

REPORT

Code Generation Implementation



전공 : 소프트웨어전공

학번 : 2013011167

이름 : 진 용 석

1. 구현 방법 및 주요 소스코드 설명

이번 과제로 구현한 C-Minus 컴파일러가 컴파일하는 과정을 요약하면 다음과 같습니다.

1) Tiny Machine 에서 동작할 수 있도록 초기화하는 코드를 삽입합니다.

stack pointer(memory pointer), frame pointer, global pointer 를 메모리의 가장 큰 주소 (memory stack 의 바닥)로 설정합니다. 또한, 구현상의 편리함을 위해 3번 레지스터를 constant 레지스터로 설정합니다. constant 의 값은 1입니다.

2) syntax tree 를 따라 코드를 생성합니다.

이 과정은 아래에서 더 자세히 설명하도록 하겠습니다.

3) global 변수의 메모리만큼 stack 메모리를 push 하고, main 함수를 호출합니다.

global 변수는 스택의 최하단에 위치하도록 구현하였습니다. 따라서 main 호출 전에, global 변수의 메모리만큼 스택을 push 하도록 설정하였습니다.

main 함수의 주소는 syntax tree 를 따라 코드를 생성하는 과정에서 확보하게 됩니다. 함수의 이름이 "main" 인 경우, 해당 주소를 저장합니다. 그 이후, syntax tree 탐색이 끝나면 main 함수를 호출하도록 설정하였습니다.

여기서 말하는 '함수 호출' 이란, 해당 함수를 종료하고 나면 다시 되돌아올 return address 를 stack 에 저장하고 해당 함수로 jump 하는 행위 등을 포함합니다.

4) HALT 명령어로 종료시킵니다.

main 함수가 끝나면 프로그램을 종료시킵니다.

컴파일러는 syntax tree 를 탐색하며, 탐색하는 node 에 따라 코드를 생성합니다. 생성하는 코드는 다음과 같습니다.

1) Declaration Node

선언의 종류는 '함수', '변수', '파라미터' 총 세 가지로 구분합니다. 이 중 함수와 변수는 IdDec 라는 키워드로 동일하게 구분되어 있으며, 컴파일 시에 해당 노드의 child[1] 의 특성으로 분류합니다.

[함수]를 선언하는 경우에는 parameter 를 위해 임시 scope 를 설정합니다. 그리고 현재 함수의 주소를 저장합니다. main 인 경우 이후에 jump 하기 위해 다른 변수에도 따로 주소를 저장합니다.

```

/* get location of function. */
loc = st_lookup(currentTable, tree->attr.name)->location;

/* store current location to array. */
currentLoc = emitSkip(0);
functionLocations[loc] = currentLoc;

/* store main function's location. */
if (strcmp(tree->attr.name, "main") == 0)
{
    mainFunctionLoc = currentLoc;
}

```

또한, 해당 함수의 메모리 영역을 설정하기 위해 stack pointer 와 frame pointer 를 설정합니다. 기본적으로 stack pointer 와 frame pointer 를 3씩 감소시키는데, 현재 frame pointer 위치와 return address 의 주소를 저장하기 위함입니다. 지역 변수의 위치는 frame pointer - offset(offset은 0부터 시작) 로 설정하고 있습니다.

```

/* push previous frame pointer address. */
emitRM("ST", fp, -2, mp, "store previous frame pointer address.");

/* set frame pointer. */
emitRM_Abs("LDA", ac, 3, "load value 3 to ac.");
emitRO("SUB", fp, mp, ac, "fp = mp - 3");
emitRO("SUB", mp, mp, ac, "mp = mp - 3");

```

함수 선언 노드의 child[0] 은 변수 선언, child[1] 은 compound statement 가 존재합니다. 해당 노드들의 코드를 마찬가지로 cGen 함수를 통해 생성한 뒤, return 구문을 삽입합니다. return 구문은 stack pointer 와 frame pointer 를 원상복구시키고, pc 를 이전의 주소 + 1 로 설정하는 동작을 포함합니다.

ac 레지스터는 해당 함수의 return 값을 가집니다. 따라서, return 시에 절대로 ac 레지스터의 값을 변경하지 않도록 합니다.

```

/* create return instruction :
do not use ac, since it has return value. */
emitComment("Return Statements.");
emitRM_Abs("LDA", ac1, 3, "load value 3 to ac1.");
emitRO("ADD", mp, fp, ac1, "mp = fp + 3");
emitRM("LD", fp, 1, fp, "set fp to previous frame pointer.");
emitRM("LD", ac1, -1, mp, "set ac1 to previous address.");
emitRO("ADD", pc, ac1, constant, "pc = previous address + 1");
emitComment("Return Statements ended.");

```

[변수] 선언 시에는 해당 변수가 지역 변수인지 전역 변수인지를 확인합니다. 전역 변수인 경우에는 globalOffset, 지역 변수인 경우에는 localOffset 의 값을 size 만큼 더합니다. 이는 항상 지역 변수 인 [파라미터] 선언의 경우에도 동일하게 적용됩니다.

