**Design and Development of Compiler**

**for C-Language**

**Phase 3: Design and Implementation of Semantic Analyzer**

**1. Project3 제안서**

**2. project3 결과 보고서**

**과목명: [CSE4120] 기초 컴파일러 구성**

**담당교수: 서강대학교 컴퓨터공학과 정성원**

**개발자: 2조 이상범(팀장, 20121618)  
모지수(팀원, 20101631)  
박영훈(팀원, 20121591)**

**개발기간: 2015. 05. 22. -2015. 06. 05.**

**Project3 결과 보 고 서**

**프로젝트 제목: Design and Development of Compiler for C-language: Phase 3: Design and Implementation of Semantic Analyzer**

**제출일: 2015. 06. 05.**

**2조 팀원: 이상범(팀장, 20121618), 모지수(20101631), 박영훈(20121591)**

**I. 개발 목표**

* **단계별 제안서 I. 개발 목표 참조**

**II. 개발 범위 및 내용**

**가. 개발 범위**

* **단계별 제안서 II. 개발 범위 참조**

**나. 개발 내용**

* **단계별 제안서 II. 개발 내용 참조**

**III. 추진 일정 및 개발 방법**

**가. 추진 일정**

* **단계별 제안서 III. 추진 일정 참조**

**나. 개발 방법**

* **단계별 제안서 III. 개발 방법 참조**

**다. 연구원 역할 분담**

* **팀:** pass1의 symbol creation과 scoping error check를 구현 및 test case 제작 후 시험 및 보고서 작성
* **이상범:** pass2의 semantic error check
* 변수나 parameter 선언 시 array 변수인지 확인 함
* 변수가 array인 경우 array index가 integer가 아닌 경우
* array가 아닌 변수를 array처럼 사용하려는 경우
* 함수 호출 시 parameter의 개수와 type이 일치하는지 check
* 함수의 return 값과 return type이 일치하는지 check 및 함수의 return type이 void일 경우 return이 없어야 함
* **모지수:** pass2의 semantic error check
* main함수는 가장 마지막에 선언되어야 함
* main함수는 반드시 void로 선언되어야 함
* main함수는 반드시 parameter가 존재하지 않는다.
* loop문의 반복 조건을 표시하는 부분에 오는 statement는 integer 값을 가질 수 있어야 한다.
* **박영훈:** pass2의 semantic error check
* 선언되지 않은 변수나 함수는 사용 불가
* 변수나 함수, 함수 parameter 선언 시 중복된 이름 check
* 변수나 parameter 선언 시 void type으로 선언할 수 없음
* 변수에 값을 assign 하는 경우 type에 대한 check
* 함수 호출 시 호출된 것이 함수가 맞는지 (즉 변수 등이 아니었는지) 확인

**IV. 연구 결과**

1. **합성 내용:**
2. **BNF grammar of C- and bison**

C-에 대한 parse tree를 만들기 위해선 C-의 BNF grammar를 이해하여 bison 문법으로 코드를 구성해야 한다. C-의 BNF grammar는 위의 개발방법에 명시되어 있다.

Bison코드는 header, global variable, tokens, grammar 부분으로 나누어져 있다. Header부분에서는 사용되는 변수나 함수들이 있는 헤더들이 추가되고, global variable부분에서는 grammar에서 쓰일 변수들을 선언해준다. Token부분에서는 grammar에서 쓰이는 enum type과 같은 상수가 정의된다. 마지막으로 grammar 부분에서 cminus에 대한 BNF문법을 정의한다.

1. **Flex usage**

Flex는 어휘를 분석하는 프로그램으로, 파서를 호출하여 입력들을 토큰화하여 토큰들을 파서에 넘겨준다. 사용법은 flex [file name] 이다.



위와 같이 실행하면 flex가 cminus.l file의 어휘들을 분석하여 lex.yy.c. file을 생성해준다. 이번 프로젝트에서는 이 작업을 Makefile에 넣어 한꺼번에 처리해주도록 하였다.

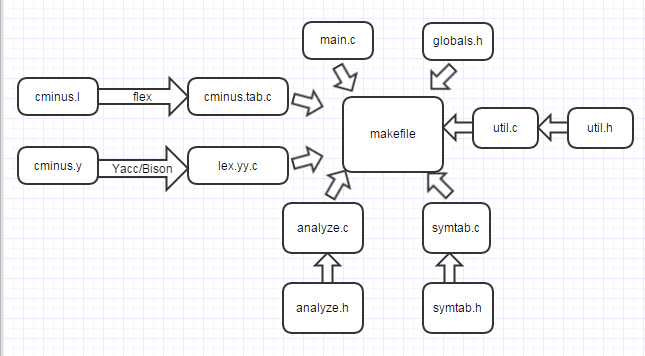
1. **Bison usage**

C-의 BNF 문법이 기술되어 있는 cminus.y file은 bison 문법으로 기술되어 있다. 따라서 이 file을 bison tool을 이용해서 C-의 file로 변환해야 한다. 사용법은 bison –d –v [file name] 이다. –d 옵션은 macro 정의를 포함하는 extra output file을 생성한다. –v 옵션은 output file에 문법과 parser에 대한 설명을 넣는다.



위와 같이 명령어를 치면 c-file과 header-file이 생성되고 output file도 함께 생성된다. 이를 Makefile에 넣어 compile하였다.

1. **프로그램 구성도**



1. **Files**

* **cminus.l:** flex file로 c- language를 위한 lex specification이다.
* **flex:** fast lex로 빠른 lexical analyzer로, flex file(\*.l)을 읽어서 c file(lex.yy.c)을 생성해준다.
* **cminus.y:** Yacc file로 c- language의 BNF 문법을 기술하는 부분이다.
* **Yacc/Bison:** Yet Another Compiler Compiler로, Yacc(\*.y)을 읽어서 c file(\*.tab.c)을 생성해준다. 이 tool을 사용할 때 쓰는 프로그램이 Bison이고 BNF로 작성되어야 한다.
* **globals.h:** 프로그램이 실행되는데 필요한 전역변수들이 선언되어 있다.
* **util.h/util.c:** Abstract Syntax Tree를 출력하는 함수로 project2에서 구현했다.
* **main.c:** program의 main으로 여러 옵션들을 선택 가능하다. 이번 프로젝트에서는 NO\_ANALYZE를 FALSE로 세팅해 analyze를 실행시키고, TraceAnalze를 True로 세팅해 symbol table을 생성한다.
* **symtab.h/symtab.c:** C-는 tiny와 다르게 scope가 여러 개가 존재할 수 있다. 따라서 하나만 있던 bucket을 여러 개 생성할 수 있도록 하고 이들을 연결 및 관리하는 함수들이 필요하다. Bucket은 hash함수를 통해 접근된다. Compound statement이거나 function declaration이면 새로운 scope가 생성되어야 하고 변수들의 offset을 표현하는 값을 초기화해야 한다. 이 scope는 compound statement가 끝나거나 function declaration이 끝날 때 delete한다. 변수나 함수에 대한 정보는 명세서에서 요구하는 정보들과 함께 symbol table에 저장된다. 명세서에서 요구하는 정보들은 아래와 같다.

|  |  |
| --- | --- |
| Location | Offset 정보 |
| Name | 해당 symbol의 이름 |
| Line number | 해당 symbol이 나타난 line number들에 대한 정보 |
| Kind | Variable, parameter, function 중 어느 것인지 |
| Array 여부 | Array인지 아닌지에 대한 정보 |
| Array 크기 | Array의 size |
| int / void | Function의 경우는 return type, 그 외는 해당 symbol의 type |
| Scope | 해당 symbol이 선언된 scope number |

**analyze.h**

* **analyze.h/analyze.c:** 2번의 traverse를 구현 하는데 pass 1에서는 실제로 symbol table을 작성하고 scope error 체크한다. Pass2에서는 type checking 및 다음의 sematic error를 체크한 뒤 에러 출력을 해준다.

