Design and Development of Compiler

for C- Language

Phase 4: Design and Implementation of Code Generator

과목명: [CSE4120] 기초 컴파일러 구성

담당교수: 서강대학교 컴퓨터공학과 정 성 원

개발자: 4조 박상욱(팀장, 20141522)

이예진(20141564)

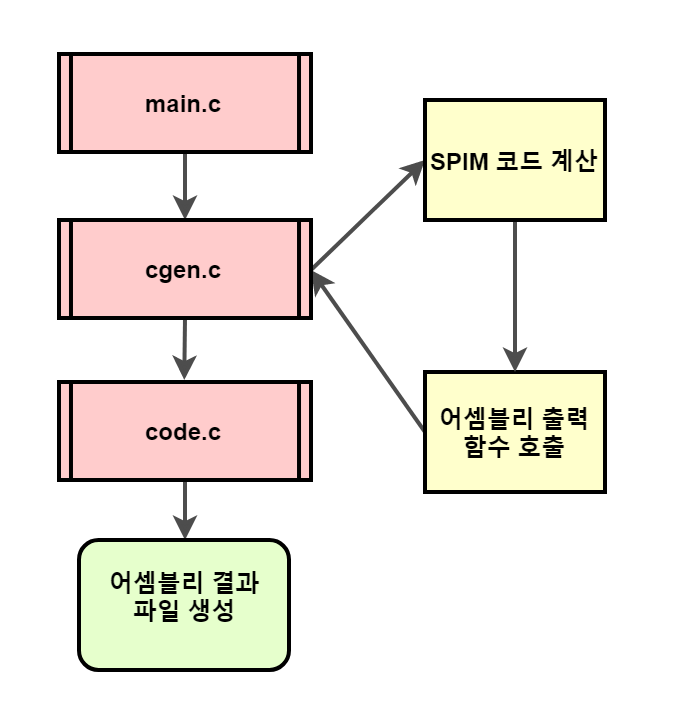
개발기간: 2017. 06. 07 - 2017. 06. 28

1. **개발 목표**

본 프로젝트의 목표는 Semantic Analyzer까지 구현된 프로젝트 3을 기반으로 정상으로 판단되는 코드를 어셈블리 언어로 생성하는 것이다. 이 때 SPIM이라는 툴을 사용한다는 전제하에 개발을 진행하였다. C- 문법을 확인하고 runtime environment을 생성, 유지하면서 code를 생성하여야 하며 이를 위해서는 SPIM reference manual에 정의된 레지스터들과 명령어를 숙지해야 한다. 또한 주어진 레지스터 및 명령어를 적절히 활용하기 위해서 코드가 어떻게 메모리에 적재되고 각 포인터들이 무엇을 의미하는지에 대한 기본적인 흐름에 대해서도 확실한 개념 정립이 필요하다. 최종적으로는 생성된 어셈블리 코드를 QTSPIM을 통해 실행해 봄으로써 프로그램이 의도된 방향으로 진행되는지 확인한다.

1. **개발 범위 및 내용**
   1. **개발범위**

기존에 작성된 코드를 활용하여 생성된 abstact syntax tree와 scope별 hash table을 이용하여 SPIM 어셈블리 코드를 생성한다. 이 때 수정되어야 하는 코드는 크게 code.c와 cgen.c이다. cgen.c는 abstact syntax tree를 순회하면서 scope별 hash table을 이용하여 각 노드의 타입별 알맞은 코드를 생성하도록 가이드 역할을 해주며 code.c에서는 cgen.c에서 지시된 코드 생성을 실질적으로 어셈블리 파일에 출력하는 역할을 한다.



* 1. **개발내용**
     1. 교재에 TM머신에 대한 기본 code가 존재하므로 이를 보면서 어떠한 프로그램을 작성해야 하는지, SPIM을 사용했을 때 어떤 식으로 접근해야 하는지 전체적인 계획을 세운다.
     2. 우선 SPIM머신의 출력형식에 맞게 프로그램을 출력하기 위해 code.c와 code.h를 작성한다. 어셈블리 파일의 출력 양식에 따라서 함수를 여러 개 생성해주는데, 레지스터의 경우 R, immediate 상수일 경우 M으로 구분하여 함수 이름을 설정해주었다. 예를 들어 addi t1,t2,t3일 경우 operand 다음에 레지스터의 이름이 세개 연속 등장하기 때문에 emitRRR와 같이 함수명을 지어주었다. 또한 코드에 대한 주석을 추가하기 위해 주석을 넘기고 주석을 #을 통해 출력할 수 있도록 한다.
     3. SPIM머신에서 작동할 수 있는code를 생성하기 위해 SPIM에 돌아가기 위한 전처리, 전역변수 선언, input, output함수 선언 등에 해당하는 코드를 생성하는 것을 구현하고 앞선 프로젝트에서 생성한 Abstract Syntax tree를 순회하는 함수를 제작한다. 그리고 각 node의 종류에 따라(StmtK, DeclK, ExpK) code를 생성하는 함수를 제작한다.
     4. 지금까지 작성한 파일들과 cgen.c, cgen.h, code.c, code.h의 종속관계를 명시하여 간편하게 실행할 수 있게 Makefile을 제작한다.

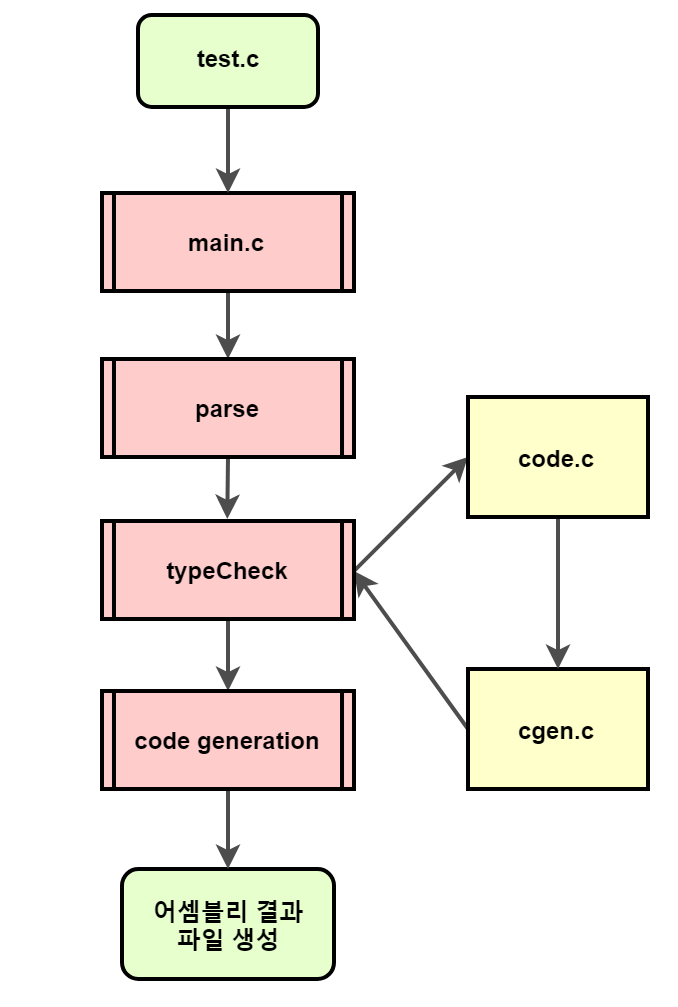
1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 날짜 | 일정 |
| 6/07 ~ 6/10 | 프로젝트 이해 |
| 6/11 ~ 6/12 | 프로젝트 분석 |
| 6/13 ~ 6/17 | 기말고사 기간 |
| 6/18 ~ 6/19 | TM 머신에 해당하는 code 제작해보며 SPIM과의 차이 알아보기 |
| 6/20 ~ 6/20 | SPIM 머신 코드 살펴보기, SPIM명령어 이해 |
| 6/21 | code.c, code.h 제작 |
| 6/22 ~ 6/24 | cgen.c, cgen.h와 3차 프로젝트 수정 |
| 6/25 ~ 6/27 | 보고서 작성 |
| 6/28 | 최종 점검 및 제출 |