2) Statement Node

statement 노드는 Compound, Selection, Iteration, Return statement 로 구분됩니다.

[Compound] 는 지역 변수 선언을 포함합니다. 따라서 scope 를 설정하고, 지역 변수 선언 처리가 끝난 뒤(이 경우에는 localOffset 을 계산합니다), stack pointer 를 push 합니다. push 된 stack pointer 는 이후 compound statement 가 종료하면 pop하도록 구현하였습니다.

```
/* calculate local offset. */
cGen(tree->child[0]);

/* store current localOffset. */
offset = localOffset;

/* set stack pointer. */
emitRM_Abs("LDA", ac, offset, "load size of local vars to ac.");
emitRO("SUB", mp, mp, ac, "mp = mp - localOffset");

/* set local offset into 0, since setting stack pointer is finished. */
localOffset = 0;
```

[Selection] 은 조건식 이후 분기문으로 나뉩니다. 조건식을 가지는 child[0] 의 코드를 생성하고 나면 ac 레지스터에 결과값이 저장됩니다. 0이면 true, 그 이외에는 false 로 처리합니다. true 이면 firstBlock, false 이면 secondBlock 으로 jump 하도록 구현하였습니다. secondBlock 은 else 구문이 있는 경우 else 구문으로, 없는 경우에는 firstBlock 의 직후로 설정하였습니다.

```
/* set jump condition to restored area. */
emitBackup(firstLoc);
emitRM_Abs("JEQ", ac, firstBlock, "jump to firstBlock if ac == 0.");
emitRM_Abs("JNE", ac, secondBlock, "jump to secondBlock if ac != 0.");
```

[Iteration] 은 Selection 과 매우 유사한 구조를 가집니다. 차이점은 firstBlock 직후의 unconditional jump 를 통해 조건식을 계산하는 곳으로 다시 이동한다는 점입니다.

[Return] 은 상당히 단순한 구조를 가집니다. child[0] 의 코드를 cGen 으로 생성한 뒤 종료합니다. 해당 expression 코드가 실행되고 나면 ac 레지스터에 return value 가 저장됩니다.

```
/* child[0] : expression or NULL */
case ReturnStmt:
    /* generate code for expression. */
    cGen(tree->child[0]);

    /* returned value is already in register 'ac'. */
```

3) Expression Node

Expression node 는 ID, Const, Operator 로 구분됩니다.

[**Operator**] 의 경우에는 child[0], child[1] 을 가집니다.

ASSIGN operator 의 경우에는 child[0] 은 reference 형태를 가지게 되며, child[1] 의 값을 child[0] 의 위치에 저장하게 됩니다. 그 이외의 operator 의 경우에는 child[0], child[1] 의 값을 참조하며, operator 에 따라 계산 결과를 산출합니다. 따라서, ASSIGN 과 다른 operator 를 구분하여 구현하였습니다.

Operator 의 모든 경우에 대해, 우항은 expression 의 결과값을 가지게 됩니다. 따라서 우항을 먼저 계산한 뒤, 레지스터가 충돌할 수 있으니 stack 에 push 하도록 구현하였습니다. 좌항의 값을 계산한 뒤 이 값은 pop 되어, ac1 레지스터에 저장하여 계산하도록 구현하였습니다.

```
/* get expression value from right. */
cGen(tree->child[1]);

/* store right expression value on stack. */
emitRM("ST", ac, -1, mp, "mem[mp - 1] = right expression");

/* push mp 1. */
emitRO("SUB", mp, mp, constant, "mp = mp - 1");
```

