REPORT

(컴파일러 기말고사)



|  |  |
| --- | --- |
| 제목 | 기말고사 |
| 제출일자 | 21.12.13 |
| 소속학과 | 컴퓨터공학과 |
| 학번 | 2017305039 |
| 성명 | 신동민 |

**문제) Mini C 프로그램에 대한 ICG를 구현하시오.**

**프로그램 동작**

1. 파일 입력 -> 2. 스캐너 -> 3. 파서 -> 4. SDT -> 5. ICG -> 6. U-code 출력

Perfect.mc

Bubble.mc

**파일을 입력**

**Scanner()**

**Parser()**

**SDT**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 이름 | 타입 | 타입 | base | offset | width |  |
| A | 변수 | 정수 | 2 | 1 | 1 |  |
| B | 변수 | 정수 | 2 | 2 | 1 |  |
| C | 변수 | 정수 | 2 | 3 | 1 |  |
|  |  |  |  |  |  |  |

**SymTab**

**트리탐색**

**ICG**

**변수 관리**

**U-code출력**

**종료**

**/\* scanner \*/**

**+**

**/\* parser \*/**

**+**

**/\* AST \*/**

**+**

**<ICG>**

// 연산자들의 이름마다 열거값을 지정

enum opcodeEnum {

notop, neg, incop, decop, dup,

add, sub, mult, divop, modop, swp,

andop, orop, gt, lt, ge, le, eq, ne,

lod, str, ldc, lda,

ujp, tjp, fjp,

chkh, chkl,

ldi, sti,

call, ret, retv, ldp, proc, endop,

nop, bgn, sym

};

// 연산자들의 이름들

const char\* opcodeName[] = {

"notop", "neg", "inc", "dec", "dup",

"add", "sub", "mult", "div", "mod", "swp",

"and", "or", "gt", "lt", "ge", "le", "eq", "ne",

"lod", "str", "ldc", "lda",

"ujp", "tjp", "fjp",

"chkh", "chkl",

"ldi", "sti",

"call", "ret", "retv", "ldp", "proc", "end",

"nop", "bgn", "sym"

};

// ICG에서 에러를 출력하는 함수

void icg\_error(int num)

{

printf("ICG\_ERROR: %d\n", num);

}

// 연산자만 있을 경우를 출력하는 함수

void emit0(const char\* value) {

fprintf(ucodeFile, " %s\n", value);

printf(" %s\n", value);

}

// 연산자와 1개에 비연산자가 있을 경우를 출력하는 함수

void emit1(const char\* value, int p) {

fprintf(ucodeFile, " %s %d\n", value, p);

printf(" %s %d\n", value, p);

}

// 연산자와 2개에 비연산자가 있을 경우를 출력하는 함수

void emit2(const char\* value, int p, int q) {

fprintf(ucodeFile, " %s %d %d\n", value, p, q);

printf(" %s %d %d\n", value, p, q);

}

// 지정 label로 이동하는 경우를 출력하는 함수

void emitJump(const char\* value, const char\* label) {

fprintf(ucodeFile, " %s %s\n", value, label);

printf(" %s %s\n", value, label);

}

// 지정 label를 출력하는 함수

void emitLabel(const char\* label) {

fprintf(ucodeFile, "%s nop\n", label);

printf("%s nop\n", label);

}

// 비연산자들 중에 연산에 필요한 값을 읽는 함수

void rv\_emit(NODE\* ptr)

{

int stIndex;

if (ptr->token.number == tnumber)

emit1("ldc", ptr->token.value.num);

else {

stIndex = lookup(ptr->token.value.id);

if (stIndex == -1) return;

if (symbolTable[stIndex].typeQualifier == CONST\_TYPE) // constant

emit1("ldc", symbolTable[stIndex].initialValue);

else if (symbolTable[stIndex].width > 1) { // array var

emit2("lda", symbolTable[stIndex].base, symbolTable[stIndex].offset);

}

else // simple var

emit2("lod", symbolTable[stIndex].base, symbolTable[stIndex].offset);