* 1. **개발방법**

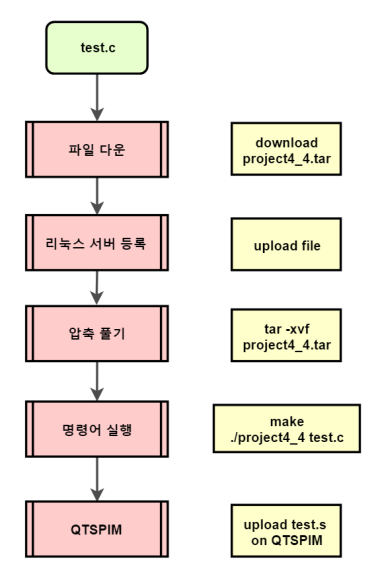
FLEX, YACC 툴이 설치되어 있는 Linux 환경(cspro9.sogang.ac.kr)에서 교재를 참고하여 만든 지난 프로젝트의 Abstract Syntax Tree와 scope별 hash table을 기반으로 개발한다. 정확한 구현을 할 수 있도록 SPIM툴의 매뉴얼을 통해 사용법을 숙지하여 SPIM 언어로 된 .s 파일을 제작할 수 있도록 한다. 또한 메모리에 변수, parameter 및 코드의 각 요소들이 어떻게 쌓이는지에 대한 개념을 정확히 확립한다. (아래의 그림3. 은 메모리에 어떻게 코드가 쌓이는지 명확하게 나타낸다.) 그리고 최종적으로 C- language로 작성된 test 파일에 대해서 생성된 어셈블리 결과 코드(.s)를 QTSPIM이라는 툴에 실행하여 정상적으로 실행되는지 확인한다.

1. **연구 결과**
   1. **합성 내용:**



본 프로젝트에서 주로 수정한 코드는 code.c, cgen.c, code.h, cgen.h이다. cgen.c는 register의 번호, 상수, 명령어 등을 인자로 code.c에 있는 출력함수를 호출한다. code.c의 함수들은 인자들을 가지고 SPIM 어셈블리 코드를 생성해준다.

* 1. **실행 방법:**

본 프로젝트의 실행 방법은 아래의 순서와 같으며 QTSPIM의 경우 9.1.17버전 사용을 권장한다.

* 1. **분석 내용**
     1. **code.c/code.h**

code.c는 cgen.c에서 프로그램 수행 중에 각 노드의 타입에 맞는 어셈블리 코드 생성에 호출되며 cgen.c에서 인자로 명령어(op)의 문자열과 register의 번호와 필요 시 immediate값을 받고, 해당 명령어에 대한 주석을 받는다. 여기서 작성한 함수는 모두 Void 형이며, 주어진 어셈블리 코드를 출력하는 역할만을 수행한다.

1. emitLabel

어셈블리 코드에서 Label을 출력하기 위해 사용하는 함수이다. emitLabel(int label\_num)으로 사용하며, 주어진 label\_num에 대한 label 어셈블리 코드를 만들어 준다.

1. emitComment

어셈블리 코드에서 디버깅을 위한 함수이다. emitComment(char \*c)로 사용하며, #을 이용해 SPIM에서 어셈블리 코드로 인식하지 않고, 주석으로 인식하는 코드를 작성한다.

1. emitRR

두 개의 Register를 사용하는 어셈블리 코드를 생성할 때 사용한다. 함수는 emitRR(char\* op, char\* r, char\* s, char \*c)로 사용하며, r은 첫 번째 Register의 이름을, s는 두 번째 Register의 이름을 의미한다. Op는 어셈블리 Instruction이 된다.

1. emitRM

하나의 Register, 하나의 Immediate를 사용하는 어셈블리 코드를 생성할 때 사용한다. 함수는 emitRM(char\* op, char\* r, int s, char \*c)로 사용하며, r은 Register의 이름을, s는 Immediate에 사용되는 정수 값을 의미한다. Op는 어셈블리 Instruction이 된다.

1. emitRRR

세 개의 Register를 사용하는 어셈블리 코드를 생성할 때 사용한다. 함수는 emitRRR(char\* op, char\* r, char\* s, char\* t, char \*c)로 사용하며, r은 첫 번째 Register의 이름을, S는 두 번째 register의 이름, t는 세 번째 Register이름을 의미한다. Op는 어셈블리 Instruction이 된다.

1. emitRRM

두 개의 Register, 하나의 Immediate를 사용하는 어셈블리 코드를 생성할 때 사용한다. 함수는 emitRRM(char\* op, char\* r, char\* s, int t, char \*c)로 사용하며, r과 s는 Register의 이름을, t는 Immediate에 사용되는 정수 값을 의미한다. Op는 어셈블리 Instruction이 된다.

1. emitRMR

한 개의 Register, 하나의 Immediate, 그리고 하나의 Register를 사용하는 어셈블리 코드를 생성할 때 사용한다. 함수는 emitRMR(char\* op, char\* r, int s, char\* t, char \*c)로 사용하며, r과 t는 Register의 이름을, s는 Immediate에 사용되는 정수 값을 의미한다. Op는 어셈블리 Instruction이 된다.

1. emitBranch  
   branch instruction에 해당하는 어셈블리 코드를 출력하는 함수이다. 예를 들어 j label1 또는 bnez label3 등의 어셈블리 코드를 출력한다. emitBranch(char\* op, char\* r, char\* label, int a, int\*c)로 사용하며 op는 j, bnez와 같은 명령어에 해당하며 label에는 “label”이라는 string에 해당하고, a는 몇 번째 label인지 저장되어 있다.
   * 1. **cgen.c/cgen.h**

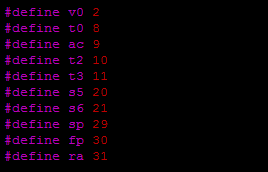
cgen은 AST tree를 순회하면서, 특정 Symbol에 대한 bucketlist을 사용하여 입력으로 받은 code를 SPIM 어셈블리 코드로 생성해주는 프로그램이다.

1. static void cGen(TreeNode \* tree)

앞서 생성된 Syntax Tree를 순회하며, 어셈블리 코드를 Generate하는 함수이다.  
codeGen함수에서 처음 호출하며, TreeNode의 kind에 따라 getStmt, getExp, getDecl 함수를 호출한다.

1. static char\* getRgisterName(int num)

주어진 Register의 Number에 따른 이름을 반환한다. Register Number와 이름에 대한 관계는 아래와 같다.



1. static void genStmt

현재 TreeNode가 StmtK일 때 어셈블리 코드를 생성한다. 앞서 정의한 StmtK의 종류 IfelseK, IfK, WhileK, ReturnK, CompK에 대해서 어셈블리 코드를 생성해준다.

1. static void genExp

현재 TreeNode가 ExpK일 때 어셈블리 코드를 생성한다. 마찬가지로 앞서 정의한 ExpK에 해당하는 ConstK, IdK, OpK, ArrK, CallK에 대해서 어셈블리 코드를 생성해준다.

1. static void genDecl

현재 TreeNode가 DclrK일 때 어셈블리 코드를 생성한다. 마찬가지로 앞서 정의한 DclrK에 해당하는 VarK, VarArrK, FuncK에 대해 어셈블리 코드를 생성한다.

1. void codeGen

SPIM 어셈블리 코드를 생성하기 위해서 가장 먼저 호출되는 함수이다. 가장 먼저 함수가 호출되면, Global Variable에 대해서 .data 부분에 추가한다. C-에서 선언하고 사용 가능한 모든 변수의 Type은 Integer로 제한되므로, SPIM의 32bit 데이터 형인 word를 이용해 Global Variable을 선언한다. 다음으로 입력과 출력에 사용되는 Input 함수와 Output 함수를 정의한다. 각각 ‘input’과 ‘output’이라는 label을 사용하며, 프로그램 실행 시 사용자로부터 입력을 받고, 프로그램의 결과를 출력해줄 수 있도록 한다. 이후 입력받은 C- 프로그램에 대한 어셈블리 코드를 생성하기 위해, fp와 ra를 초기화 하고 cGen함수를 호출한다.

1. void find

C- 프로그램의 어떤 함수가 몇 개의 Local Variable을 가지고 있는지 구하는 함수이다. 이 함수를 이용해서 function이 선언될 때, Local Variable을 위한 Stack Size를 결정할 수 있다.

1. void push

SPIM 코드에서 Stack을 이용해 Register의 내용을 Push하기 위한 어셈블리 코드를 생성한다.