[**ID**] 의 경우에는 여러 경우의 수가 있습니다. 해당 ID 가 function 인 경우에는 function call 이 됩니다. 우선 Built-in 함수인 input 과 output 함수를 처리하고, 그 이외의 경우에는 parameter 를 push 한 뒤 함수를 호출하도록 구현하였습니다. 그 뒤, 해당 함수를 호출하도록 하였습니다.

```
/* get function's real location. */
location = functionLocations[location];

/* set function variables. */
param = tree->child[0];
offset = -3; /* above sfp, return address. */
emitComment("putting arguments");
while(param != NULL)
{
    /* generate expression. */
    genExp(param);

    /* store ac to stack. */
    emitRM("ST", ac, offset, mp, "memory[mp+offset] = ac");

    /* advance. */
    offset--;
    param = param->sibling;
}
emitComment("argument put on stack");

/* call function. */
emitComment("Function Call Statements.");
emitRM("ST", pc, -1, mp, "store return address to stack");
emitRM_Abs("LDA", pc, location, "jump to function");
emitComment("Function Call Statements ended.");
```

해당 ID 가 Array 인 경우에도 여러 경우의 수가 있습니다. child[0] 이 NULL 인 경우에는 해당 array 자체를 가리킵니다. C-Minus 에서 해당 연산이 일어나는 경우는 parameter 전달이 있습니다. 따라서, 이 array 를 그대로 가리키도록 만들기 위해 reference 를 전달하도록 구현하였습니다.

해당 ID 가 Array 이며 child[0] 이 존재하는 경우에는, array[index] 의 값을 호출하는 경우입니다. 해당 변수가 global 변수인 경우에는 global pointer 로부터 offset 만큼 뺀 값으로 처리하고, local 변수인 경우에는 frame pointer 로부터 offset 만큼 뺀 값으로 처리합니다.

```
/* if called array itself, return reference. */
if (tree->child[0] == NULL)
{
    emitRM_Abs("LDA", ac, location, "load -location to ac");
    /* global variable : gp - location */
    if (var->is_global == 1)
    {
        emitRO("ADD", ac, gp, ac, "ac = gp - location");
    }
    /* local variable : fp - location */
    else
    {
        emitRO("ADD", ac, fp, ac, "ac = fp - location");
    }
}
/* global variable : subtract from global pointer. */
else if (var->is_global == 1)
{
    emitRO("SUB", ac1, gp, ac, "ac1 = gp - offset");
    emitRM("LD", ac, location, ac1, "ac = memory[ac1 - location]");
}
/* local variable : subtract from frame pointer. */
else
{
    emitRO("SUB", ac1, fp, ac, "ac1 = fp - offset");
    emitRM("LD", ac, location, ac1, "ac = memory[ac1 - location]");
}
```

해당 ID 가 일반 변수인 경우에는 값을 참조하는 경우로, 위의 array[index] 의 값을 호출하는 경우와 동일하게 적용합니다. 해당 변수가 global 변수인 경우에는 global pointer 로부터 offset 만큼 뺀 값으로 처리하고, local 변수인 경우에는 frame pointer 로부터 offset 만큼 뺀 값으로 처리합니다.

[**Const**] 의 경우에는 단순히, tree node 에 저장되어 있는 attr.val 값을 그대로 ac 에 저장합니다. 이 값은 이후 계산에서 변수의 값 등과 동일하게 사용하게 됩니다.

컴파일의 용이함을 위해 심볼 테이블에 일부 정보들을 추가하였습니다. 추가한 정보들은 다음과 같습니다.

```
20
21 /* The record in the bucket lists for
22  * each variable, including name,
23  * assigned memory location, and
24  * the list of line numbers in which
25  * it appears in the source code
26  */
27 typedef struct BucketListRec
28 {
29     char * name;
30     int lineno;
31     int is_function;
32     Type type;
33     struct BucketListRec * next;
34     struct BucketListRec * param;
35
36     int is_param;
37     int is_global;
38     int location; /* address that this symbol is stored in */
39 } *BucketList;
```

is_param, is_global, location 변수가 추가되었습니다.

is_param 은 해당 변수가 해당 함수의 파라미터인지를 나타냅니다. 파라미터에 array 가 들어가게 되면 해당 array 를 참조형으로 전달해야 하기 때문에, 해당 변수가 파라미터인지에 따라 접근 방식이 달라지게 됩니다. 따라서 이 변수를 추가하여 컴파일 시 참조하도록 했습니다.

is_global 은 해당 변수가 global 변수인지, local 변수인지를 나타냅니다. 해당 변수를 global pointer 로부터 계산할 것인지, frame pointer 로부터 계산할 것인지를 결정합니다.

location 은 해당 변수의 위치 값입니다. 정확히는 global / frame pointer 로부터 떨어져 있는 거리, offset 을 가집니다. 컴파일 시 위치를 불러오고, 해당하는 레지스터의 값에서 빼는 방식으로 계산합니다.