}

}

// 변수를 선언하는 sym을 만드는 함수

void emitSym(int base, int offset, int size) {

fprintf(ucodeFile, " %s %d %d %d\n", "sym", base, offset, size);

printf(" %s %d %d %d\n", "sym", base, offset, size);

}

// label를 만드는 함수

void genLabel(char\* label) {

sprintf(label, "$$%-8d", labelNum++);

}

// 변수에 대한 타입을 확인하는 함수

int typeSize(int typeSpecifier)

{

if (typeSpecifier == INT\_TYPE)

return 1;

else {

printf("not yet implemented\n");

return 1;

}

}

// 단순 변수인지 확인하고 맞다면 심벌테이블에 저장하는 함수

void processSimpleVariable(NODE\* ptr, int typeSpecifier, int typeQualifier)

{

NODE\* p = ptr->son; // variable name(=> identifier)

NODE\* q = ptr->brother; // initial value part

int stIndex, size, initialValue;

int sign = 1;

if (ptr->token.number != SIMPLE\_VAR) printf("error in SIMPLE\_VAR\n");

if (typeQualifier == CONST\_TYPE) { // constant type

if (q == NULL) {

printf("%s must have a constant value\n", ptr->son->token.value.id);

return;

}

if (q->token.number == UNARY\_MINUS) {

sign = -1;

q = q->son;

}

initialValue = sign \* q->token.value.num;

stIndex = insert(p->token.value.id, typeSpecifier, typeQualifier,

0/\*base\*/, 0/\*offset\*/, 0/\*width\*/, initialValue);

}

else {

size = typeSize(typeSpecifier);

stIndex = insert(p->token.value.id, typeSpecifier, typeQualifier, base, offset, width, 0);

offset += size;

}

}

// 배열 변수인지 확인하고 맞다면 심벌테이블에 저장하는 함수

void processArrayVariable(NODE\* ptr, int typeSpecifier, int typeQualifier)

{

NODE\* p = ptr->son; // variable name(=> identifier)

int stIndex, size;

if (ptr->token.number != ARRAY\_VAR) {

printf("error in ARRAY\_VAR\n");

return;

}

if (p->brother == NULL) // no size

printf("array size must be specified\n");

else size = p->brother->token.value.num;

size \*= typeSize(typeSpecifier);

stIndex = insert(p->token.value.id, typeSpecifier, typeQualifier, base, offset, size, 0);

offset += size;

}

// 선언을 위해서 어떤 변수인지 구분하는 함수

void processDeclaration(NODE\* ptr)

{

int typeSpecifier, typeQualifier;

NODE\* p, \* q;

if (ptr->token.number != DCL\_SPEC) icg\_error(4);

// printf("processDeclaration\n");

// step 1: process DCL\_SPEC

typeSpecifier = INT\_TYPE; // default type

typeQualifier = VAR\_TYPE;

p = ptr->son;

while (p) {

if (p->token.number == INT\_NODE) typeSpecifier = INT\_TYPE;

else if (p->token.number == CONST\_NODE)

typeQualifier = CONST\_TYPE;

else { // AUTO, EXTERN, REGISTER, FLOAT, DOUBLE, SIGNED, UNSIGEND

printf("not yet implemented\n");

return;

}

p = p->brother;

}

// step 2: process DCL\_ITEM

p = ptr->brother;

if (p->token.number != DCL\_ITEM) icg\_error(5);

while (p) {

q = p->son; // SIMPLE\_VAR or ARRAY\_VAR

switch (q->token.number) {

case SIMPLE\_VAR: // simple variable

processSimpleVariable(q, typeSpecifier, typeQualifier);

break;

case ARRAY\_VAR: // array variable

processArrayVariable(q, typeSpecifier, typeQualifier);

break;

default:

printf("error in SIMPLE\_VAR or ARRAY\_VAR\n");

break;

}

p = p->brother;

}

}

void processOperator(NODE\* ptr);

// if, read, write를 call로 부르는 경우를 출력하는 함수

int checkPredefined(NODE\* ptr) {

NODE\* p = ptr;

char\* functionName;

int noArguments;

int stIndex;

functionName = p->token.value.id;

if (strcmp("read", p->token.value.id) == 0) // read를 call했을 때

{

noArguments = 1;

emit0("ldp");

p = p->brother; // ACTUAL\_PARAM

while (p)