․ 선언되지 않은 변수나 함수는 사용 불가

․ 변수나 함수, 함수 parameter 선언 시 중복된 이름 check

․ 변수나 parameter 선언 시 void type으로 선언할 수 없음.

․ 변수나 parameter 선언 시 array 변수인지 확인해야 함.

․ 변수에 값을 assign 하는 경우 type에 대한 check

․ 변수가 array인 경우 array index가 int가 아닌 경우

․ array가 아닌 변수를 array처럼 사용하려 하는 경우

․ 함수 호출 시 parameter의 개수와 type이 일치하는지 check

․ 함수 호출 시 호출된 것이 함수가 맞는지 (즉 변수 등이 아니었는지) 확인해야

․ 함수의 return 값과 return type이 일치하는지 check

함수의 return type이 void일 경우 return이 없어야 함

․ main함수는 가장 마지막에 선언되어야 함

․ main함수는 반드시 void로 선언되어야 함

․ main함수는 반드시 parameter가 존재하지 않는다.

․ loop문의 반복 조건을 표시하는 부분에 오는 statement는 int값을 가질 수

있어야 한다.

* **project3\_2:** 최종 실행 파일로 input file(\*.c)을 받아서 output(symbol table 및 scope error 및 semantic error)을 linux 화면에 프린트 한다.

1. **분석 내용:**
   * **Symtab.c**

Symbol table을 구성하고, 접근하고 출력하는 함수들이 있다. 이 함수들은 모두 analyze.c에서 호출된다. Scope들은 stack형식으로 관리가 된다.

프로그램이 시작되면 st\_init() 함수가 symbol table을 구성하기 위한 linked list의 head들의 공간을 잡고 초기화를 한다. 그 다음 global영역을 위한 scope를 insert하고 input함수와 output함수를 해당 scope에 insert한다.

scope\_insert() 함수에서는 새로운 scope 공간을 잡고 scope에 대한 정보를 set하고scope를 관리하는 linked list에 넣는다. scope\_delete() 함수에서는 stack의 맨 위에 있는 scope를 삭제한다.

st\_insert() 함수에서는 parameter로 bucket에 대한 정보들을 가져온다. Scope에 넣고자 하는 name이 이미 symbol table에 존재한다면 line number만 추가하고, 그렇지 않으면 공간을 잡아 이름을 hashing하여 hashTable에 넣는다. 여기서 name이 해당 scope에 존재하는지 확인하는 함수는 st\_lookup()이다. Name이 stack의 top에 있는 scope에 존재한다면 해당 offset을 return하고, 존재하지 않으면 -1을 return한다.

scope\_check() 함수에서는 해당 name이 어느 scope에 선언되어 있는지를 알려준다. 존재하면 scope의 번호를 return하고, 존재하지 않으면 -1을 return한다.

printSymTab() 함수에서는 stack의 top의 scope에 있는 모든 bucket들을 명세서에 맞게 출력을 한다.

위의 함수들은 pass1에서 사용되는 함수들이다. Pass1에서는 scope가 다 삭제되어 사용할 수 없다. 따라서 scope\_delete() 함수에서 삭제되는 scope들을 pass2에서 사용되는 linked list에 insert한다. 또한 scope number로 scope를 구별하면 중복되는 scope number가 존재할 수 있다. 따라서 insert할 때 scope마다 고유의 scope number를 저장한다.

isArray() 함수에서는 parameter로 name과 scope의 고유 number를 가져온다. 그 다음 linked list에서 해당 scope number를 가진 scope를 찾는다. 만약 그 scope에 name이 존재하면 그것이 array이면 1을 아니면 0을 return하고, 존재하지 않는다면 고유 scope number가 아닌 scope number를 가져와서 1씩 감소시키면서 해당 scope number와 같은 scope를 찾는다. 이는 scope를 넣는 방식이 stack이므로 문제가 생기지 않는다.

isFunc() 함수는 isArray와 같은 방식으로 name이 있는 scope를 찾아 해당 name이 함수이면 1을, 아니면 0을 return한다. getType() 함수도 같은 방식으로 name의 type을 return한다.

* + **Analyze.c:**

총 2번의 traverse를 거치면서 Symbol table creation과 typeCheck를 한다. 먼저 첫번째 Pass에서는 buildSymtab() 함수를 호출함으로써 Symbol table을 생성한다. buildSymtab() 함수 에서는 st\_init()을 호출 함으로써 scope bucket list를 초기화 한다. 이를 구현한 함수가 insertNode()인데 parameter로 넘어오는 TreeNode를 각각의 case별로 구현하는데, Declaration 부분에서는 symbol table에 있는지 확인한 후 있으면 st\_insert() 함수를 호출해 scope bucjet list에 추가하고 만약 이미 symbol table에 존재한다면 중복선언 에러를 출력해준다. Expression 부분에서는 함수 호출과 ID가 참조될 때 사용가능한 범위의 scope에 존재하는지 확인 후 만약 그 변수를 참조할 수 없으면 Scope Error를 출력하고 아니라면 symbol table에서 참조된 변수나 호출된 함수 이름을 찾아 line number를 추가해준다.

두번째 Pass에서는 typeCheck() 함수를 호출함으로써 typeCheck 및 Semantic Error를 체크한다. 이를 구현한 함수가 checkNode()인데 parameter로 넘어오는 TreeNode를 각각의 case별로 구현을 한다.

Declaration 부분에서는 변수, 함수, 파라미터, 배열 파라미터의 4가지 경우로 나뉜다. 먼저 변수 선언 일 경우에는 type이 void의 경우인지 체크해 void로 선언된 변수인지 확인한다. 다음으로 함수 선언 일 경우에는 함수의 계속적인 호출을 대비해 stack 형식으로 최대 100개의 call stack을 관리 할 수 있게 필요한 포인터들을 setting해준다. 그 후 다른 node에서 체크하기 쉽게 return이 integer인지 void인지 구별해주는 flag, return을 호출 한 적이 있는지 구별해주는 flag, main 함수인지 구별해주는 flag를 관리해준다. 추가로 만약 main() 함수가 선언된 부분이라면 return type이 void인지 확인하고 parameter 부분도 void인지 확인하고 아까 설정한 main\_flag를 체크함으로써 마지막으로 선언된 함수인지 체크한다. 다음으로 변수 parameter나 배열 parameter가 선언된 경우에는 parameter를 관리하는 변수에 구분해서 추가하고 선언된 parameter들이 integer 타입인지 체크한다.

그 다음 statement 부분에서는 compound, selection, iteration, return 부분이 있다. 먼저 compound 부분에서는 각 compound에 진입할 때 마다 scope\_number\_for\_type 값을 증가시켜 각 scope symbol table에 고유의 식별자를 준다. Selection이나 itertation은 조건문이 integer값인지 확인한 후 그 외의 경우에는 Error를 출력해준다. 마지막으로 return 부분에서는 함수의 return타입이 integer인데 return이 없는 경우(return ;)나 return타입이 void인데 return이 있는 경우의 Error를 출력해준다.

