**Design and Development of Compiler**

**for C-Language**

**Phase 4: Design and Implementation of Code Generator**

**1. Project4 결과 보고서**

**과목명: [CSE4120] 기초 컴파일러 구성**

**담당교수: 서강대학교 컴퓨터공학과 정성원**

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

**개발기간: 2015. 06. 08. - 2015. 06. 29.**

**Project4 결과 보 고 서**

**프로젝트 제목: Design and Development of Compiler for C-language: Phase 4: A Code Generator**

**제출일: 2015. 06. 29.**

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

**I. 개발 목표**

* Implementing a code generator for C- according to the runtime environment. The code generated by my code generator should be able to run on SPIM machine

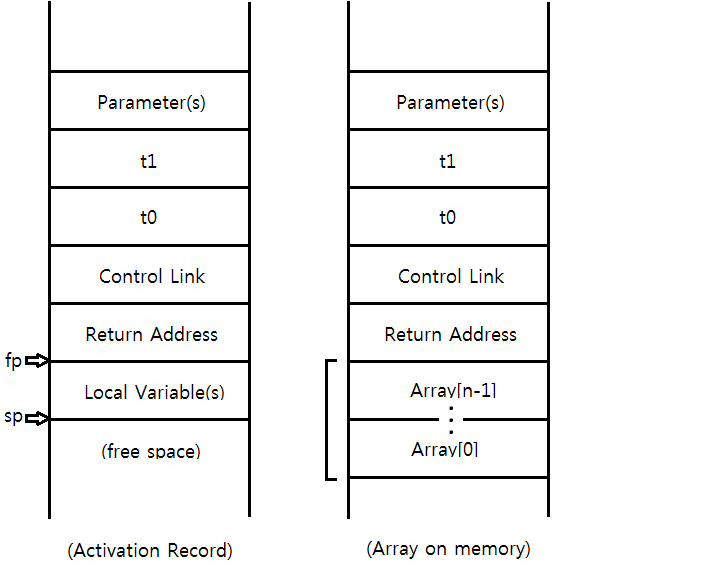
**II. 개발 범위 및 내용**

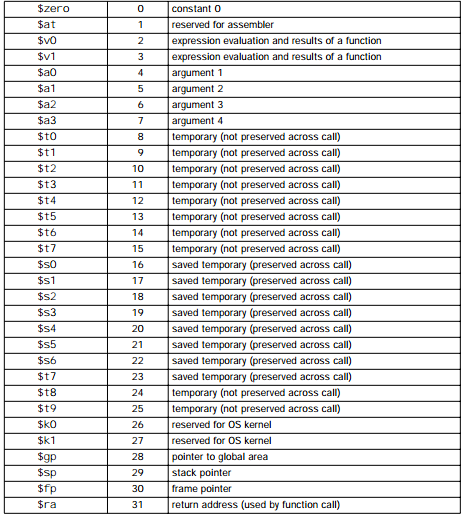
**가. 개발 범위**

* Construct Runtime environment
* Control link
* Return address
* Data area
* Frame pointer
* Stack pointer
* Local variables space
* Saving register value space
* Parameters space
* Free space
* Generate code for C- language

**나. 개발 내용**

* Construct Runtime environment
* Control link: 항상 바로 이전 frame pointer를 가리키게 구현한다.
* Return address: code에서 call이 발생 할 때 callee가 돌아갈 주소를 저장하고 있게 구현한다.
* data area: C- language에서 global영역(scope가 0인 symbol table)에 생성된 변수를 저장하는 공간으로 global variable일 경우 이 area에 저장하게 구현한다.
* Frame pointer: call이 일어날 경우 fp(Frame pointer)는 항상 return address를 가리키게 구현한다.
* Stack pointer: stack area에서 항상 top(activation record에서는 맨 아래)을 가리키게 구현한다.
* Local variables space: return address 아래로 내려가며 offset에 맞춰 push를 이용해 sp(Stack pointer)값을 변경하며 stack에 저장하게 구현한다.
* Saving register value space: t0 register와 t1 register를 저장하는 영역으로 register의 이전 값을 보존하기 위해 존재하는 영역으로 control link 위로 차례로 t0, t1 값을 저장하고 있게 구현한다.
* Parameters space: register값 2개를 저장하고 있는 영역(Saving register value space) 위로 올라가며 parameter 순서(offset)에 맞춰 push를 이용해 stack에 저장하게 구현한다.
* Free space: 자유롭게 사용하는 공간으로 operator 연산이나 assignment등을 처리할 때 push, pop을 이용하여 sp값을 변경하며 stack에서 처리 할 수 있게 구현한다.