{

if (p->noderep == nonterm)

{

processOperator(p);

}

else {

stIndex = lookup(p->token.value.id);

if (stIndex == -1)

{

break;

}

emit2("lda", symbolTable[stIndex].base, symbolTable[stIndex].offset);

}

noArguments--;

p = p->brother;

}

if (noArguments > 0)

{

printf("%s: too few actual arguments\n", functionName);

}

if (noArguments < 0)

{

printf("%s: too many actual arguments\n", functionName);

}

emitJump("call", functionName);

return 1;

}

else if (strcmp("write", p->token.value.id) == 0) // write를 call했을 때

{

noArguments = 1;

emit0("ldp");

p = p->brother; // ACTUAL\_PARAM

while (p)

{

if (p->noderep == nonterm)

processOperator(p);

else {

stIndex = lookup(p->token.value.id);

if (stIndex == -1) break;;

emit2("lod", symbolTable[stIndex].base, symbolTable[stIndex].offset);

}

noArguments--;

p = p->brother;

}

if (noArguments > 0)

{

printf("%s: too few actual arguments\n", functionName);

}

if (noArguments < 0)

{

printf("%s: too many actual arguments\n", functionName);

}

emitJump("call", functionName);

return 1;

}

else if (strcmp(functionName, "lf") == 0) // lf를 call했을 때

{

emitJump("call", functionName);

return 1;

}

return 0;

}

// 연산자들을 처리하는 함수

void processOperator(NODE\* ptr)

{

switch (ptr->token.number) {

// assignment operator

case ASSIGN\_OP:

{

NODE\* lhs = ptr->son, \* rhs = ptr->son->brother;

int stIndex;

// step 1: generate instructions for left-hand side if INDEX node.

if (lhs->noderep == nonterm) { // array variable

lvalue = 1;

processOperator(lhs);

lvalue = 0;

}

// step 2: generate instructions for right-hand side

if (rhs->noderep == nonterm) processOperator(rhs);

else

\_emit(rhs, "rhs");

// step 3: generate a store instruction

if (lhs->noderep == terminal) { // simple variable

stIndex = lookup(lhs->token.value.id);

if (stIndex == -1) {

printf("undefined variable : %s\n", lhs->token.value.id);

return;

}

emit2("str", symbolTable[stIndex].base, symbolTable[stIndex].offset);

}

else

emit0("sti");

break;

}

// complex assignment operators

case ADD\_ASSIGN: case SUB\_ASSIGN: case MUL\_ASSIGN:

case DIV\_ASSIGN: case MOD\_ASSIGN:

{

NODE\* lhs = ptr->son, \* rhs = ptr->son->brother;

int nodeNumber = ptr->token.number;

int stIndex;

ptr->token.number = ASSIGN\_OP;

// step 1: code generation for left hand side

if (lhs->noderep == nonterm) {

lvalue = 1;

processOperator(lhs);

lvalue = 0;

}

ptr->token.number = nodeNumber;

// step 2: code generation for repeating part

if (lhs->noderep == nonterm)

processOperator(lhs);

else rv\_emit(lhs);

// step 3: code generation for right hand side

if (rhs->noderep == nonterm)

processOperator(rhs);

else rv\_emit(rhs);

// step 4: emit the corresponding operation code

switch (ptr->token.number) {

case ADD\_ASSIGN: emit0("add"); break;

case SUB\_ASSIGN: emit0("sub"); break;

case MUL\_ASSIGN: emit0("mult"); break;

case DIV\_ASSIGN: emit0("div"); break;

case MOD\_ASSIGN: emit0("mod"); break;

}

// step 5: code generation for store code

if (lhs->noderep == terminal) {

stIndex = lookup(lhs->token.value.id);

if (stIndex == -1) {

printf("undefined variable : %s\n", lhs->son->token.value.id);

return;

}

emit2("str", symbolTable[stIndex].base, symbolTable[stIndex].offset);

}

else

emit0("sti");

break;