마지막으로 expression부분에서는 operation, constant, identifier, assign, call, array identifier부분이 있다. operation부분에서는 현재 stack 에서 operator 개수를 관리한다. 다음으로 constant 부분에서는 현재 호출될 때 넘어온 argument의 위치를 나타내고 있는 ptr2를 이용해 가리키는 배열의 값이 선언된 parameter를 관리하는 배열의 위치보다 뒤에 있으면 Error를 출력해주고 종료시에 parameter를 관리하는 배열의 위치가 더 뒤에 있을 때도 Error를 출력해준다. Identifier 부분도 constant와 비슷하게 구현되었는데 추가로 배열인자를 받아야 하는데 변수가 넘어오거나 변수를 받아야 하는데 배열인자가 넘어올 때 Error를 출력해준다. 추가로 만약 array 이름 일 경우 index가 integer가 아니면 Error를 출력해준다. 또 추가로 assign flag가 세팅된 상태라면 모든 변수가 integer이어야 하므로 integer가 아닌 identifier가 한 개라도 나타나면 그 즉시 Error를 출력해 준다. 그 다음 assign 부분에서는 현재 부분부터 assign이 시작된다는 assign\_flag를 세팅하고 getType() 함수를 호출해 integer type인지 확인한다. Array identifier 부분은 이 name이 array로 선언된 name인지 확인하는 isArray() 함수를 호출해 제대로 참조했는지 확인한다. 마지막으로 call 부분에서는 현재 호출된 함수의 이름을 저장한 funcs[]에서 찾아 cur[]이라는 stack pointer에 함수 고유 번호를 저장한다. 함수도 마찬가지로 return 값이 integer인지 확인해서 만약 void인 경우에 assign이나 array index로 참조하려는 경우가 나타나면 Error를 출력해준다. 또한 isFunc() 함수를 호출해 현재 호출한 함수가 symbol table에 존재하는 함수인지도 확인한다.

1. **제작 내용:**
   * **Symtab.c**

**1) 함수**

* static int hash ( char \* key )

Key의 각 byte마다의 값을 더해 hashTable의 크기인 SIZE로 modulo연산을 한 값을 return한다.

* void st\_init()

Pass1에서 scope를 저장하는 linked list의 head인 ScopeHead와 pass2에서 scope를 저장하는 linked list의 head인 ScopeTypeHead의 변수의 공간을 잡고 다음 scope를 가리키는 next를 NULL로 한다. 그리고 ScopeHead에 scope를 insert하고 C-에서 사용되는 global function인 input(), output() 함수를 insert한다.

* void scope\_insert()

ScopeList type의 임시변수에 공간을 할당하여 scope number와 고유 scope number를 set하고 ScopeHead에 inert한다.

* void scope\_delete()

ScopeHead의 top의 scope를 printSymTab을 이용하여 출력하고 linked list에서 제거한다. Pass2에서 해당 scope를 다시 사용하기 위해서 ScopeTypeHead에 inert한다.

* void st\_insert( char \* name, int lineno, int loc, int kind, int is\_array, int array\_size, int type, int scope, char \*func\_name)

name : symbol의 이름

lineno : symbol이 사용된 line number

loc : scope에서의 offset

kind : variable, function, parameter중 어느 것인지의 정보

is\_array : array인지에 대한 정보

array\_size : array의 크기

type : int/void

scope : scope number

func\_name : 선언된 함수의 name

ScopeHead의 top의 hashTable에 name이 있는지 확인한다. 존재하면 bucket을 위한 새로운 공간을 잡아 parameter에 대한 정보들을 저장하고 hashTable에 insert한다. 존재하지 않는다면 line number만 추가한다.

* int st\_lookup(char \* name)

name을 hash를 이용하여 key값을 받아 ScopeHead의 top의 hashTable에 존재하는지 확인한다. 존재하면 해당 name의 offset을 return하고 존재하지 않으면 -1을 return한다.

* int scope\_check(char \*name, int lineno)

name을 hash를 이용하여 key값을 받아 Scopehead의 top의 hashTable에 존재하는지 확인한다. 존재하면 해당 name의 scope를 return하고 존재하지 않으면 -1을 return한다.

* int isArray(char \*name, int scope\_number\_for\_type)

name을 hash를 이용하여 key값을 받아 저장한다. 그 다음 scope의 고유 number와 같은 scope를 ScopeTypeHead에서 찾는다. 해당 scope에 name이 존재하면 is\_array값을 리턴하고 그렇지 않으면 bucket에서 scope의 number를 가져와 scope가 0보다 크거나 같을 때까지 찾는다.

* int isFunc(char \*name, int scope\_number\_for\_type, int \*type)

위의 isArray와 같은 방식으로 name이 있는 scope를 찾아 bucket에서 scope의kind를 return한다.

* int getType(char \*name, int scope\_number\_for\_type

위의 isArray와 같은 방식으로 name이 있는 scope를 찾아 bucket에서 scope의 type을 return한다.

**2) 변수**

* void printSymTab(FILE \* listing)

ScopeHead의 top에 있는 scope의 모든 bucket들을 출력한다. hashTable의 size인 SIZE만큼 loop를 돌면서 hashTable에 접근한다. 각 hashTable에는 여러 bucket들이 존재할 수 있으므로 한번 더 loop를 bucket이 null일 때까지 돈다. 출력양식은 아래와 같다.



* FILE\* listing

내용을 출력할 file이다. 이 프로그램에선 stdout에 출력된다.

* typedef struct LineListRec {

int lineno;

struct LineListRec \* next;

} \* LineList;

Line number들을 저장하는 구조체이다. Linked list의 자료구조를 사용하여 line number들이 저장된다.

* int scope\_number, scope\_number\_for\_type

scope\_number는 상대적인 scope number이고, scope\_number\_for\_type은 절대적인 scope number이다. 예를 들면 scope1, 2가 있는데 2가 끝나서 삭제되고 새로운 scope가 들어올 때 그 scope의 number도 2가 된다. 이는 상대적인 scope number이고, 절대적인 scope number는 unique하다. Scope를 구별하기 위한 용도이다.

* typedef struct BucketListRec

{ char \* name;

LineList lines;

int memloc ; /\* memory location for variable \*/

int kind; /\*변수인지 0 함수인지 1 파라미터인지 2 \*/

int is\_array;

int array\_size;

int type;

int scope;

char \*func\_name;

struct BucketListRec \*next;

} \* BucketList;

Symbol을 저장하기 위한 구조체이다. Symbol table과 pass1, pass2에서 필요한 변수들이 있다. 같은 hash의 key값을 가질 수 있으므로 linked list로 다음 symbol을 가리키게 한다. 변수들의 내용은 위의 st\_insert와 동일하므로 생략하겠다.

* typedef struct ScopeListRec

{

int scope;

int scope\_for\_type;

BucketList hashTable[SIZE];

struct ScopeListRec \*next;

} ScopeList;

Scope를 관리하기 위한 구조체이다. Scope의 상대적인 number와 절대적인 number를 저장한다. 또한 symbol들을 저장하기 위한 hashTable도 이곳에 선언되어 있다. Scope 역시 Linked list로 연결되어 있다.

* static ScopeList \*ScopeHead, \*ScopeTypeHead

Pass1에서 사용되는 scope들을 관리하기 위한 head인 ScopeHead와 pass2에서 사용되는 scope들을 관리하기 위한 head인 ScopeTypeHead이다.