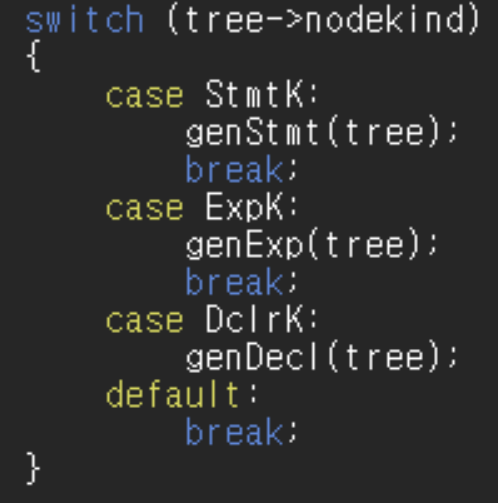
1. void pop

SPIM 코드에서 Stack의 Top을 Register로 Pop하기 위한 어셈블리 코드를 생성한다.

* 1. **제작 내용**cgen은 AST tree를 순회하면서, 특정 Symbol에 대한 bucketlist을 사용하여 입력으로 받은 code를 SPIM 어셈블리 코드로 생성해주는 프로그램이다.

1. static void cGen(TreeNode \* tree)

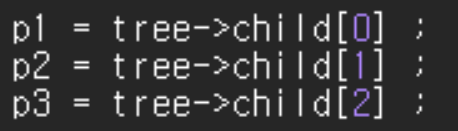
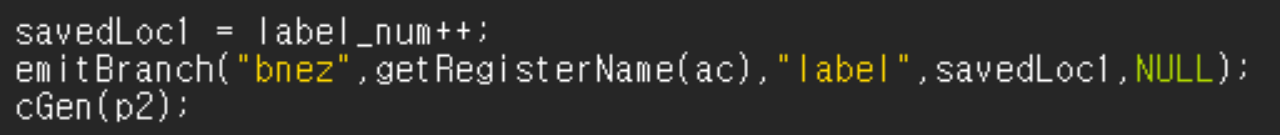
앞서 생성된 Syntax Tree를 순회하며, 어셈블리 코드를 Generate하는 함수이다.  
codeGen함수에서 처음 호출하며, TreeNode의 kind에 따라 다음과 같이 switch문을 사용하여 getStmt, getExp, getDecl 함수를 호출한다.

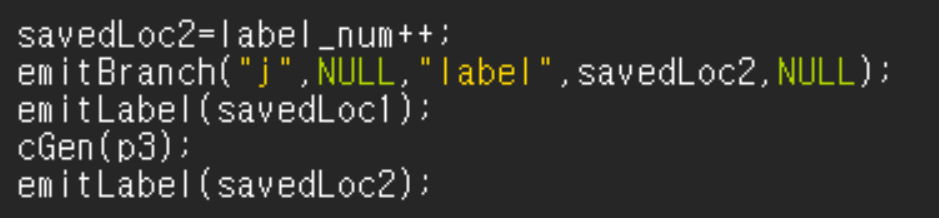


1. static void genStmt

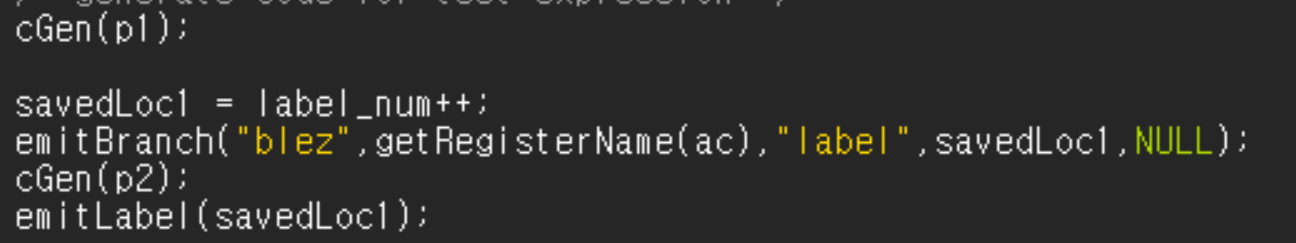
현재 TreeNode가 StmtK일 때 어셈블리 코드를 생성한다. 앞서 정의한 StmtK의 종류 IfelseK, IfK, WhileK, ReturnK, CompK에 대해서 어셈블리 코드를 생성해준다.

* 1. IfelseK

P1는 if문의 조건, p2는 compound statement, p3는 else문의 compound statement를 가리키도록 설정한다. 그리고 if문에 조건에 대한 코드를 생성하기 위해 cGen(p1)을 재귀적으로 호출해준다.   
  
그리고 난 뒤 if문의 조건에 맞지 않을 경우 else문의 body로 jump해야 하므로 이를 수행하기 위해 라벨을 설정해주어 이 역할을 하는 어셈블리 코드를 설정해준다.  


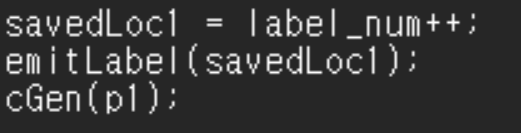
하지만 if문의 조건에 맞을 경우 else문의 body를 수행하면 안되기 때문에 else문의 compound statement다음에 또 다른 라벨을 생성하여 이를 구현한다.  


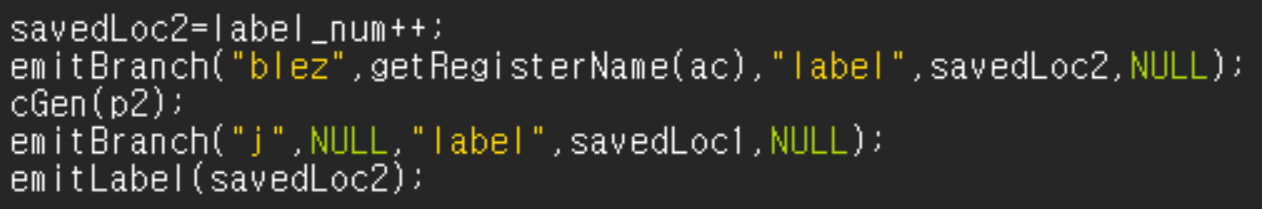
* 1. IfK

P1는 if문의 조건, p2는 if문의 compound statement을 가리키도록 한다. 조건에 해당하는 코드를 생성해주고, if문의 조건에 맞지 않다면 if문 compound statement의 다음으로 jump해주는 코드를 다음과 같이 라벨을 생성하여 구현해주었다.  


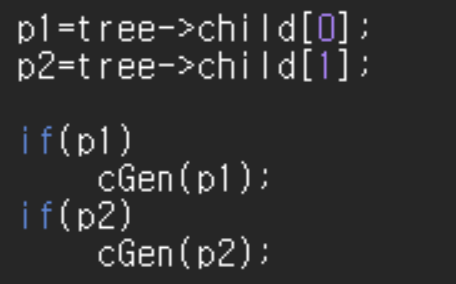
만약 if문의 조건을 만족한다면 if문의compound statement가 실행되도록 하였다.

* 1. WhileK

P1는 while문의 조건, p2는 while문의 compound statement를 가리키도록 하였다. while문의 compound statement가 끝난 후 다시 조건을 체크하러 돌아와야 할 수도 있기 때문에 이 부분에 먼저 라벨을 출력해주고, 조건에 대한 코드를 생성하였다.  


while문의 조건을 만족하지 않는다면, while문의 compound statement의 다음으로 jump 하는 코드를 다음과 같이 라벨을 생성하여 구현해주었다. 이와 반대로 while문의 조건을 만족한다면 while문의 compound statement가 실행되도록 해주었다.  


* 1. ReturnK  
     return문에는 단일 상수나 변수가 올 수도 있지만 expression도 올 수 있다. 만약 expression이 return문에 등장한다면 이 expression으로 cGen함수를 재귀적으로 호출하여 코드를 생성하였다.   
     return하게 될 주소를 sp에 저장해 준 후, stack에서 return address와 fp를 꺼내준다. 그리고 마지막으로 return address로 jump해준다.
  2. CompK

Compound statement가 다중으로 존재할 수 있기 때문에 다음과 같이 종속된 compound statement가 존재하는지 체크하여 재귀적으로 cGen을 호출해주었다.  