* Generate code for C- language
* SPIM machine에서 실행이 가능한 SPIM code(\*.spim)를 생성한다. Abstract Syntax Tree를 순회하면서 각 node에 대한 code를 SPIM 문법에 맞춰 생성한다.
* Registers of SPIM machine  
  

**III. 추진 일정 및 개발 방법**

**가. 추진 일정**

* 6/8 ~ 6/9 Runtime environment에 대한 이해 및 설계
* 6/10 ~ 6/11 SPIM machine에 대한 이해
* 6/12 ~ 6/13 Code generation에 대한 이해
* 6/21 ~ 6/27 Code generator for C- language 구현
* 6/28 ~ 6/29 Test 및 결과 보고서 작성

**나. 개발 방법**

* **cgen.h**
* cgen.c에 필요한 함수 및 변수 선언
* **cgen.c**
* pop\_stack(): stack에서 sp가 가리키고 있는 값을 register에 저장해주는 함수를 구현한다.
* push\_stack(): register에 저장된 값을 stack에서 sp가 가리키고 있는 값에 저장하는 함수를 구현한다.
* genStmt(): C- language에 맞게 수정한다.
* genExp(): C- language에 맞게 수정한다.
* genDecl(): C- language에 만 있는 declaration node에 대한 code를 generation 해주는 함수를 구현한다.
* cGen(): genDecl() 함수를 호출하는 부분이 있게 수정한다.
* codeGen(): Abstract Syntax Tree를 순회하면 실제적으로 각 node마다 code generation 하기 전에 처리 해야 하는 부분이 있게 수정한다.
* calBase(): activation record에서 현재 symbol table의 offset 기준으로 실제 address 값을 구해주는 함수를 구현한다.
* **symtab.h**
* cgen.c에서 사용할 structure들이나 상수를 symtab.c에서 가져오고 symtab.c에 추가로 생성한 함수 및 변수 선언
* **symtab.c**
* getScopeTypeHead(): analyze 할 때 생성한 symbol table의 head를 반환하는 함수 구현한다
* getScopeList(): 현재 변수가 어느 symbol table에 있는지 찾아주는 함수 구현한다
* getBucket(): 현재 변수가 어느 symbol table의 Bucket에 있는지 찾아주는 함수 구현한다
* **main.c**
* Flag들을 code generator에 맞게 setting 한다.

**다. 연구원 역할 분담**

* **이상범:** Code generation 구현
* **모지수:** Runtime environment 설계 및 구현
* **박영훈:** Code generation 구현

**IV. 연구 결과**

1. **합성 내용:**
2. **SPIM Manual**

**1) Arithmetic Operator**

add Rdest, Rsrc1, Src2 Addition (with overflow)

addu Rdest, Rsrc1, Src2 Addition (without overflow)

addiu Rdest, Rsrc1, Imm Addition Immediate (without overflow)sub Rdest, Rsrc1, Src2 Subtract (with overflow)

subu Rdest, Rsrc1, Src2 Subtract (without overflow)

div Rsrc1, Rsrc2 Divide (signed)

divu Rsrc1, Rsrc2 Divide (unsigned)

div Rdest, Rsrc1, Src2 Divide (signed, with overflow)

divu Rdest, Rsrc1, Src2 Divide (unsigned, without overflow)

mul Rdest, Rsrc1, Src2 Multiply (without overflow)