* + **Analyze.c:**

1. **변수**

* int location: symbol table에서의 offset
* int temp\_flag: function declaration후에 compound statement가 호출이 되는 경우에 scope\_level이 중복해서 증가하는 경우를 막는 flag
* int pass\_flag: 현재 상태가 pass1인지 pass2인지 구별하는 flag
* int main\_flag: 현재 선언된 함수가 main 함수인지 구별하는 flag
* int expr\_flag : 현재 while문이나 if문의 조건문(expression) 안에 있으면 1, 그렇지 않을 경우 0을 값으로 갖는 flag.
* int scope\_number\_for\_type: scope가 같을 때 구별하기 위한 고유 number
* char\* func\_name: 현재 호출하거나 선언된 함수의 이름을 저장하기 위한 변수
* int return\_flag: return 타입이 integer인지 void인지 구별하는 flag
* int int\_but\_not: return 타입이 integer인 경우에 return statement가 호출된 적이 있는지 체크하는 flag
* unsigned char params[][]: 함수가 선언될 때 각 함수의 parameter들의 타입을 순서대로 저장하는 변수
* char\* funcs[]: 함수의 이름들을 고유 번호로 저장하기 위한 변수
* int ptr: params[][]의 첫 번째 index에 사용되는 pointer로 몇 번째 함수인지 나타내는 포인터
* int ptr2[]: params[][]의 두 번째 index에 사용되는 pointer로 stack call일 경우를 대비해 배열로 잡았다. 몇 번째 parameter인지 나타내는 포인터
* int cur[]: 현재 호출한 함수가 params[][]의 첫 번째 index에서 몇 번째 위치인지 저장한 포인터로 stack call일 경우를 대비해 마찬가지로 배열로 잡았다. 몇 번째 함수인지 나타내는 포인터
* int prev\_lineno: 이전 line number을 임시로 저장하기 위한 변수
* int array\_flag: arrayIdK가 호출된 적이 있는지 구별해주는 flag
* int assign\_flag: assignK가 호출된 적이 있는지 구별해주는 flag
* int operator\_cnt[]: operator가 이번 statement에서 몇 개가 호출 됬는지 카운트 하는 변수로 stack call을 대비해 배열로 잡았다.
* int sp: stack call이 발생했을 때, 몇 번째 stack call인지 나타내는 포인터

1. **함수**

* Static void traverse(TreeNode \* t, void (\* preProc) (TreeNode \*), void (\* postProc) (TreeNode \*))

Abstract Syntax Tree를 순회하는 함수로 인자에 따라 pre-order이나 post-order로 선택할 수 있다. Pass1 and pass2 모두 pre-order로 순회하면서 child를 먼저 순차적으로 순회한 뒤 sibling을 따라 순회한다. 순회하면서 만약 현재 tree node가 AssignK이면 child를 순회 하기 전에 flag를 세팅해 표시해 놓고 childe를 다 순회한 뒤 assign\_flag를 0으로 clear함으로써 assign statement가 끝남을 표시한다. 또 만약 pass\_flag가 1로 setting되어 있다면 pass1 단계라는 것이고 만약 현재 tree node가 CompoundK이면 compound statement가 끝남을 나타낸 것이므로 scope\_delete() 함수를 호출해 현재 scope bucket list를 삭제한다.

* Static void insertNode(TreeNode \*t)

Tree node t를 parameter로 받아 그 내용을 실제로 symbol table을 구현 및 scope error를 체크한다.

먼저 tree node t가 Declaration부분이면 4가지 경우로 나누어서 symbol table을 구현할 수 있다.

먼저 kind.decl 값이 VarK라면 st\_lookup() 함수를 호출해 symbol table에 이미 존재하는 name인지 확인한 뒤 만약 존재하지 않는 변수라면 st\_insert() 함수를 호출해 symbol table에 t에 대한 정보를 저장한다. 만약 이미 존재하는 name이라면 중복 선언이므로 semantic error에서 “변수나 함수, 함수 파라미터 선언 시 중복된 이름 check” Semantic Error를 출력한다.

다음으로 kind.decl 값이 FuncK라면 VarK와 마찬가지로 구현한다. 추가로 새로운 nested scope가 생겼다는 의미이므로 scope\_insert() 함수를 호출해 scope\_bucketlist를 하나 더 생성해 바로 scope\_head의 next로 연결한다. 마찬가지 이유로 location을 0으로 초기화하고 compound statement와 겹쳐서 중복해서 scope\_bucketlist를 생성하는 경우를 막기 위해 temp\_flag도 0으로 clear한다.

다음으로 kind.decl 값이 ParaK라면 VarK와 마찬가지로 구현한다.

마지막으로 kind.decl 값이 ArrayParaK라면 VarK와 마찬지로 구현한다.

다음으로 tree node t가 Statement 부분이면 compound statement을 확인한다. 이는 새로운 scope가 생성되었다는 의미이므로 함수 선언 시와 마찬가지로 scope\_insert() 함수를 호출하고 location을 0으로 초기화 한다.

마지막으로 tree node t가 Expression 부분이면 3가지 경우로 나누어서 symbol table을 구현할 수 있다.

먼저 kind.exp 값이 IdK라면 scope\_check() 함수를 호출해 참조할 수 있는 변수인지 확인한 뒤 s 가 -1 즉, 참조할 수 없는 scope에 있다면 Scope Error를 출력하고 반대로 참조 가능한 scope에 있는 변수라면 st\_insert() 함수를 호출해 line number만 symbol table에 추가해준다.

다음으로 kind.exp 값이 CallK라면 IdK와 마찬가지로 구현한다.

마지막으로 kind.decl 값이 ArrayIdK라면 IdK와 마찬가지로 구현한다.

* buildSymtab(TreeNode \* syntaxTree)

pass1을 실행하는 부분으로 실제 symbol table을 구현하는 함수이다. st\_init() 함수를 호출해 scope\_bucketlist 들의 값을 초기 setting한다. 이때 location값이 2부터 시작하는 이유는 초기화 할 때 기본적으로 c- language에서 input() 함수와 output()함수를 내장하고 있으므로 미리 symbol table에 추가하고 location을 2부터 시작한다. 그 후 traverse()함수를 호출해 Abstract Syntax Tree를 pre-order로 순회한다. 그 후 마지막 global 지역인 0번째 scope\_bucketlist를 delete하기 위해 scope\_delete() 함수를 호출하고 printSymTab() 함수를 호출해 화면에 symbol table을 출력해 준다.