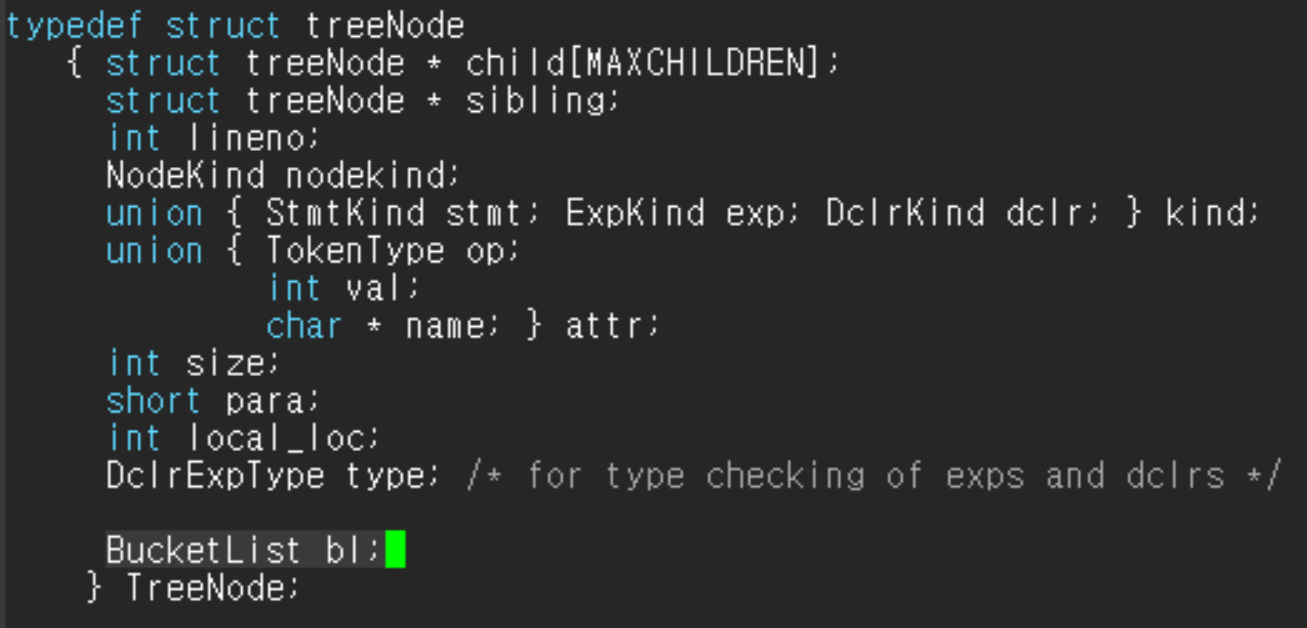
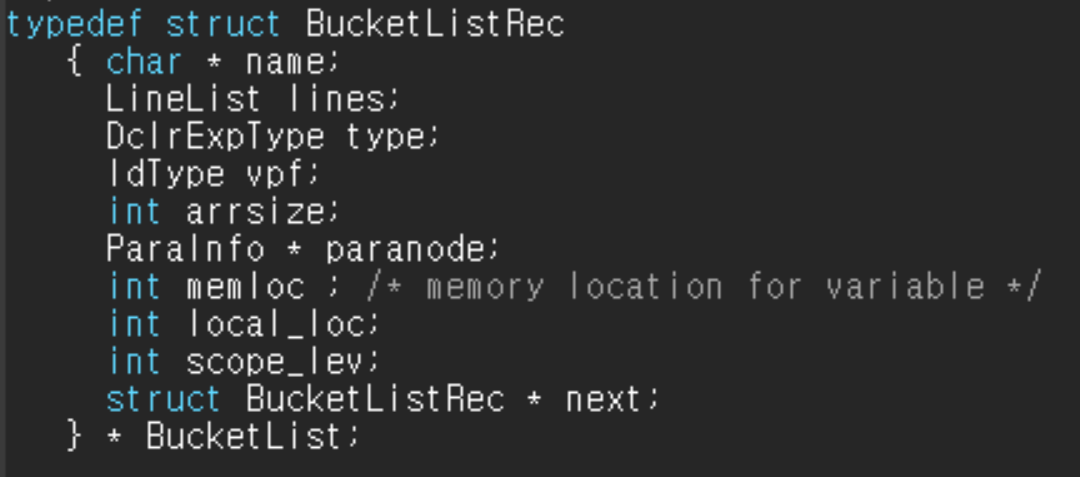
1. static void genExp

현재 TreeNode가 ExpK일 때 어셈블리 코드를 생성한다. 마찬가지로 앞서 정의한 ExpK에 해당하는 ConstK, IdK, OpK, ArrK, CallK에 대해서 어셈블리 코드를 생성해준다.

* 1. ConstK

상수에 대한 참조가 있을 경우, li 연산자를 사용해 tree->attr.val에 저장되어있는 실제 상수 값을 t1 register에 넣어준다.

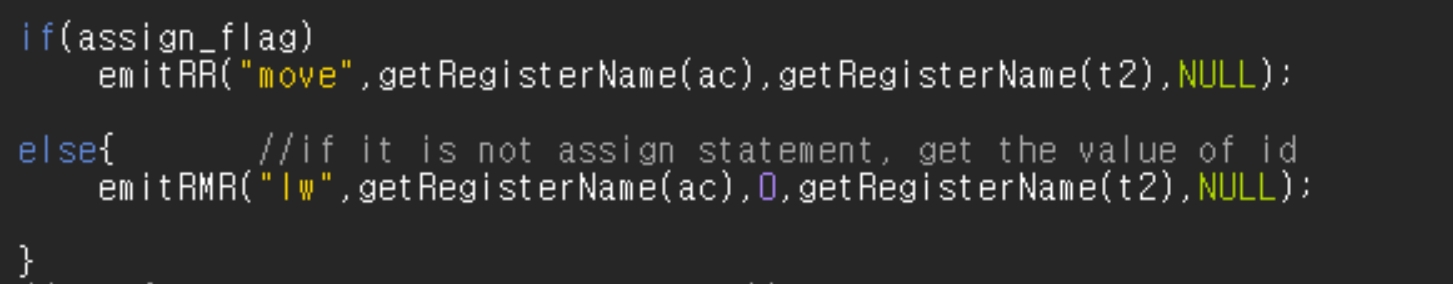
* 1. IdK

Compound statement안에서 local변수나 global변수에 대한 참조가 일어났을 때, 이 변수들이 선언됐을 때 저장해놨던 정보들이 필요하다. 따라서 이러한 부분을 구현하기 위해 다음과 같이 기존의 TreeNode에 선언 시 정보들이 저장돼있는 BucketList 구조체를 추가해주었다.  
따라서 tree->bl에 들어가보면 다음과 같은 정보들에 접근할 수 있게 된다.  
.

따라서 이 BucketList에서 정보를 받아와서 우선 arrsize로 배열인지 아닌지 판단한다. 왜냐하면 배열의 주소가 compound statement에서 접근될 경우 ArrayK가 아닌 IdK노드로 인식되기 때문에 이에 대한 적절한 처리를 해주기 위함이다.   
  
따라서 실제로 배열타입일 경우, 먼저 global변수인지 local변수인지 구분해준다. global변수라면 offset을 기준으로 접근하는 것이 아니라, 직접 배열 변수의 이름을 가져와서 static하게 접근할 수 있도록 한다.  
local변수라면, 먼저 이 배열이 parameter에 해당하는지 실제 compound statement내에서 선언된 지역변수인지 확인하여 각각의 경우에 맞는 처리를 해준다.

그 다음으로 배열이 아닌 일반 변수가 들어왔을 때에도 마찬가지로 먼저 global변수인지 local변수인지 구분해준다. Global일 경우 위에서 설명한 바와 같이 변수의 이름으로 static하게 접근이 가능하도록 처리해주고, local변수일 경우 parameter로 넘어온 변수인지 compound statement 내에서 지역적으로 선언된 변수인지 구분하여 각각의 상황에 맞는 처리를 해준다.