**2) Comparision Operator**

seq Rdest, Rsrc1, Src2 Set Equal

sge Rdest, Rsrc1, Src2 Set Greater Than Equal

sgeu Rdest, Rsrc1, Src2 Set Greater Than Equal Unsigned

sgt Rdest, Rsrc1, Src2 Set Greater Than

sgtu Rdest, Rsrc1, Src2 Set Greater Than Unsigned

sle Rdest, Rsrc1, Src2 Set Less Than Equal

sleu Rdest, Rsrc1, Src2 Set Less Than Equal Unsigned

slt Rdest, Rsrc1, Src2 Set Less Than

slti Rdest, Rsrc1, Imm Set Less Than Immediate

sltu Rdest, Rsrc1, Src2 Set Less Than Unsigned

sltiu Rdest, Rsrc1, Imm Set Less Than Unsigned Immediate

sne Rdest, Rsrc1, Src2 Set Not Equal

**3) Jump/Branch Operator**

b label Branch instruction

beqz Rsrc, label Branch on Equal Zero

jr Rsrc Jump Register

**4) Load/Store Operator**

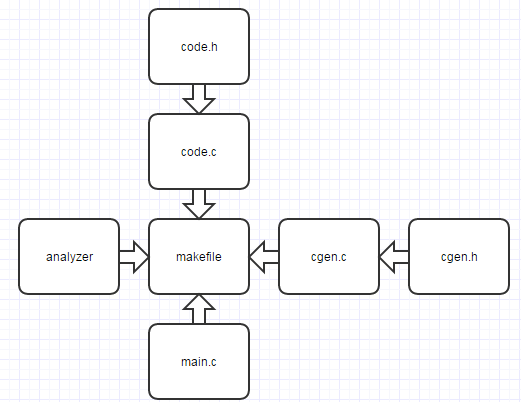
la Rdest, address Load Address

lw Rdest, address Load Word

li Rdest, imm Load Immediate  
sw Rsrc, address Store Word

move Rdest, Rsrc Move content of Rsrc to Rdest

1. **프로그램 구성도**



1. **Files**

* Analyzer: project3의 구현 내용으로 code generate를 위해 필요한 정보(Abstract Syntax Tree, Symbol table)를 저장 및 관리 한다.
* code.h/code.c: SPIM code를 파일에 출력하는데 도움을 주는 유틸리티를 관리한다. 이 file에서는 static scope rule을 기반으로 code를 generation하는 유틸리티들을 가지고 있다. Static scope rule이란 가장 안쪽의 nested된 statement의 변수부터 차례로 확인하며 바깥쪽으로 확장해 가며 변수가 있는지 check한다. 즉 이번 프로젝트에서는 scope값을 한 개씩 줄여가며 현재 nested된 statement와 연결된 symbol table을 차례로 확인한다.
* cgen.h/cgen.c: 실질적으로 코드를 생성하는 파일로 우리가 설계한 Runtime environment를 기반으로 SPIM machine에서 돌아 갈 수 있는 C- language의 code를 생성한다.
* main.c: program의 main으로 여러 옵션들을 선택 가능하다. 이번 프로젝트에서는 TraceCode를 True로 setting하고 code를 생성해 \*.spim file에 저장한다.

1. **분석 내용:**

* **symtab.c:** code generation 할 때 필요한 함수들을 구현해야 한다. 필요한 함수는 다음과 같다.

1. 현재 variable이 있는 symbol table의 BucketList를 반환해주는 함수
2. 현재 variable이 있는 symbol table의 ScopeList를 반환해주는 함수
3. Symbol table의 ScopeTypeHead를 반환해주는 함수

* **cgen.c**

code generation 단계가 되면 main 함수에서 codeGen() 함수를 호출해 C- language를 위한 code generation이 시작된다. 먼저 data area에 저장할 변수들을 찾아 stack에 push 한다. 그 후 기본 내장 함수인 input(), output()에 대한 code를 generation 한다. 그 후 cGen() 함수를 호출해 각 node에 따라 code를 generation 한다.

1. tree -> nodekind 가 statement node kind일 경우

genStmt() 함수를 호출해 statement node에 대한 code generation을 한다. 이번 프로젝트에서는 compound, selection, iteration, return인 경우데 대해 구현해야 한다. 먼저 compound일 경우에는 scope\_number\_for\_type을 증가시키고 child를 recursive로 cGen() 함수를 호출한다. 다음으로 selection일 경우에는 앞에서와 마찬가지로 child를 recursive로 cGen() 함수를 호출한다. 그 후 if문인지 else 문인지 구별해 각각의 flow에 대한 code를 generation한다. 다음으로 iteration일 경우에는 앞에서와 마찬가지로 child를 recursive로 cGen() 함수를 호출한다. 그 후 while문의 flow에 대한 code를 generation 한다. 마지막으로 return일 경우에는 child를 recursive로 cGen() 함수를 호출 한 뒤 return 값을 register에 저장한다.

1. tree -> nodekind 가 expression node kind일 경우

genExp() 함수를 호출해 expression nod에 대한 code generation을 한다. 이번 프로젝트에서는 op, const, id, assign, call, array id인 경우에 대해 구현해야 한다. 먼저 op일 경우에는 child를 recursive로 cGen() 함수를 호출 한 뒤, operation의 left와 right 값을 register에 저장해 plus, minus, mul, backslash, lt, lte, gt, gte, eq, neq에 대한 code를 generation 한 뒤 계산된 값을 다시 stack에 push 한다. 다음으로 const일 경우에는 값을 가져와 stack에 push 한다. 다음으로 id일 경우에는 먼저 parameter의 배열 여부를 확인한 뒤 global일 경우, parameter일 경우, local 일 경우에 따라 나누어 각각 code를 generation 한 뒤, 가져온 값을 stack에 push 한다. 다음으로 assign일 경우 child를 recursive로 cGen() 함수를 호출 한 뒤 값들을 register로 가져와 assign의 left 변수의 주소에 right 변수의 값을 저장하는 code를 generation 한다. 다음으로 call일 경우 argument list 즉, child를 마찬가지로 recursive로 cGen() 함수를 호출해 flow에 맞는 code를 generation 한다. 만약 return 타입이 integer이면 그 값을 가져온다. 마지막으로 array id 일 경우에는 index 값을 가져오기 위해 child를 recursive로 cGen() 함수를 호출해 register에 저장한 뒤 index에 따른 code를 generation 한 뒤 그 값을 stack에 push 한다.