이와 같은 변수들을 추가하여, 복잡하게 접근할 필요 없이 컴파일 시 symbol table 참조만으로 컴파일을 용이하게 진행할 수 있도록 하였습니다. 데이터 추가에 따라, symbol table 에 symble 을 추가하도록 하는 함수 st_insert 또한 수정하였습니다.

2. 컴파일 방법 및 환경

이 프로젝트는 Ubuntu 14.04, 커널 버전 3.13.0-34-generic 에서 개발 및 테스트하였습니다. 바이너리 명은 이전과 다르게, cminus 로 지정하였습니다. 가상 머신인 tm 과 cminus 를 모두 컴파일 하여 실행합니다. 또한, make clean 명령어를 통해 중간 생성물과 최종 생성된 바이너리를 삭제할 수 있습니다. 컴파일 명령어는 다음과 같습니다.

```
$ make cminus tm
```

컴파일된 산출물은 기존 경로의 이름과 동일하며, 확장자 명만 .cm 에서 .tm 으로 변경됩니다. .cm 소스 코드를 컴파일하는 명령어는 다음과 같습니다.

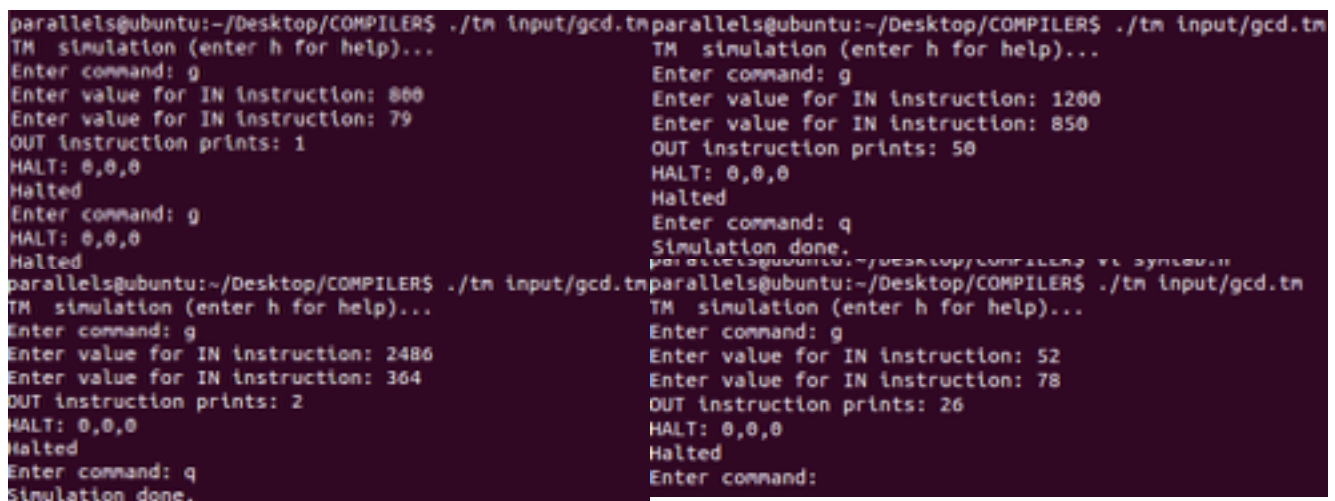
```
$ ./cminus [source_path]
```

컴파일된 코드를 실행하는 명령어는 다음과 같습니다.

```
$ ./tm [code_path]
```

3. 예시 및 결과 화면

주어진 예시 중 gcd 계산 결과입니다. 입력한 값을 올바르게 받고 있으며, 재귀적 함수 호출을 통한 계산이 오류 없이 이루어지고 있습니다.



```
parallels@ubuntu:~/Desktop/COMPILERS$ ./tm input/gcd.tmparallels@ubuntu:~/Desktop/COMPILERS$ ./tm input/gcd.tn
TM simulation (enter h for help)...
Enter command: g
Enter value for IN instruction: 800
Enter value for IN instruction: 79
OUT instruction prints: 1
HALT: 0,0,0
Halted
Enter command: g
HALT: 0,0,0
Halted
parallels@ubuntu:~/Desktop/COMPILERS$ ./tm input/gcd.tmparallels@ubuntu:~/Desktop/COMPILERS$ ./tm input/gcd.tn
TM simulation (enter h for help)...
Enter command: g
Enter value for IN instruction: 2486
Enter value for IN instruction: 364
OUT instruction prints: 2
HALT: 0,0,0
Halted
Enter command: q
Simulation done.

TM simulation (enter h for help)...
Enter command: g
Enter value for IN instruction: 1200
Enter value for IN instruction: 850
OUT instruction prints: 50
HALT: 0,0,0
Halted
Enter command: q
Simulation done.
parallels@ubuntu:~/Desktop/COMPILERS$ ./tm input/gcd.tn
TM simulation (enter h for help)...
Enter command: g
Enter value for IN instruction: 52
Enter value for IN instruction: 78
OUT instruction prints: 26
HALT: 0,0,0
Halted
Enter command:
```