* static void checkNode(TreeNode \* t)

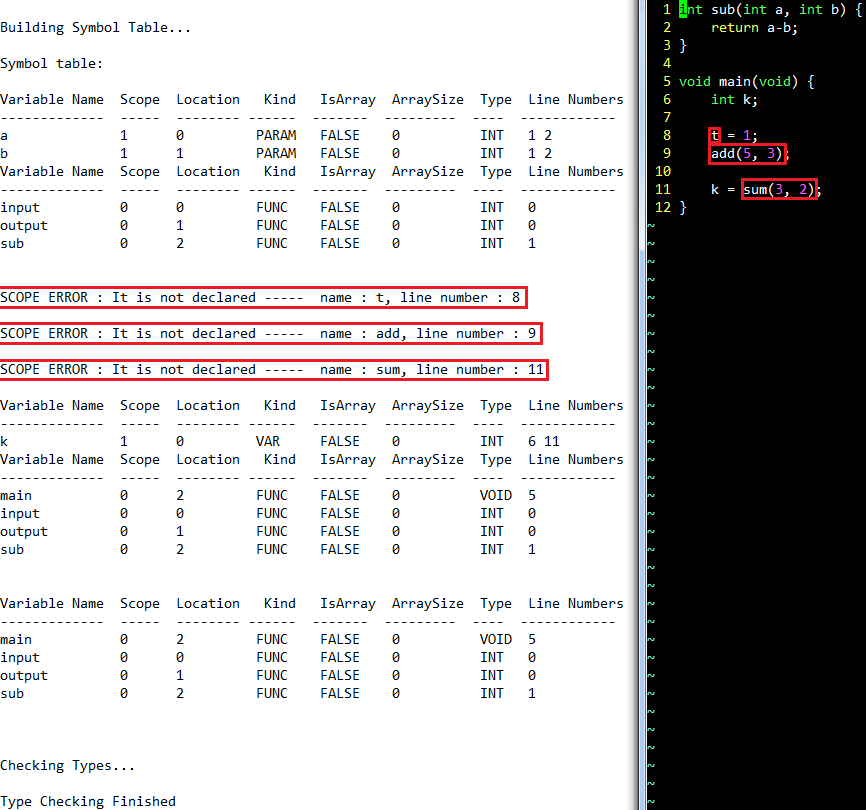
Tree node t를 parameter로 받아 type check와 semantic error들에 대해서 check한다.   
- Iteration, Selection 의 expression에 return type이 void인 함수 call이 올 수 없다. : AST의 현재 node kind가 statement의 iteration/selection kind 인 경우 static int 전역변수인 expr\_flag를 1로 set하고, 그 외의 경우 0으로 set한다. expr\_flag가 1인 동안 AST의 현재 node kind가 expression의 call kind인 경우, isFunc()의 리턴값 함수의 return type을 조사한다. 리턴값이 0인 경우, 즉 함수의 return type이 void인 경우 에러 처리한다.

* void typeCheck(TreeNode \* syntaxTree)

pass\_flag를 0으로 clear함으로써서 pass2인걸 표시하고 cur[] 변수를 -1로 초기화 하고 traverse()함수를 호출해 Abstract Syntax Tree를 pre-order로 순회한다. 마지막으로 int\_but\_not이 1이라면 마지막 함수 진입한 후 아직 return statement가 나오지 않았다는 의미이다. 따라서 “함수의 return type값과 return type check” 에서 Semantic Error를 출력한다.

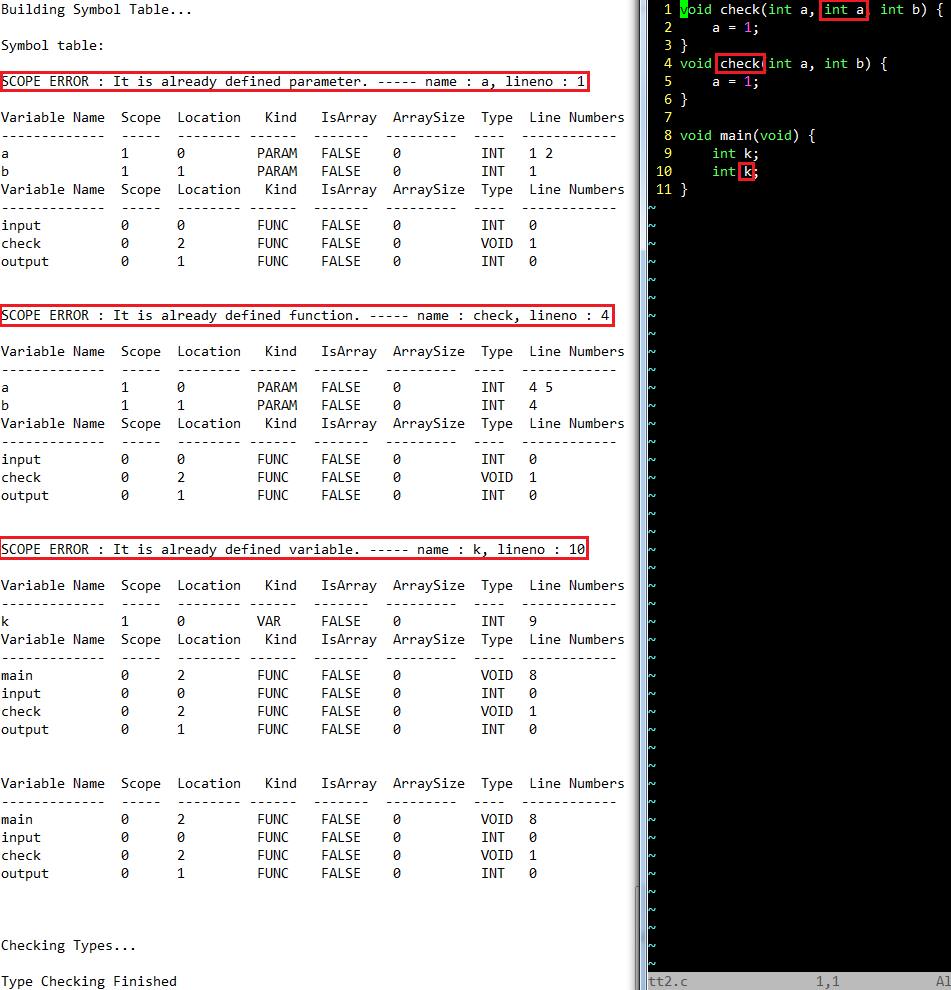
**4. 시험 내용:** 자신이 개발한 소프트웨어를 자체 평가하기 위한 기준 및 수행한 평가 방법, 그 리고 그 것에 기반을 둔 평가 결과를 상세히 기술하라. (반드시 평가 점수가 높은 것이 좋은 것이 아니라 자신이 설정한 방법이 문제점을 찾는데 도움이 되었다는 것을 보이는 것이 더 중요함.) 캡쳐 포함

1. 선언되지 않은 변수나 함수는 사용 불가



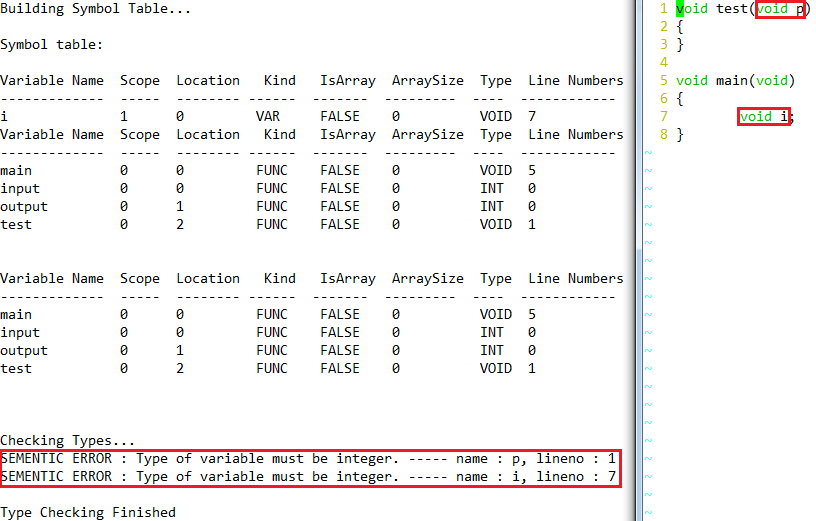
선언되지 않은 변수 t, 함수 add, sum에 대한 scope error를 올바르게 출력한다.