}

// binary(arithmetic/relational/logical) operators

case ADD: case SUB: case MUL: case DIV: case MOD:

case EQ: case NE: case GT: case LT: case GE: case LE:

case LOGICAL\_AND: case LOGICAL\_OR:

{

NODE\* lhs = ptr->son, \* rhs = ptr->son->brother;

// step 1: visit left operand

if (lhs->noderep == nonterm) processOperator(lhs);

else rv\_emit(lhs);

// step 2: visit right operand

if (rhs->noderep == nonterm) processOperator(rhs);

else rv\_emit(rhs);

// step 3: visit root

switch (ptr->token.number) {

case ADD: emit0("add"); break; // arithmetic operators

case SUB: emit0("sub"); break;

case MUL: emit0("mult"); break;

case DIV: emit0("div"); break;

case MOD: emit0("mod"); break;

case EQ: emit0("eq"); break; // relational operators

case NE: emit0("ne"); break;

case GT: emit0("gt"); break;

case LT: emit0("lt"); break;

case GE: emit0("ge"); break;

case LE: emit0("le"); break;

case LOGICAL\_AND: emit0("and"); break; // logical operators

case LOGICAL\_OR: emit0("or"); break;

}

break;

}

// unary operators

case UNARY\_MINUS: case LOGICAL\_NOT:

{

NODE\* p = ptr->son;

if (p->noderep == nonterm) processOperator(p);

else rv\_emit(p);

switch (ptr->token.number) {

case UNARY\_MINUS: emit0("neg"); break;

case LOGICAL\_NOT: emit0("not"); break;

}

break;

}

// increment/decrement operators

case PRE\_INC: case PRE\_DEC: case POST\_INC: case POST\_DEC:

{

NODE\* p = ptr->son; NODE\* q;

int stIndex; // int amount = 1;

if (p->noderep == nonterm) processOperator(p); // compute operand

else rv\_emit(p);

q = p;

while (q->noderep != terminal) q = q->son;

if (!q || (q->token.number != tident)) {

printf("increment/decrement operators can not be applied in expression\n");

return;

}

stIndex = lookup(q->token.value.id);

if (stIndex == -1) return;

switch (ptr->token.number) {

case PRE\_INC:

emit0("inc");

// if(isOperation(ptr)) emit0(dup);

break;

case PRE\_DEC:

emit0("dec");

// if(isOperation(ptr)) emit0(dup);

break;

case POST\_INC:

// if(isOperation(ptr)) emit0(dup);

emit0("inc");

break;

case POST\_DEC:

// if(isOperation(ptr)) emit0(dup);

emit0("dec");

break;

}

if (p->noderep == terminal) {

stIndex = lookup(p->token.value.id);

if (stIndex == -1) return;

emit2("str", symbolTable[stIndex].base, symbolTable[stIndex].offset);

}

else if (p->token.number == INDEX) { // compute index

lvalue = 1;

processOperator(p);

lvalue = 0;

emit0("swp");

emit0("sti");

}

else printf("error in increment/decrement operators\n");

break;

}

case INDEX:

{

NODE\* indexExp = ptr->son->brother;

int stIndex;

if (indexExp->noderep == nonterm) processOperator(indexExp);

else rv\_emit(indexExp);

stIndex = lookup(ptr->son->token.value.id);

if (stIndex == -1) {

printf("undefined variable: %s\n", ptr->son->token.value.id);

return;

}

emit2("lda", symbolTable[stIndex].base, symbolTable[stIndex].offset);

emit0("add");

if (!lvalue) emit0("ldi"); // rvalue

break;

}

case CALL:

{

NODE\* p = ptr->son; // function name

char\* functionName;

int stIndex; int noArguments;

if (checkPredefined(p)) // predefined(Library) functions

break;

// handle for user function

functionName = p->token.value.id;

stIndex = lookup(functionName);

if (stIndex == -1) break; // undefined function !!!

noArguments = symbolTable[stIndex].width;

emit0("ldp");

p = p->brother; // ACTUAL\_PARAM

while (p) { // processing actual arguemtns

if (p->noderep == nonterm) processOperator(p);

else rv\_emit(p);

noArguments--;

p = p->brother;