1. tree -> nodekind 가 declaration node kind일 경우

genDecl() 함수를 호출해 declaration node에 대한 code generation을 한다. 이번 프로젝트에서는 variable일 경우와 function인 경우에 대해 구현해야 한다. 먼저 var 일 경우에는 array가 아니면 integer size인 4만큼 공간을 할당하고 배열일 경우에는 array\_size\*4 만큼 공간을 할당한 뒤 sp를 조정한다. 다음으로 function일 경우에는 새로운 scope가 생기므로 scope\_number\_for\_type을 1증가 시키고 child를 recursive로 cGen() 함수를 호출 한 뒤 나중에 call 할 경우 공간을 잡기 위해 parameter 개수를 또 다른 stack 형식으로 차례로 저장한다. 함수 처음 시작 시, activation record관리를 위한 공간을 잡는다. 맨 위부터 t1, t2 register를 백업하기 위한 공간을 잡는다. 그 다음 이전 frame pointer를 저장하기 위한 공간이다. 마지막으로 return address를 저장하기 위한 공간이다. 총 16byte의 고정된 공간으로 써 모든 함수가 가지고 있는 공간이다. 그 다음으로 함수의 state들을 처리해주기 위해 node의 child[2]를 cGen() 함수의 parameter로 넘겨 호출한다. 마지막으로 함수가 끝날 때 저장해 놓았던 return address, frame pointer와 t0, t1 register를 되돌려 놓고 stack pointer를 함수가 생성되기 전으로 돌려놓는다.

1. **제작 내용:**

* **symtab.c:**

1. **변수**

* **추가된 변수 없음**

1. **함수**

* ScopeList\* getScopeTypeHead()

: ScopeTypeHead를 반환한다.

* BucketList getBucket(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 BucketList를 리턴하고 그렇지 않으면 bucket에서 scope의 number를 가져와 scope가 0보다 크거나 같을 때까지 찾는다. 그 후 만약 variable을 찾는다면 현재 BucketList를 반환한다.

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

: getBucket() 함수와 마찬가지 flow로 진행해서 찾으면 현재 ScopeList를 반환한다.

* **cgen.c**

1. **변수**

* int label\_num: label을 사용할 때 label들을 식별하기 위한 number 변수
* int flag: scope\_number\_for\_type가 하나의 compound statement에서 중복해서 증가하는 것을 방지하기 위한 변수
* int scope\_number\_for\_type: symbol table들을 구별하기 위한 각 symbol table들의 고유 number 변수
* int base: stack에서 값을 가져올 때 기존의 symbol table의 offset으로 접근하기 위해 그 전의 위치를 계산해서 실제 address를 가져오기 위한 변수
* int param\_cnt[]: 각 함수들의 parameter 개수를 저장하고 있는 배열
* int top\_param\_cnt: param\_cnt[] 배열의 top을 가리키고 있는 변수
* int top: free space의 top을 가리키고 있는 변수
* int param\_flag: 현재 node가 argument상태인지 check 해주는 변수

1. **함수**

* void pop\_stack(int num)

: activation record의 stack 영역에 있는 값을 원하는 register로 값을 저장하는 code를 generation하는 함수로 원하는 number(parameter에서 num)의 t register에 값을 load 하고 sp값을 증가시켜 stack의 끝을 이동한다. top pointer의 값을 1 감소 시켜 저장된 개수를 감소시킨다.

* void push\_stack(void)

: activation record의 stack 영역에 값을 저장하는 code를 generation하는 함수로 sp의 값을 감소시켜 stack의 끝을 이동하고 t0 register의 값을 stack에 push한다. top pointer의 값을 1증가 시켜 저장된 개수를 증가시킨다.

* void genStmt(TreeNode \*tree, int is\_addr)

: tree의 node가 statement일 경우 code를 generation하는 함수로 is\_addr이 0이면 값을 1이면 주소를 가져온다. 구현해야 하는 종류는 다음과 같다.