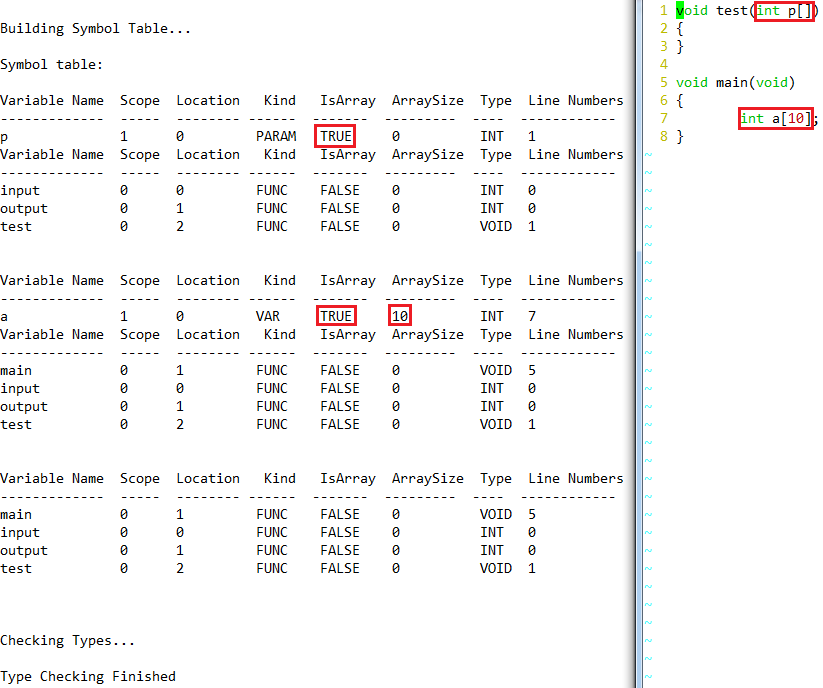
2. 변수나 함수, 함수 파라미터 선언 시 중복된 이름 check

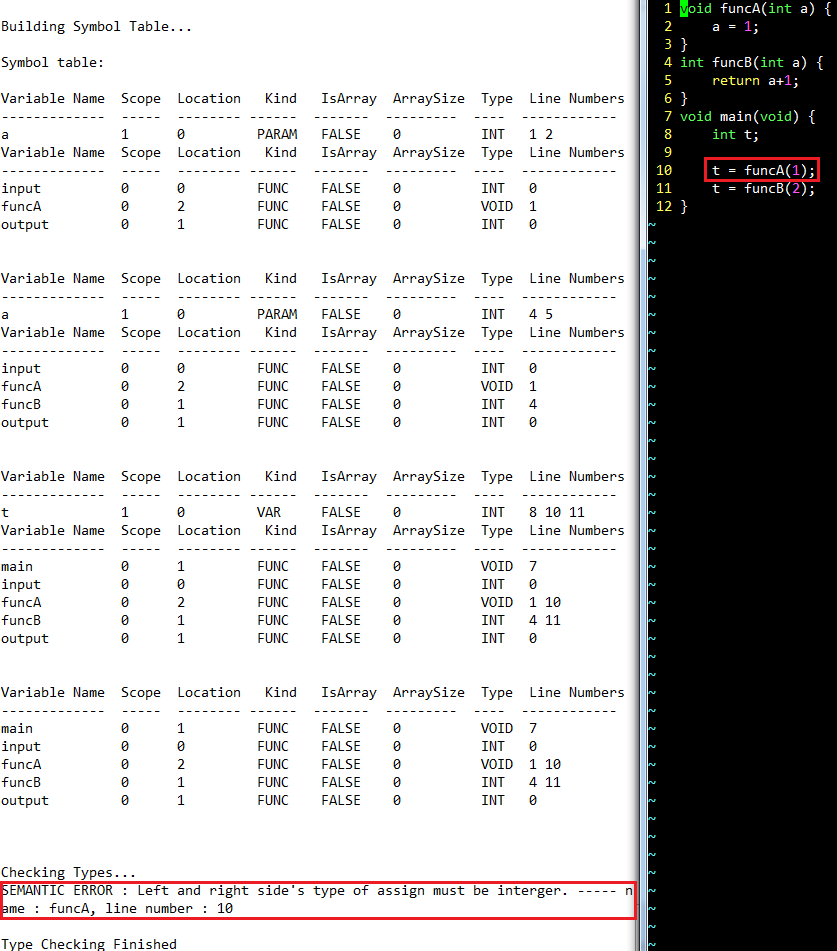


check()의 parameter인 int a가 중복 선언되었고 함수 check(), 그리고 main의 int k가 중복 선언되었다. 이에 대한 에러 메시지를 올바르게 출력한다.

3. 변수나 parameter 선언 시 void type으로 선언할 수 없음

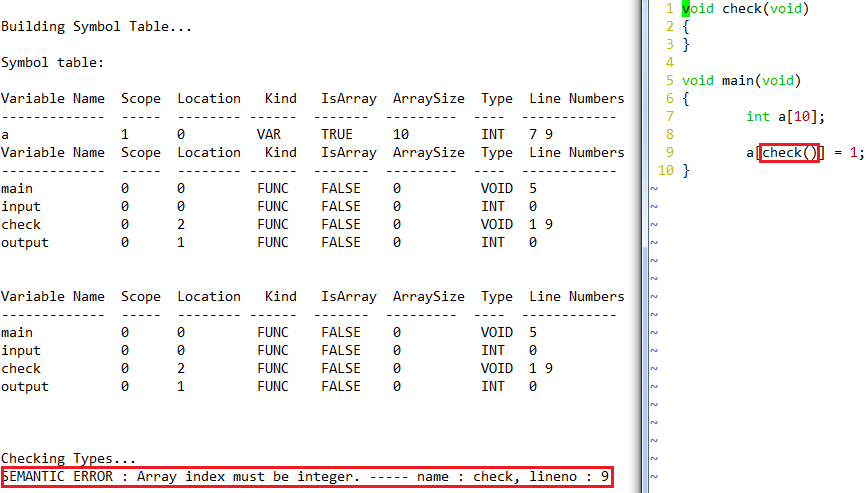


함수 test의 parameter p와 main의 변수 i를 void로 선언하였다. 위와 같이 올바른 에러메시지를 출력하였다.   
  
4. 변수나 parameter 선언 시 array 변수인지 확인해야 함  
  
 함수 test의 parameter를 integer array로, main의 변수 a를 size 10인 integer array로 선언하였다. 이를 위에서 출력된 symbol table에서 확인 할 수 있다.  
5. 변수에 값을 assign 하는 경우 type에 대한 check



Integer 변수 t에 void return value funcA(1)를 assign 하는 경우 올바른 에러 메시지를 출력한다.