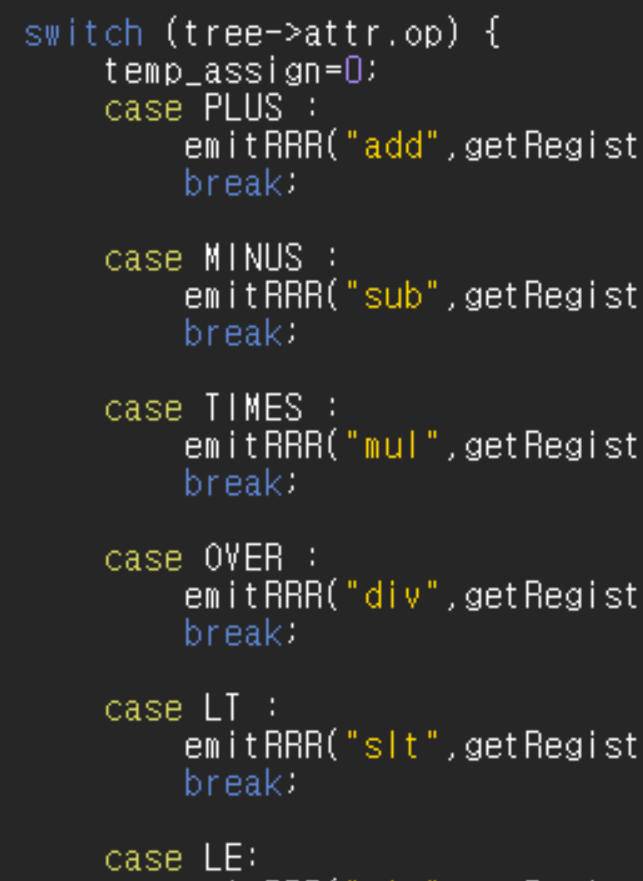
마지막으로 중요한 처리는, assignment statement의 왼쪽 항에 등장한 변수일 경우이다. 이 경우 변수의 값을 가져오는 것이 아니라, 변수의 주소를 가져와서 값을 저장해주어야 하기 때문에, 다음과 같이 assignment의 왼쪽 항인 경우와 아닌 경우를 구분해준다. (assignment\_flag가 1이라면 이에 해당한다. Assignment\_flag에 대해서는 뒤에서 자세히 설명한다)



* 1. OpK

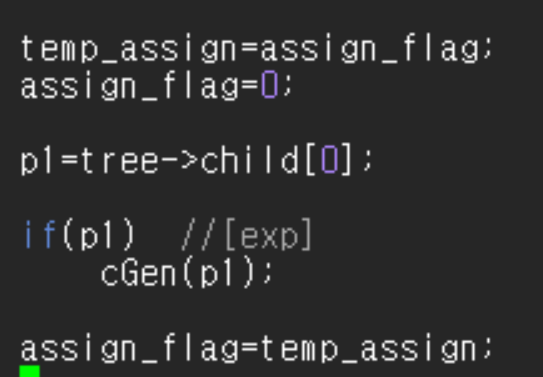
논리 연산, 사칙 연산을 위한 어셈블리 코드를 생성하도록 한다. Operator에 따라 모든 경우를 나누어, 각 Operator에 해당하는 Instructiuon을 수행하도록 하였다.

Operator가 Assign이 아닌 경우, 먼저 Left Child에 대한 어셈블리 코드를 생성 후 Right Child에 대한 exp의 어셈블리 코드를 우선 생성한다. Operator가 Assign인 경우에는 Right Child의 exp를 먼저 코드를 생성한 뒤, Left Child의 Id에 대한 메모리 주소에 저장할 수 있도록 어셈블리 코드를 생성한다.

다음과 같이 switch문으로 C- language에서 가능한 모든 operand들에 대해 저적한 어셈블리 코드를 생성해준다.  


이 operand들 중 Assign의 경우 따로 처리를 해주어야 한다. Assign의 경우 위의 다른 operand들과 다르게 오른쪽 항에 대한 코드를 재귀적으로 cGen을 호출하여 생성해준다. 생성된 오른쪽 항을 stack에 임시로 push해주어 추후에 왼쪽 항으로의 대입이 성공적으로 일어날 수 있도록 한다. 그리고 난 뒤, assign\_flag를 1로 setting해주어 이후에 생성되게 될 왼쪽 항에 해당하는 주소값(값이 아닌)을 받아올 수 있도록 명시해준다. 왼쪽 항에 대한 코드 생성이 끝나면stack에서 오른쪽 항을 pop하여 대입을 진행한다.  
이 과정이 끝나고 나면 assignment 연산이 끝났다는 의미이므로 assignment\_flag를 다시 0으로 초기화해준다.

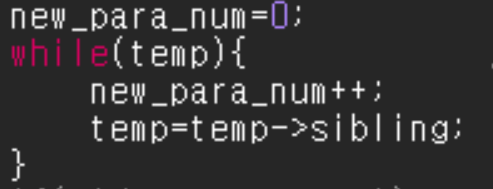
* 1. ArrK

Exp가 Array인 경우에 대해 어셈블리 코드를 생성한다. 하지만 이를 수행하기 전에 먼저 처리해야 할 사항이 있다. 앞서 assignment statement의 왼쪽 항에 arr[i]와 같은 배열이 등장했을 경우를 생각해보자. 왼쪽 항의 코드를 생성하는 중에는 assignment\_flag가 1이기 때문에 i에 대한 코드를 생성할 때도 i의 값이 아닌 주소를 가져오는 어셈블리 코드가 생성될 것이다. 이를 방지하기 위해 만약 array의 index에 expression이 존재한다면 다음과 같이 잠시 assign\_flag를 꺼두고 expression에 대한 코드 생성을 마친 후, 다시 assign\_flag를 켜주는 방식을 적용하였다.  


Array를 호출할 때, [] 내부에 또다른 Exp가 올 수 있으므로, 내부의 exp에 대한 어셈블리 코드를 먼저 수행할 수 있도록 한다. Array에 대한 코드를 생성할 때, 호출한 Array가 Global Variable인지, Local Variable인지 판단하여, 전자인 경우 미리 선언한 변수를 호출할 수 있도록 하고, 후자인 경우 parameter에 해당하는 배열인 경우와 아닌 경우를 나누어 각각 Frame Pointer를 이용해 적절하게 Stack의 Array를 호출하도록 한다.

마지막으로, assignment\_flag가 켜져 있지 않는 경우 배열의 값을 가져와야 하므로 이에 필요한 추가적인 어셈블리 코드를 출력해주었다.

* 1. CallK

함수를 호출하는 어셈블리 코드를 생성한다. 우선, 함수 호출 시 parameter의 개수를 알아야 하기 때문에 다음과 같이 해당 함수에 몇 개의 parameter가 존재하는지 세주었다.  


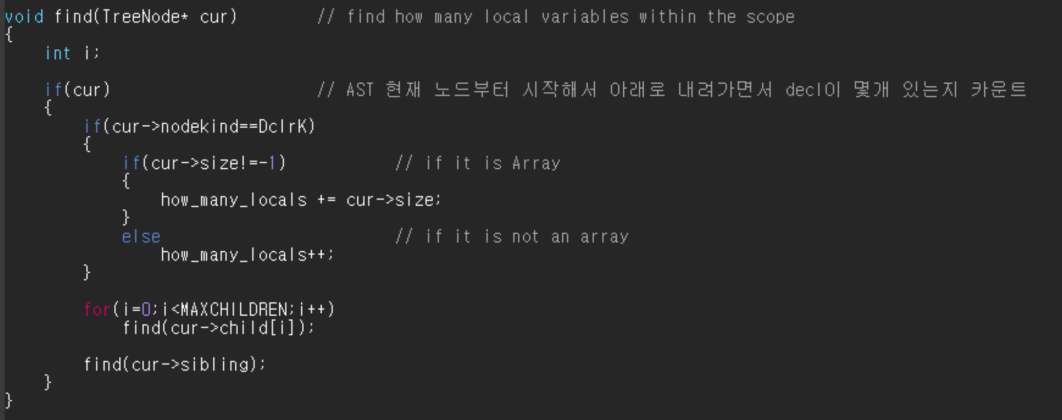
그 후, 함수를 호출하는 데에 사용하는 Parameter의 코드를 생성해준다. 각각의 Parameter에 대해서 genExp함수를 통해서 코드를 생성한다. Parameter에 대한 코드가 모두 생성되었다면, call\_flag를 켜주고 함수를 호출에 필요한 부가적인 코드를 출력한다. 이 call\_flag는 함수 호출 시, 배열의 주소를 넘기게 되는 경우에 알맞은 어셈블리 코드 출력을 위한 flag로 사용된다.

1. static void genDecl

현재 TreeNode가 DclrK일 때 어셈블리 코드를 생성한다. 마찬가지로 앞서 정의한 DclrK에 해당하는 VarK, VarArrK, FuncK에 대해 어셈블리 코드를 생성한다.