1. Compound: scope\_number\_for\_type을 1증가시켜 scope마다의 symbol table을 구별한다. 그 후 cGen() 함수를 호출해 recursive하게 child의 code를 generation 한다.
2. Selection: 먼저 if 문일 경우와 if-else문일 경우 2가지로 나누어 다음의 P code의 flow에 맞게 code를 generation 한다. Code의 내용들은 cGen() 함수를 호출해 recursive하게 child의 code를 generation 한다.
3. If 문

<code to evaluate E>

Fjp L1

<code for S>

Lab L1

1. If-else 문

<code to evaluate E>

Fjp L1

<code for S1>

Ujp L2

Lab L1

<code for S2>

Lab L2

1. Iteration: while문일 경우 다음의 P code의 flow에 맞게 code를 generation 한다. Code의 내용들은 cGen() 함수를 호출해 recursive하게 child의 code를 generation 한다.

While 문

Lab L1

<code to evaluate E>

Fjp L2

<code for S>

Ujp L1

Labl L2

1. Return: cGen() 함수를 호출해 recursive하게 child의 code를 generation 한다. 그 후 그 결과 값을 stack에서 pop해 $t2 register에 저장한다.

* void genExp(TreeNode \*tree, int is\_addr)

: tree의 node가 expression일 경우 code를 generation하는 함수로 is\_addr이 0이면 값을 1이면 주소를 가져온다. 구현해야 하는 종류는 다음과 같다.

1. Op: cGen() 함수를 호출해 recursive하게 child의 code를 generation 한다. 그 후 그 결과 값을 stack에서 pop해 right value는 t0 register에, left value는 t1 register에 저장한다. 그 후 operation 종류에 따라 code를 generation 한다. Spim machine에서 대응되는 명령어는 정리한 합성내용을 참고하면 된다.
2. Const: const의 실제 값(tree->attr.val)을 t0 register에 load 한 뒤 stack에 push 하는 code를 generation 한다.
3. Id: variable이 존재하는 symbol table의 bucket list를 얻는다. 만약 array가 아니라면 다음의 3가지 경우에 따라 분류해 memory location을 계산해 stack에 push 하는 code를 generation 한다.
4. Data area: tree -> attr.name을 가져와 data 영역에서의 변수 위치나 값을 t0 register에 load 하는 code를 generation 한다.
5. Parameter: 개발내용에서 보여준 activation record를 참고하면 parameter의 location은 caller의 parameter 개수(param\_cnt [top\_param\_cnt-1]) + 4(control link. Return address, t0 값, t1값) – 1(포인터가 아래를 가리켜야 하므로) – l -> memloc 이다. 그 후 t0 register에 앞에서 구한 location의 변수 위치나 값을 load 하는 code를 generation 한다.
6. Local variable: getScopeList() 함수를 호출해 현재 변수의 scope를 가져와 calBase() 함수를 호출해 추가 base offset을 구한다. 그 후 개발내용에서 보여준 activation record를 보면 현재 변수의 주소는 l->memloc – caller의 parameter 개수(param\_cnt [top\_param\_cnt-1]) + 1을 해서 구한 뒤 위에서 구한 base값을 구해 실제 location을 구한다. 그러나 local은 stack의 방향이 밑으로 내려가므로 값을 음수로 바꾼 뒤 t0 register에 앞에서 구한 location의 변수 위치나 값을 load 하는 code를 generation 한다.

반대의 경우 즉, variable이 array인 경우 즉, 현재 id가 argument의 인자이고(param\_flag == 1), array일 경우(l->is\_array == 1) 다음의 2가지 경우에 따라 분류해 memory location을 계산해 stack에 push 하는 code를 generation 한다.