주어진 예시 중 sort 계산 결과입니다. 10번 입력한 값을 array 에 각각 입력받고 있으며, array parameter 전달을 통한 reference 연산도 정상적으로 이루어지고 있음을 확인할 수 있습니다.

<pre> parallels@ubuntu:~/Desktop/COMPILERS\$./tm input/input.tm TM simulation (enter h for help)... Enter command: g Enter value for IN instruction: 10 Enter value for IN instruction: 9 Enter value for IN instruction: 8 Enter value for IN instruction: 7 Enter value for IN instruction: 6 Enter value for IN instruction: 5 Enter value for IN instruction: 4 Enter value for IN instruction: 3 Enter value for IN instruction: 2 Enter value for IN instruction: 1 OUT instruction prints: 1 OUT instruction prints: 2 OUT instruction prints: 3 OUT instruction prints: 4 OUT instruction prints: 5 OUT instruction prints: 6 OUT instruction prints: 7 OUT instruction prints: 8 OUT instruction prints: 9 OUT instruction prints: 10 HALT: 0,0,0 Halted Enter command: q Simulation done. </pre>	<pre> parallels@ubuntu:~/Desktop/COMPILERS\$./tm input/input.tm TM simulation (enter h for help)... Enter command: g Enter value for IN instruction: 8 Enter value for IN instruction: 6 Enter value for IN instruction: 3 Enter value for IN instruction: 4 Enter value for IN instruction: 1 Enter value for IN instruction: 99 Enter value for IN instruction: 600 Enter value for IN instruction: 53 Enter value for IN instruction: 23 Enter value for IN instruction: 65 OUT instruction prints: 1 OUT instruction prints: 3 OUT instruction prints: 4 OUT instruction prints: 6 OUT instruction prints: 8 OUT instruction prints: 23 OUT instruction prints: 53 OUT instruction prints: 65 OUT instruction prints: 99 OUT instruction prints: 600 HALT: 0,0,0 Halted Enter command: q Simulation done. parallels@ubuntu:~/Desktop/COMPILERS\$ </pre>
---	--

개인적으로 따로 테스트한 예시 중 fibonacci 연산이 있습니다. gcd 와 유사한 재귀함수 구조이며, 특별한 오류 없이 잘 처리되고 있음을 확인할 수 있습니다.

<pre> 1 int fibo(int a) { 2 if (a == 1) return 1; 3 else { 4 if (a == 2) return 1; 5 else return fibo(a-1) + fibo(a-2); 6 } 7 } 8 9 void main(void) { 10 int x; 11 12 x = input(); 13 output(fibo(x)); 14 } </pre>	<pre> parallels@ubuntu:~/Desktop/COMPILERS\$./tm input/fibo.tm TM simulation (enter h for help)... Enter command: g Enter value for IN instruction: 1 OUT instruction prints: 1 HALT: 0,0,0 Halted Enter command: q Simulation done. parallels@ubuntu:~/Desktop/COMPILERS\$./tm input/fibo.tm TM simulation (enter h for help)... Enter command: 3 Command 3 unknown. Enter command: g Enter value for IN instruction: 3 OUT instruction prints: 2 HALT: 0,0,0 Halted </pre>
--	--

<pre> parallels@ubuntu:~/Desktop/COMPILERS\$./tm input/fibo.tm TM simulation (enter h for help)... Enter command: g Enter value for IN instruction: 8 OUT instruction prints: 21 HALT: 0,0,0 Halted Enter command: q Simulation done. </pre>	<pre> parallels@ubuntu:~/Desktop/COMPILERS\$./tm input/fibo.tm TM simulation (enter h for help)... Enter command: g Enter value for IN instruction: 15 OUT instruction prints: 610 HALT: 0,0,0 Halted Enter command: </pre>
---	--