야 한다. 가장 중요한 것은 varLoc같은 경우 현재 t>level==0이면 0으로 1이면 -4로 둘다 아니면 그전의 t->parent->varLoc를 상속받듯이 가지고 와야 한다.



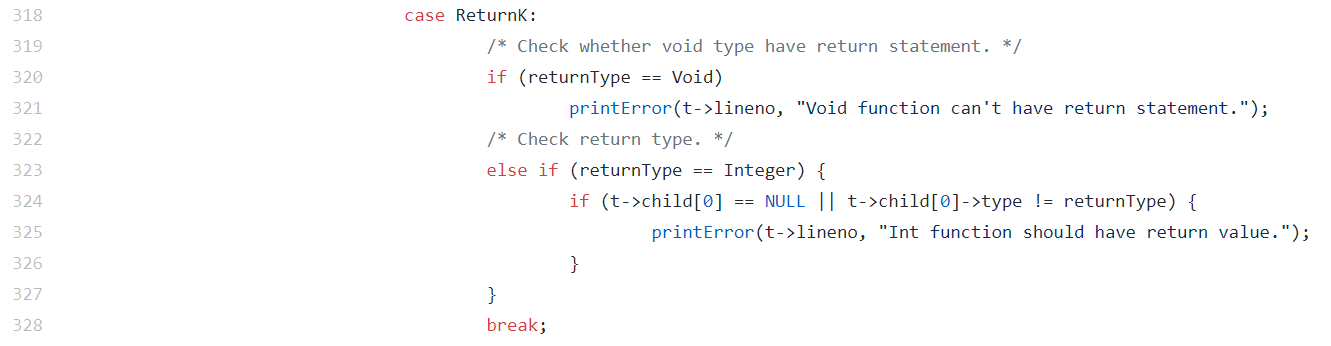
static int calcLoc (TreeNode\* t)에서는 현재 scope\_top()->level==0이면 이는 global이기 때문에 function이면 +1, variable이면 +4, array이면 +4\*arraysize만큼 더해준다. 만약 scope\_top()->level!=0이면 이는 function안의 local variable이기 때문에 global과 마찬가지의 양으로 부호만 반대로 해서 반대로 빼준다. TreeNode를 postorder로 방문하면서는 Compound일 때 scope\_pop()을 해주며 ParamK일 때는 memory location을 계산해준다. parameter만을 postorder로 계산하는 이유는 parameter가 왼쪽부터 stack에 쌓이면서 memory가 assign되기 때문에 그렇다. 그래서 parameter가 모두 stack에 쌓인 다음 fp를 토대로 +4씩하면서 위치를 잡아줄 수 있다.

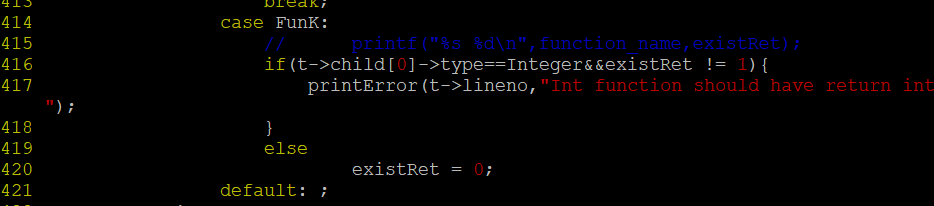
typeCheck를 하는 것은 아래의 함수를 통해서 한다. mainCheck는 main function이 올바르게 void main(void)로 선언이 되어있으며 function의 마지막에 선언이 되어있는지를 check한다. scope\_push(scope[0])을 하는 이유는 global scope을 push하기 위함이다.



typeCheck는 symbol table을 만드는 과정과 달리 postorder에서 작업이 많다. 그 이유는 type을 검사하기 위해서는 이미 자식들의 type에 대한 계산이 완료가 되어야 하기 때문이다. preCheckNode에서는 TreeNode가 CompoundK면 scope\_cont가 false인 경우, 즉 compound안의 compound일 경우에만 scope\_push(scope[i++])를 한다. i는 이전 symbol table을 만들 때의 scope를 처음부터 훑기 위한 index이고 typeCheck에서는 이전 top을 활용하여 계속 push해서 stack을 유지한다. 또한 DeclK이면서 FunK일 때 Return이 semantically 맞는지 검사하기 위해서 function name과 returnType을 지정한다. 그리고 exsitRet를 static하게 유지하여 int function에서 return이 없을 경우를 Error처리 할 수 있게 한다.

postorder의 함수인 checkNode는 크게 3가지로 나눌 수 있다. TreeNode의 nodekind가 StmtK일 때 Compound면 scope\_pop()을 실행하고 ReturnK이면 preorder에서 저장한 function\_name과 returnType으로 검사한다. 이때 int function이 return statement가 없을 경우를 처리하기 위해서 existRet를 설정한다. existRet =1이면 return statement가 존재한다는 의미가 된다.

 그리고 WhileK면 t->child[0]인 expression의 값이 Integer인지 검사한다. 두번째 ExpK는 TreeNode의 kind.exp가 OpK면 left,right operand가 type이 맞는지 검사하고 IdK면 Array의 index가 Integer인지 그리고 Array가 아닌데 Array로 subscript하고 있는지 검사한다. CallK면 functin을 call하는 것이므로 선언이 되어있는지 확인하고 그렇다면 parameter가 올바르게 작성되었는지 int paramCheck(TreeNode\* params, TreeNode\* args)로 검사한다. -1이면 parameter의 개수가 다른 것이고 0이면 type이 다른 것이고 1이면 올바른 문법이라는 뜻이다. 마지막으로 DeclK에서는 VarK인 경우에 type이 Void가 아닌 지 검사하면 된다. FunK일 경우 int function에서 return이 없을 때 Error을 처리하기 위해서 아래처럼 existRet!=1이면 이는 return이 없었다는 것이므로 Error을 출력하면 된다.



## 3. 제작 내용

## C- language로 작성된 test.c를 semantic analyze한 결과이다.