1. Data area: tree -> attr.name을 가져와 data 영역에서의 변수 위치를 t0 register에 load 하는 code를 generation 한다.
2. Local variable: getScopeList() 함수를 호출해 현재 변수의 scope를 가져와 calBase() 함수를 호출해 추가 base offset을 구한다. 그 후 개발내용에서 보여준 activation record를 보면 array의 첫 번째 주소는 l -> array\_size + l -> memloc이므로 base를 더해 실제 location을 구해 t0 register에 변수 위치를 load 하는 code를 generation 한다. 이는 data area의 배열이 저장되는 순서랑 맞추기 위함이다.
3. Assign: cGen() 함수를 호출해 recursive하게 child의 code를 generation 한다. 그 후 stack에서 pop해서 t0에는 assign의 오른쪽 value를 t1에는 assign의 왼쪽 address를 저장한다. 그 후 t0의 값을 t1에 저장하는 code를 generation 한다. 만약 top 0이 아닌 경우 즉, 마지막 assign이 아닌 경우 assign된 결과 값을 다시 push한다.
4. Call: cGen() 함수를 호출해 recursive하게 child의 code를 generation 한다. 여기서는 다음의 id나 const들은 argument임을 표시하기 위해 param\_flag를 1로 세팅해준다. 그 후 cGen() 함수 호출이 끝나면 다시 param\_flag 값을 0으로 클리어 해준다. 그 후 호출된 함수로 점프하는 code를 generation 한다. 추가로 만약 return type이 integer인 경우 그 결과 값(t2 register에 저장됨)을 stack에 push 하는 code를 generation 한다.
5. ArrayId: variable이 존재하는 symbol table의 bucket list를 얻는다. 다음의 3가지 경우에 따라 분류해 memory location을 계산해 stack에 push 하는 code를 generation 한다. 추가로 index 값을 가져오기 위해 cGen() 함수를 호출해 recursive하게 child의 값을 구한 뒤, 그 값을 t1 register에 저장하는 code를 generation 한다.
6. Data area: integer size만큼 이동하기 위해 index(t1 register 값)의 값에 4를 곱하고 tree -> attr.name으로 첫 번째 배열 주소의 위치를 load한 뒤 t1 register의 값만큼 fp를 더 이동한 뒤 stack에 그 값을 push하는 code를 generation 한다.
7. Parameter: integer size만큼 이동하기 위해 index(t1 register값)의 값에 4를 곱하고 id에서 구한 같은 방법으로 실제 location을 구해 값을 load한 뒤 t1 register의 값만큼 fp를 더 이동한 뒤 stack에 그 값을 push하는 code를 generation 한다.
8. Local variable: integer size만큼 이동하기 위해 index(t1 register 값)의 값에 4를 곱하고 id에서 구한 같은 방법으로 실제 location을 구해 주소를 load한 뒤 t1 register의 값만큼 fp를 더 이동한 뒤 stack에 그 값을 push하는 code를 generation 한다.

* void genDecl(TreeNode \*tree, int is\_addr)

: tree의 node가 statement일 경우 code를 generation하는 함수로 is\_addr이 0이면 값을 1이면 주소를 가져온다. 구현해야 하는 종류는 다음과 같다.

1. Var: 변수를 선언한 경우로 array가 아니라면 sp만 activation record에서 한 칸 내려 공간을 만들고 array라면 cGen()함수를 호출해 recursive하게 child의 값을 얻어 array의 size만큼 공간을 만든다.
2. Func: 새로운 scope가 생기므로 scope\_number\_for\_type을 1증가 시키고 child를 recursive로 cGen() 함수를 호출 한 뒤 나중에 call 할 경우 공간을 잡기 위해 parameter 개수를 또 다른 stack 형식으로 차례로 저장한다. 함수 처음 시작 시, activation record관리를 위한 공간을 잡는다. 맨 위부터 t1, t2 register를 백업하기 위한 공간을 잡는다. 그 다음 이전 frame pointer를 저장하기 위한 공간이다. 마지막으로 return address를 저장하기 위한 공간이다. 총 16byte의 고정된 공간으로 써 모든 함수가 가지고 있는 공간이다. 그 다음으로 함수의 state들을 처리해주기 위해 node의 child[2]를 cGen() 함수의 parameter로 넘겨 호출한다. 마지막으로 함수가 끝날 때 저장해 놓았던 return address, frame pointer와 t0, t1 register를 되돌려 놓고 stack pointer를 함수가 생성되기 전으로 돌려놓는다.

* void cGen(TreeNode \*tree, int is\_addr)

: tree가 NULL이 아닐경우 tree -> nodekind의 값이 statement인지 expression인지 declaration인지 구분해 각각 genStmt(), genExp(), genDecl() 함수를 호출한다. 그 후 tree -> sibling을 recursive하게 cGen() 함수를 호출한다.

* void codeGen(TreeNode \*SyntaxTreem char \*codefile)

: 생성할 파일을 결정하고 먼저 data area에 있어야 할 변수나 함수들을 저장 한 뒤 cGen() 함수를 가장 먼저 호출해 syntaxTree의 root부터 code generation을 실질적으로 순회하기 시작한다. 먼저 data area에 있어야 할 변수는 global 변수들로 scope가 0인 symbol table에 있는 변수들이다. Scope가 0인 ScopeList를 찾아 모든 hash 값을 순회하며 data area에 load하는 code를 generation 한다. 다음으로 기본으로 내장되어 있는 함수 input()과 output() 함수의 body를 미리 code generation 한다. 위에서 declaration의 func일 경우와 마찬가지로 code를 generation하는데 다른점은 body가 정해져 있으므로 그 body 내용을 작성해야 하는데 두 함수의 process는 다음과 같다.