* 1. FuncK

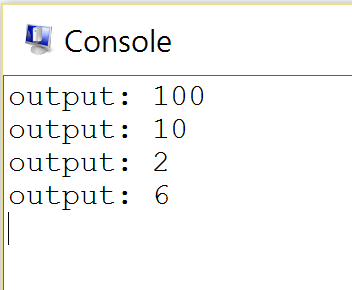
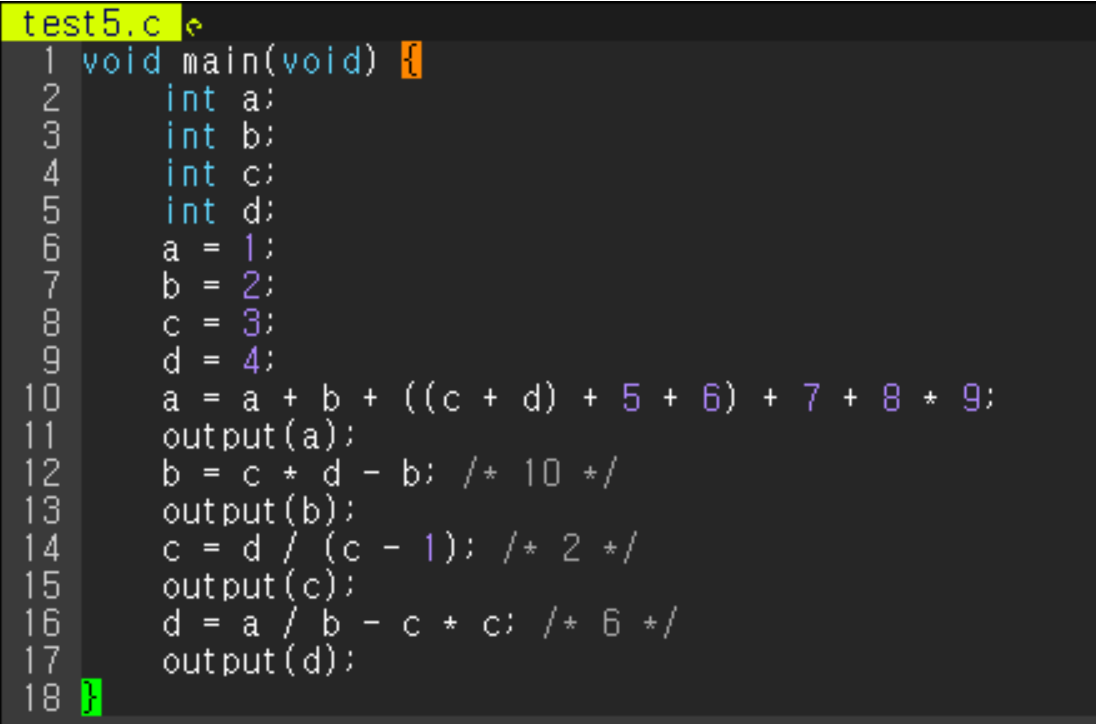
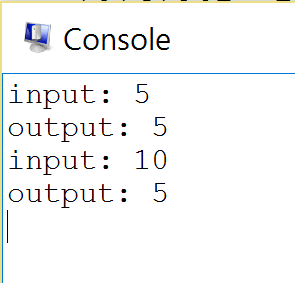
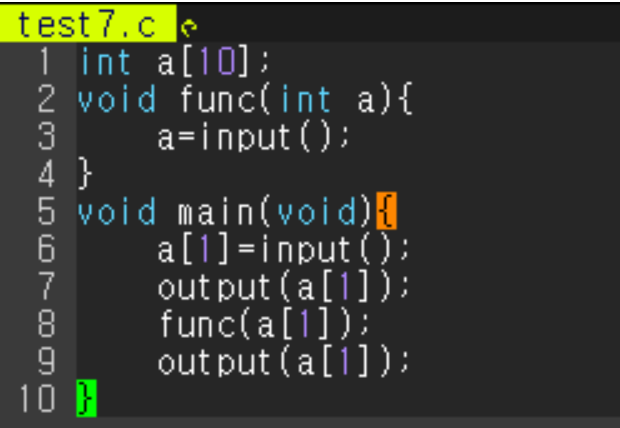
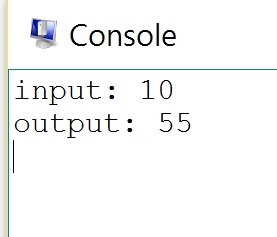
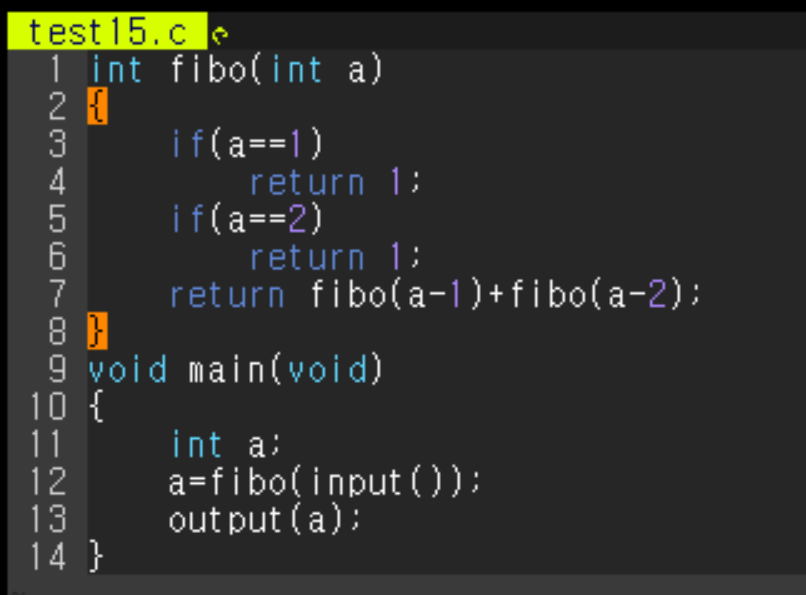
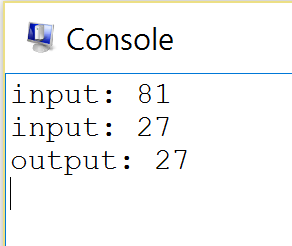
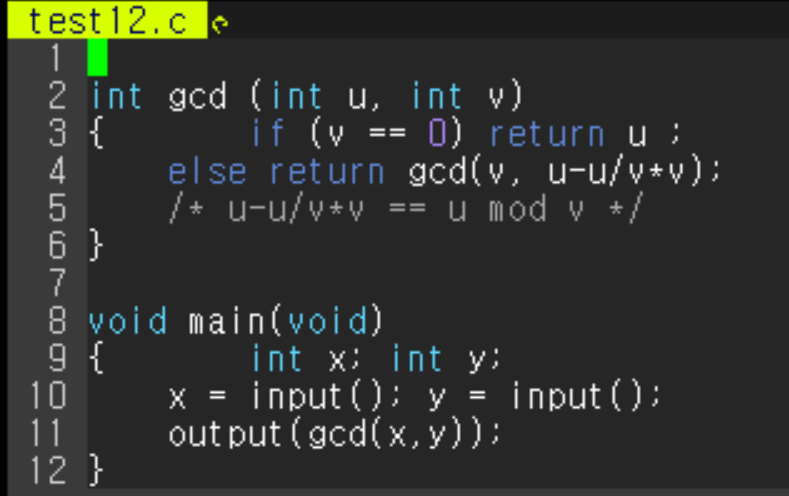
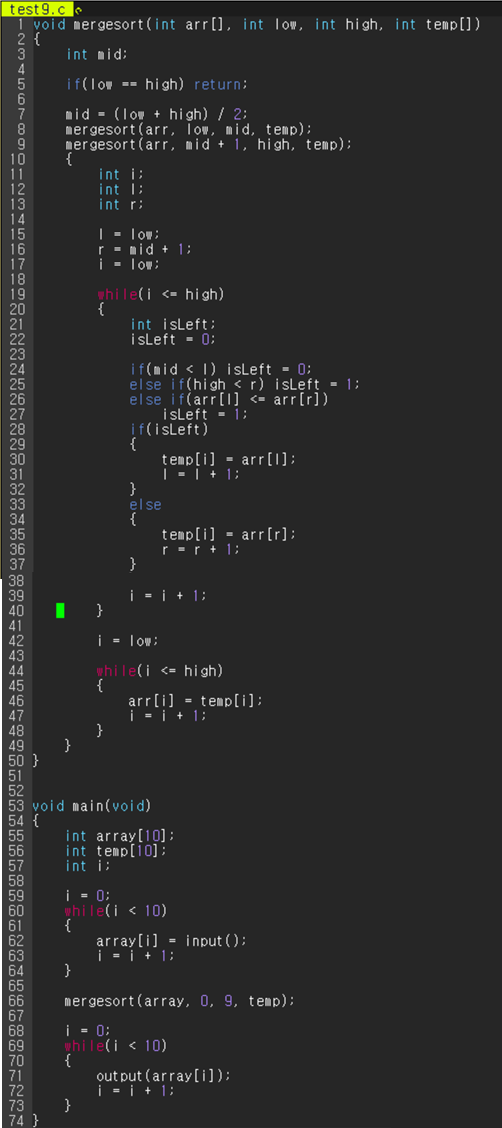
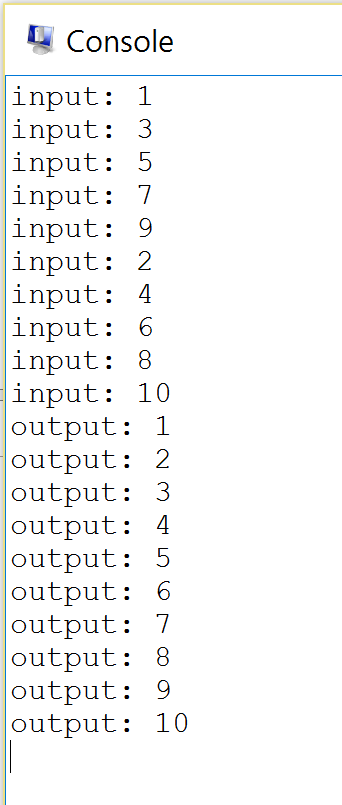