6. 변수가 array인 경우 array index가 int가 아닌 경우



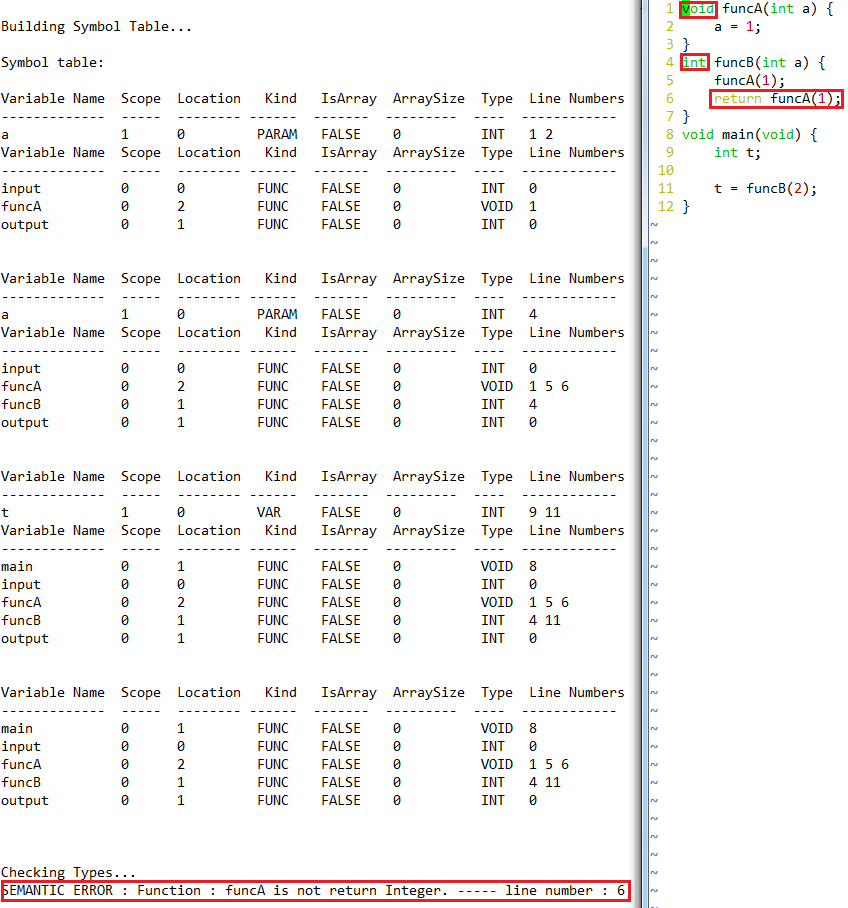
Integer array a[] 의 index에 함수 check()의 void type return value가 쓰인 에러이다. 이에 대한 에러 메시지를 출력한다.

7. array가 아닌 변수를 array처럼 사용하려 하는 경우

8. 함수 호출 시 parameter의 개수와 type이 일치하는지 check

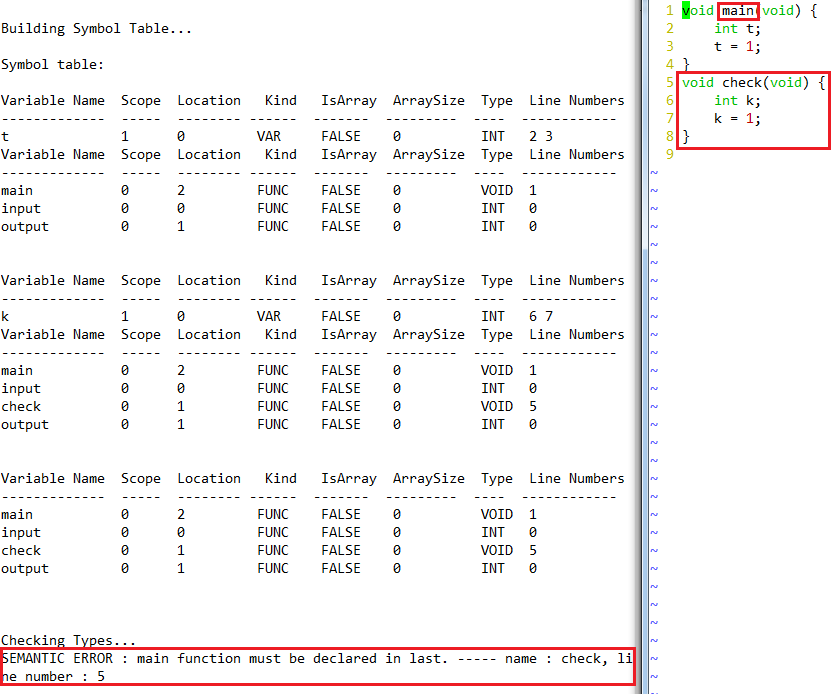
9. 함수 호출 시 호출된 것이 함수가 맞는지 (즉 변수 등이 아니었는지) 확인해야

10. 함수의 return 값과 return type이 일치하는지 check. 함수의 return type이 void일 경우 return이 없어야 함



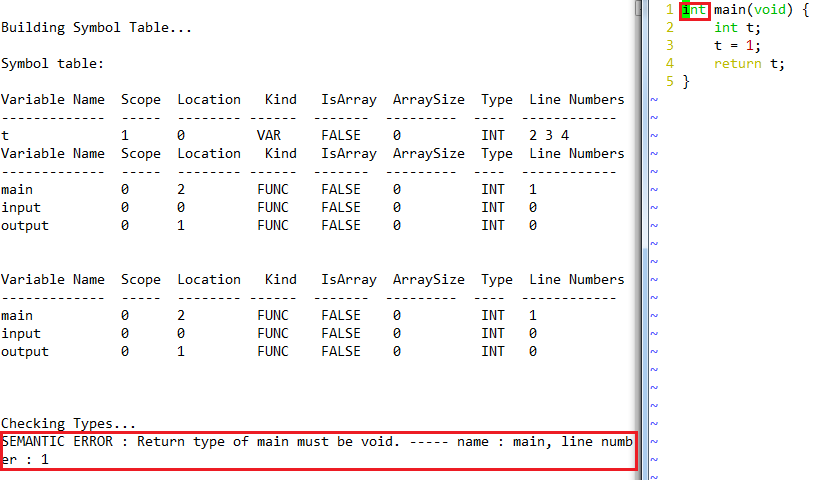
함수 funcB의 return type은 integer이다. 그러나 void type을 return value로 갖는 funcA(1)를 return value로 사용하게 되어 여기서 에러가 발생해야 한다. 이에 대한 에러 메시지를 출력한다.

11. main함수는 가장 마지막에 선언되어야 함

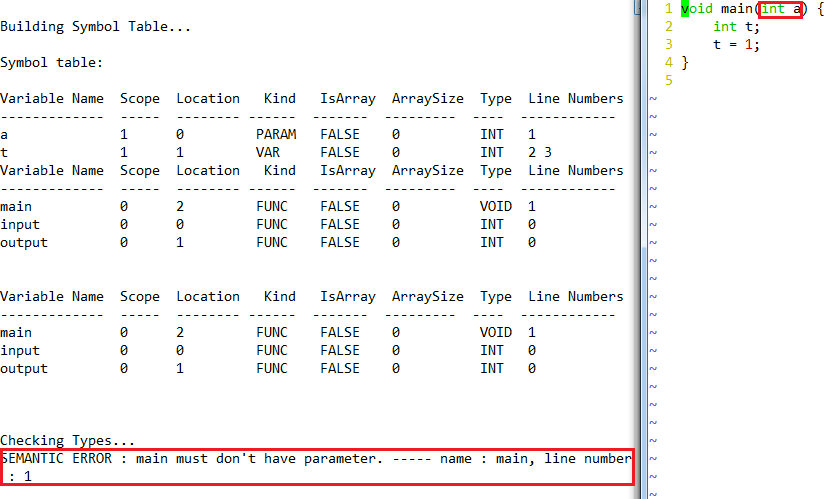


함수 check()가 main 뒤에 선언되었다. 이에 대한 에러 메시지를 위와 같이 출력한다.