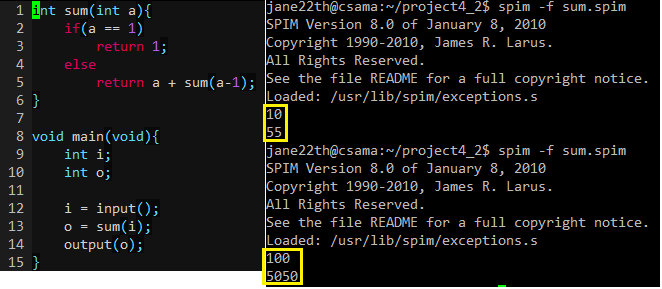
1. Output(): register v0에 syscall number 1을 저장하고 syscall을 하면 register a0에 있는 값이 출력되는 code를 generation한다.
2. Input(): register v0에 syscall number 5를 저장하고 syscall을 하면 register v0에 들어온 값이 저장되어 있는 code를 generation 한다.

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

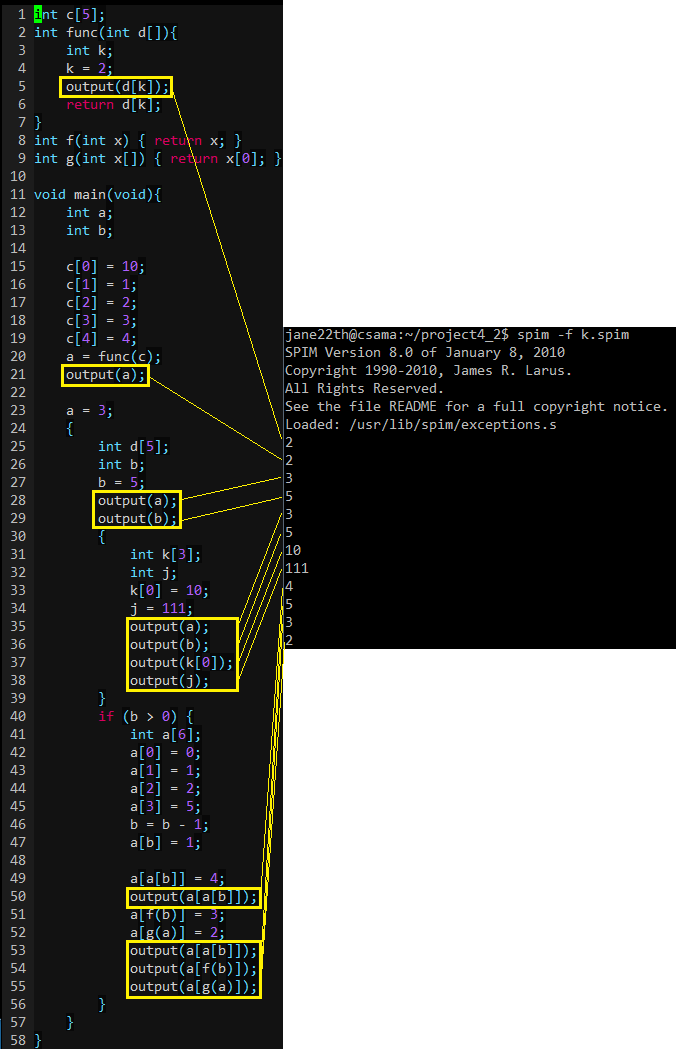
: 현재 scope symbol table은 scope마다 offset이 초기화가 되서 nested declaration일 경우 location의 위치가 중복될 염려가 있다. 따라서 같은 함수라면 실질적 offset을 구해주는 이 함수가 필요하다. Argument로 넘어온 scope\_number\_for\_type과 일치하는 ScopeList를 가져와 그 다음 ScopeList부터 같은 함수 내의 nested scope이면 모든 hash를 돌면서 array가 아니라면 base의 값을 1증가시키고 array라면 array size만큼 값을 증가시켜 실제 offset을 계산한다.

**4. 시험 내용:**   
code generation 구현 시 가장 까다로웠던 부분인 compound statement와 array handling, 그리고 이를 구현하는데 기반이 되는 activation record 구조 확립을 확인하는 데에 아래의 두 테스트 코드를 이용하였다.