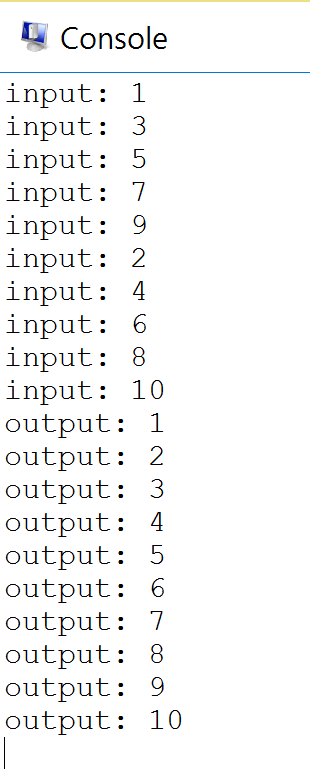
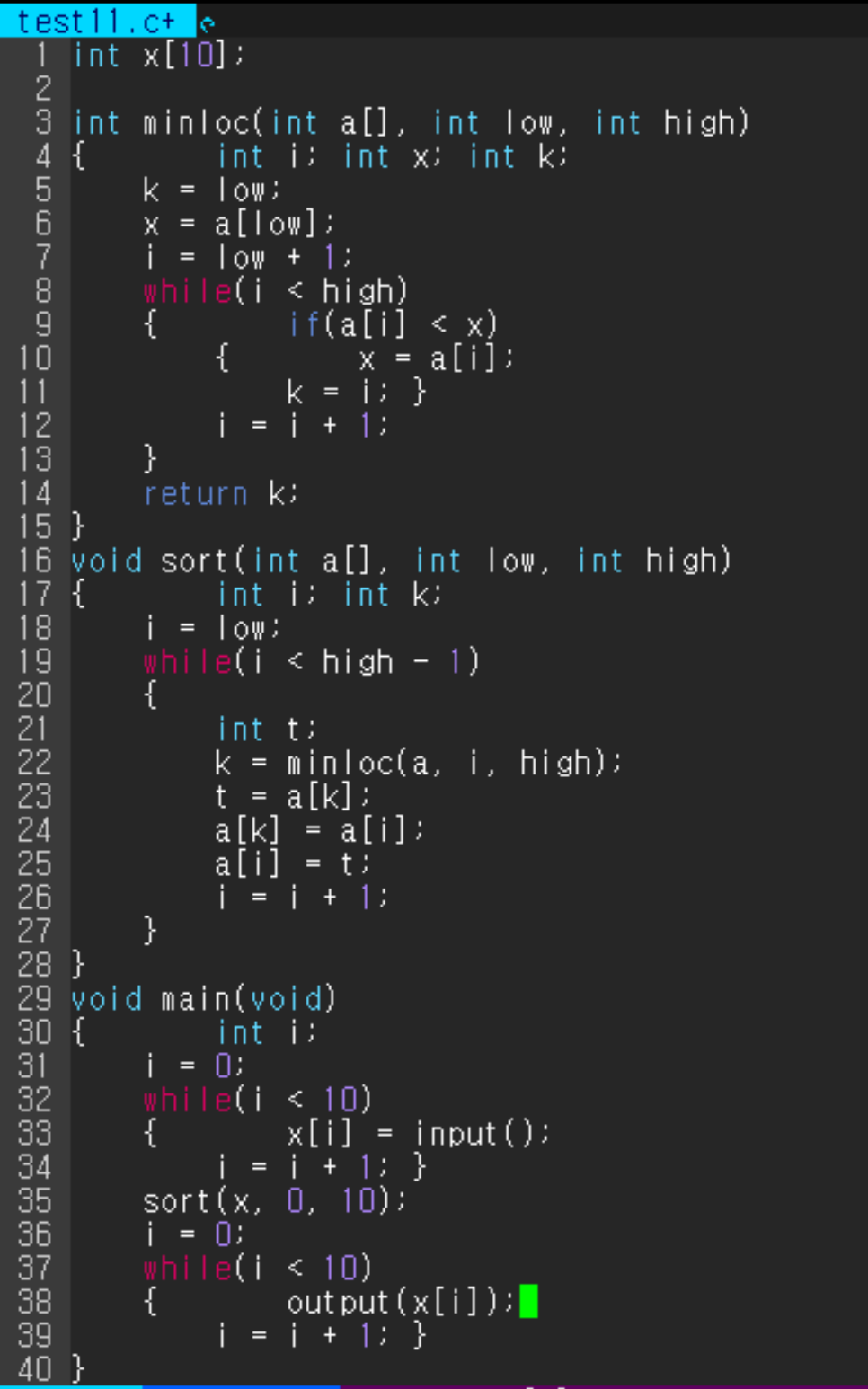
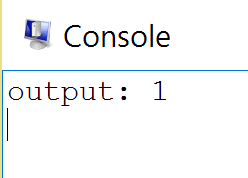
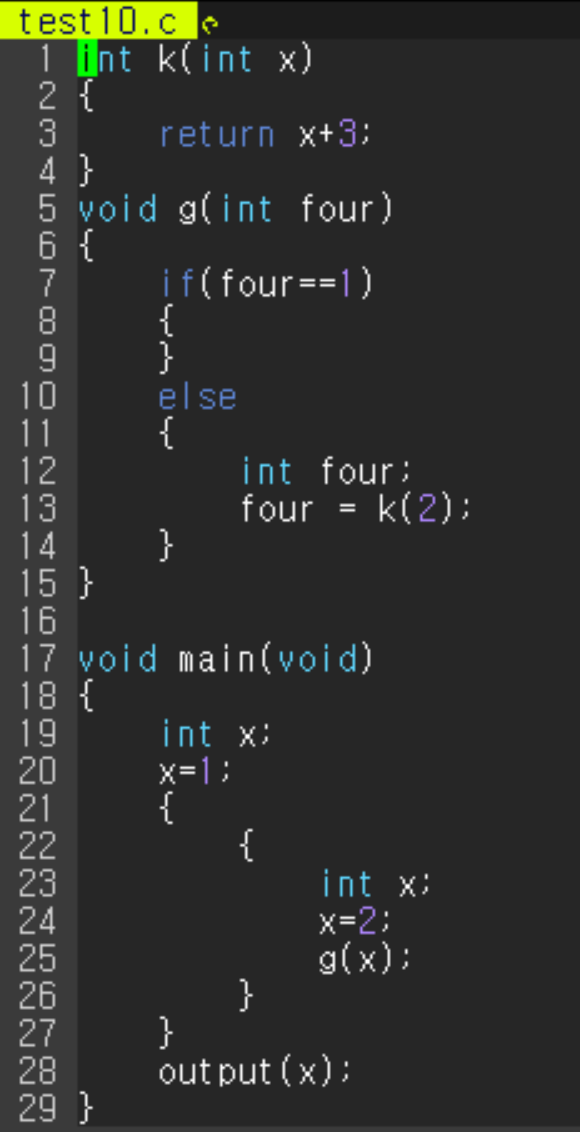
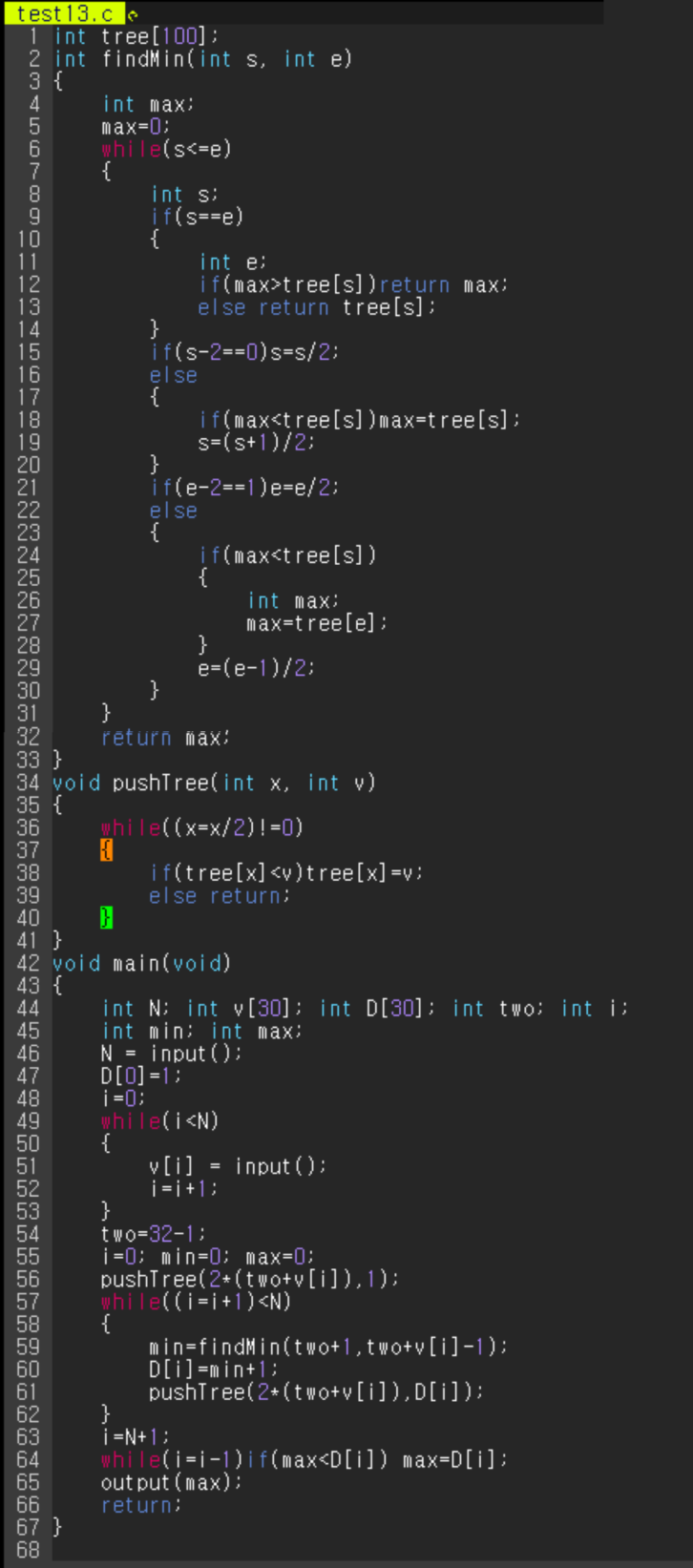
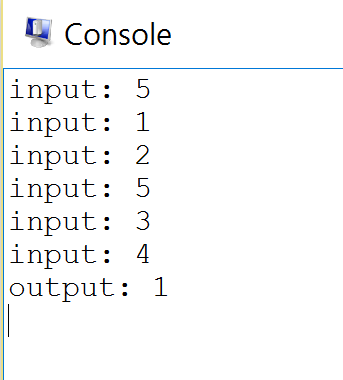
VarK와 VarArrK일 때는 별다른 처리를 해주지 않으며, FuncK일 때, 함수가 선언될 때 코드를 생성해준다. 특히 함수가 main인 경우에 대해 부가적으로 필요한 어셈블리 코드를 생성해주며, 다른 함수인 경우, label 코드를 출력하고, 함수 수행에 필요한 변수의 개수에 따른 Stack Size 설정과 같은 코드를 생성한다.