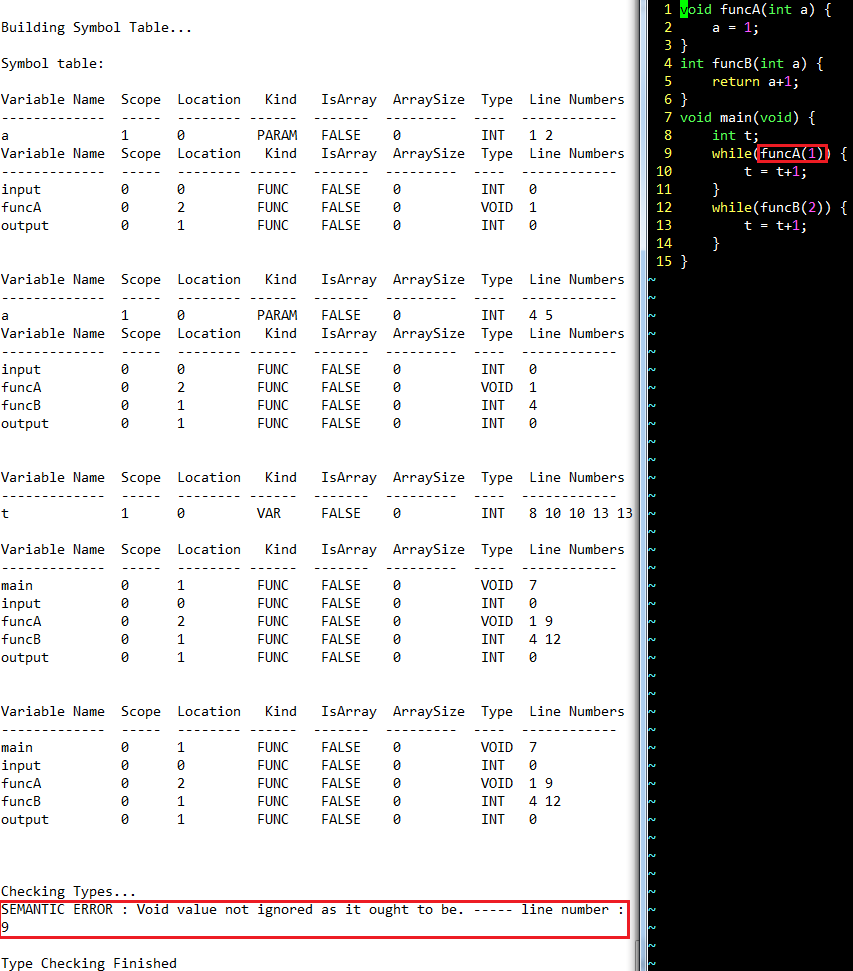
12. main함수는 반드시 void로 선언되어야 함



Main 함수의 return value type이 integer로 선언되었다. 이 에러에 대한 메시지를 출력한다.

13. main함수는 반드시 parameter가 존재하지 않는다.  


Main이 int a를 parameter로 갖는 에러가 있는 코드이다. 이에 대한 에러메시지를 출력한다.

14. loop문의 반복 조건을 표시하는 부분에 오는 statement는 int값을 가질 수 있어야 한다.  


Main의 첫 번째 while문 내 expression에 funcA에 void return value가 들어가있다. 이로 인한 에러 메시지가 출력된다.

**5. 평가 내용**

이 소프트웨어의 장점은 문법적 error가 없는 C- code를 analyze할 때 error없이 올바르게 분석한다는 것이다. 또한 scope, line number, symbol들이 아무리 많아도 linked list로 저장하기 때문에 거의 제한 없이 저장할 수 있다. 재귀함수도 올바르게 처리한다. 따라서 실행한 testcase와 같은 경우라면 안정성은 갖추고 있다.

단점은 많은 testcase에 대해 실행해보지 않아 신뢰성이 없다. 물론 엄청 많은 testcase를 넣어보았지만 예상치 못한 testcase가 들어왔을 경우 과연 이 프로그램이 잘 처리할 지는 알지 못한다.

Flex 프로그램과 bison은 open software이기 때문에 경제적으로 문제가 없다. 또한 만든 프로그램은 단순히 input을 LALR(1) parse tree를 생성하여 symbol table을 생성하고 semantic error들을 확인하는 것이기 때문에 윤리성과 사회적인 문제도 없다.

**V. 기타**

**1. 연구 조원 기여도:**

**- 이상범: 33.33%**

**- 모지수: 33.33%**

**- 박영훈: 33.33%**

**2. 자체 평가:**

프로젝트 기획 단계부터 철저한 구현 계획을 세웠다. 단순한 양적 역할 분담이 아닌, 프로젝트 개발 내용에 대한 팀원 모두의 정확한 이해를 바탕으로 한 효율적인 분담을 이행 -알고리즘 개발의 기반이 되는 자료구조를 먼저 함께 개발하였고, semantic error check의 개발 요소를 개발 범위와 유형에 따라 적절히 분담하였다- 하였다.  
 개발 기간 동안 크게 Scope 자료구조 구성/관리 문제와 C- 에 존재하는 orthogonal한 요소들(반복되는 함수 call, array index내 expression 등)을 구현하는 데에 문제가 발생했는데, 팀원간 실시간 브레인스토밍과 그에 대한 피드백을 이용하는 등 팀 프로젝트의 장점이 될 수 있는 부분을 적극 활용하였다.  
 구현한 semantic analyzer는 함수들의 기능별 적절한 모듈화와 매크로를 통해 뛰어난 코드 가독성과 유지/보수성을 가진다. 또한 자체 개발한 18여개의 다양한 test case를 이용하여 검증 과정을 거쳤으므로, 그 안정성을 보장할 수 있다.

**3. 느낀점:**

* **이상범:** 처음 project description을 읽고 tiny analyzer를 분석했을 때, 생각보다 종류만 잘 분류하면 symbol table이나 error check등을 쉽게 할 수 있을 거라 생각했는데 c- language의 scope 출력부터 헤매서 생각보다 오래 걸렸다. Semantic error check 부분도 단순한 부분 외에 함수의 계속되는 call이나 array를 index를 통해 들어가면 type이 integer가 되는 등 생각하면 생각 할수록 복잡한 예외가 자꾸 자꾸 생겨 힘들고 디버깅하는데 애 좀 먹었다. 그러나 팀 프로젝트였기 때문에 각자의 아이디어가 잘 발휘해 복잡한 케이스까지 잘 analyze한 것 같아서 뿌듯하다.
* **모지수:** tiny와는 가장 큰 C-의 차이점이자 프로젝트의 초기작업인 scope 자료구조 구성을 팀원들과 함께 한 것이, 분담한 semantic 에러 처리를 구현을 비교적 수월하게 할 수 있도록 하는데 도움이 되었다. 처음엔 세 명이 팀이 되어 구현을 하는 것의 비효율적인 면을 걱정했는데, 충분한 의사소통으로 프로젝트를 효율적으로 마무리 할 수 있었다.
* **박영훈:** 수업을 열심히 들어서 쉽게 해결될 것 같았지만 생각했던 것과는 다르게 많이 애를 먹었다. Pass1에서는 symbol table을 올바르게 구성하는 것이 가장 힘들었다. Pass2에서는 반복적으로 들어오는 assign이나 expression을 해결하는 것이 힘들었다. 많은 삽질을 하다가 결국 traverse부분에서 해결하였다.