- test1.c  
1부터 입력 받은 자연수까지의 자연수들의 합을 recursive하게 구한다. Recursive function을 통한 test는 activation record를 올바르게 확립했는지 확인할 수 있는 가장 좋은 방법이라 채택되었다. 오른편의 test case는 1부터 10까지의 합 55와 1부터 100까지의 합 5050을 구하는 case로서, 올바르게 출력됨을 확인할 수 있다.



* test2.c  
  array를 handling하는 여러 가지 케이스를 시험할 수 있는 test code로서, array를 parameter로써 passing하는 경우, return value로 사용하는 경우, 선언된 scope 내/외에서의 indexing, nested indexing, function return value를 통한 indexing 의 경우를 테스트 하였다.



1. **평가 내용:**   
    이제 구현한 compiler는 C- Language로 쓰인 모든 소스코드를 SPIM machine 상에서 구동할 수 있는 완성된 compiler가 되었다.  
    code generation 파트를 구현하는 동안 parameter passing rule과 array를 passing 하는 데에 관한 어려움을 겪었다. SPIM machine은 parameter passing에 제한된 register의 개수를 사용하기 때문에, parameter를 register와 stack을 동시해 활용하여 passing을 하는 대신 stack을 통해서만 이루어지도록 구현하였다. 이는 향후에 수정하여야 할, register를 최대한 활용하여 효율성을 극대화하는 작업 전까지 user program이 안정성 있게 컴파일 되고 구동되는 것에 대한 적절한 조치라고 할 수 있다.  
    SPIM machine simulator는 James R. Larus의 open source license인 SPIM 8.0이 사용되었으며 이에 따른 신뢰할 수 있는 환경에서 개발/디버깅이 이루어졌다.

**V. 기타**

**1. 연구 조원 기여도:**

**- 이상범:** 33.33%

**- 모지수:** 33.33%

**- 박영훈:** 33.33%

**2. 자체 평가:**

2조는 SPIM machine상에서 동작하는 code generator를 개발하는 것으로 진취적 목표를 잡았다. SPIM machine에 대한 충분한 이해를 바탕으로 하여 시작하여야 했기 때문에 SPIM machine의 assembly language와 activation record의 형태, memory stack 구조, 레지스터들 등을 공부하는 것부터 시작하였다.  
 수업 시간에 다룬 범위의 내용만을 바탕으로 한 TM machine의 컴파일러를 구현하는 것을 넘어서 추가 구현을 해낸 2조는 기초컴파일러구성 과목의 성취도가 높게 평가된다.

**3. 느낀점:**

**- 이상범:** 처음 project description을 읽고 tiny code generator를 분석했을 때, 생각보다 종류만 잘 분류하면 code generation을 쉽게 할 수 있을 거라 생각했는데 SPIM machine에 대한 이해와 Runtime environment에 대한 이해가 어려워서 생각보다 직접 구현하는데 까지 많은 시간이 걸렸다. 다행히 이번엔 project3와 달리 flow를 확실히 알고 작성을 해서 예상외의 에러가 존재 하지 않을 것이라고 확신한다. 원하는 값이 나오지 않을 경우 직접 activation record를 하나 하나 fp와 sp를 따져가며 해야 해서 짜증이 많이 났다. 그러나 팀 프로젝트였기 때문에 각자의 아이디어가 잘 발휘해 복잡한 케이스까지 잘 generation 한 것 같아서 뿌듯하다.

**- 모지수:** 주어진 TM machine이 아닌, 직접 찾아 공부 해야 하는 SPIM machine을 구현하는 데에 처음엔 불안한 감이 없지 않았지만 SPIM machine의 어셈블리언어와 메모리 stack 환경에 대해 자세한 매뉴얼을 참고 할 수 있어서 초기 작업에 많은 도움이 되었다. 또한 추가 구현 과제이기에 기초컴파일러구성 과목의 복습이나 숙제를 하는 기분이 아닌, 현장에서 프로젝트를 개발하는 듯한 느낌이 들어 흥미롭게 진행할 수 있었다. 크게는 네 번에 걸쳐 하나의 컴파일러 프로젝트를 결국 완성하였다는 면에선, 기수강한 Programming language, Assembly Programming, System Programming 등의 과목들을 하나로 아우를 수 있는, 놓치면 아쉬울 뻔했을 기회가 되었다는 생각이 든다.

**- 박영훈:** 수업 때 배웠던 three address처럼 간단하지 않아서 많이 고생했다. SPIM을 이해하는데 너무 많은 시간이 걸렸다. 하지만 이해하고 나니 코드가 금방 구성되었다. 사용되는 레지스터를 백업하지 않아 에러가 나기도 하여 고생했다. 코드를 생성할 때 indent가 없어 에러를 찾기 위해 SPIM code를 읽을 때 정말 힘들었다.