이때, 이 함수에 선언된 지역변수가 몇 개 있는지 알아야 하기 때문에 find()라는 함수를 다음과 같이 따로 만들어서 이를 구현하였다.

이 함수는 현재 tree node부터 시작해서 abstract syntax tree를 아래로 내려가면서 선언에 해당하는 노드가 몇 개이었는지 세어주는 역할을 하였다. 이때 함수 안의 함수의 선언은 불가하기 때문에 이에 대한 예외처리를 따로 하지 않았다.

1. void codeGen

가장 먼저 호출되어, 아래와 같이 Global Variable에 대한 코드를 생성해준다. Global Variable은 Local Variable과 달리 어셈블리 코드에서의 변수로 선언한다. 그리고 프로그램 수행을 위해서 Input과 Output 함수에 대한 어셈블리 코드를 출력한다.

* 1. **시험 내용**
     1. **기본 expression**
     2. **전역, 지역 변수 중복 사용**
     3. **피보나치**
     4. **GCD**
     5. **Merge sort  
        **
     6. **Bubble sort**
     7. **복잡한 Compound**
     8. **Dynamic Programming  
        **
  2. **평가 내용:** 안정성, 신뢰성, 내구성

본 프로젝트에서 앞서 여러 가지 테스트 케이스를 실험해본 결과 오류없이 올바른 결과를 보여주는 것을 확인할 수 있었다. 마찬가지로 테스트 케이스에 대해 여러 번 반복적으로 수행하였을 때도, SPIM에서 동일한 결과를 보여주었고, 오류는 발생하지 않았다.

1. **기타**
   1. **연구 조원 기여도**

박상욱 (20141522) : 50 %

이예진 (20141564) : 50 %

* 1. **자체 평가**

본 프로젝트를 구현하는 데 있어 독자적인 Symbol Table과 BucketList를 이용하였다. 이를 통해 무사히 올바른 결과를 출력하는 프로젝트를 수행할 수 있었다. 다만 프로젝트를 진행함에 있어 앞선 Proj3에서 Code Generation을 고려하지 않고 수행하였기 때문에, 이번 프로젝트를 진행하며 수정해주어야 하는 부분이 많았다. 팀원 간 분업을 통해서 새로운 개발과 앞선 프로젝트의 보수를 동시에 진행할 수 있었고, 문제없이 두 부분을 합쳐 본 프로젝트를 완성할 수 있었다. 본 프로젝트는 다양한 테스트 케이스에서 문제없이 작동하였기에 우수하다고 할 수 있다.

* 1. **느낀 점**

컴파일러의 마지막 프로젝트를 진행하며, 어셈블리까지 다루게 되었다. 총 4개의 프로젝트를 진행하게 되면서, 컴파일러가 어떻게 구성되고, 실행되는지 학습할 수 있었다. 마지막 프로젝트를 진행하면서 제작한 어셈블리 파일은 어셈블리 수업과 SP 수업과도 연관지어 생각해 볼 수 있는 계기가 되었다. 앞서 작성한 Syntax Tree와 Symbol Table을 이용해 코드를 생성하는 것이 쉽지는 않았지만, 프로젝트를 전부 완성하여, 스스로 작성한 컴파일러가 작동할 때에는 뿌듯함을 느낄 수 있었다.

1. **참고**:

참고 교재 : Compiler Construction: Principles and Practice. Kenneth C. Louden 저.