}

if (noArguments > 0)

printf("%s: too few actual arguments", functionName);

if (noArguments < 0)

printf("%s: too many actual arguments", functionName);

emitJump("call", ptr->son->token.value.id);

break;

}

} // end switch

}

// nonterm과 terminal에 따라서 처리하는 함수

void processCondition(NODE\* ptr)

{

if (ptr->noderep == nonterm) processOperator(ptr);

else rv\_emit(ptr);

}

// 문장을 처리하는 함수

void processStatement(NODE\* ptr)

{

NODE\* p;

int returnWithValue;

switch (ptr->token.number) {

case COMPOUND\_ST:

p = ptr->son->brother; // STAT\_LIST

p = p->son;

while (p) {

processStatement(p);

p = p->brother;

}

break;

case EXP\_ST:

if (ptr->son != NULL) processOperator(ptr->son);

break;

case RETURN\_ST:

if (ptr->son != NULL) {

returnWithValue = 1;

p = ptr->son;

if (p->noderep == nonterm)

processOperator(p); // return value

else rv\_emit(p);

emit0("retv");

}

else

emit0("ret");

break;

case IF\_ST:

{

char label[LABEL\_SIZE];

genLabel(label);

processCondition(ptr->son); // condition part

emitJump("fjp", label);

processStatement(ptr->son->brother); // true part

emitLabel(label);

}

break;

case IF\_ELSE\_ST:

{

char label1[LABEL\_SIZE], label2[LABEL\_SIZE];

genLabel(label1); genLabel(label2);

processCondition(ptr->son); // condition part

emitJump("fjp", label1);

processStatement(ptr->son->brother); // true part

emitJump("ujp", label2);

emitLabel(label1);

processStatement(ptr->son->brother->brother); // false part

emitLabel(label2);

}

break;

case WHILE\_ST:

{

char label1[LABEL\_SIZE], label2[LABEL\_SIZE];

genLabel(label1); genLabel(label2);

emitLabel(label1);

processCondition(ptr->son); // condition part

emitJump("fjp", label2);

processStatement(ptr->son->brother); // loop body

emitJump("ujp", label1);

emitLabel(label2);

}

break;

default:

printf("not yet implemented.\n");

break;

} // end switch

}

// 파라미터인 단수 변수를 처리하는 함수

void processSimpleParamVariable(NODE\* ptr, int typeSpecifier, int typeQualifier)

{

NODE\* p = ptr->son; // variable name(=> identifier)

int stIndex, size;

if (ptr->token.number != SIMPLE\_VAR) printf("error in SIMPLE\_VAR\n");

size = typeSize(typeSpecifier);

stIndex = insert(p->token.value.id, typeSpecifier, typeQualifier, base, offset, 0, 0);

offset += size;

}

// 파라미터인 배열 변수를 처리하는 함수

void processArrayParamVariable(NODE\* ptr, int typeSpecifier, int typeQualifier)

{

NODE\* p = ptr->son; // variable name(=> identifier)

int stIndex, size;

if (ptr->token.number != ARRAY\_VAR) {

printf("error in ARRAY\_VAR\n");

return;

}

size = typeSize(typeSpecifier);

stIndex = insert(p->token.value.id, typeSpecifier, typeQualifier, base, offset, width, 0);

offset += size;

}

// 파라미터 선언을 처리하는 함수

void processParamDeclaration(NODE\* ptr)

{

int typeSpecifier, typeQualifier;

NODE\* p, \* q;

if (ptr->token.number != DCL\_SPEC) icg\_error(6);

// printf("processParamDeclaration\n");

// step 1: process DCL\_SPEC

typeSpecifier = INT\_TYPE; // default type

typeQualifier = VAR\_TYPE;

p = ptr->son;

while (p) {

if (p->token.number == INT\_NODE) typeSpecifier = INT\_TYPE;

else if (p->token.number == CONST\_NODE)

typeQualifier = CONST\_TYPE;

else { // AUTO, EXTERN, REGISTER, FLOAT, DOUBLE, SIGNED, UNSIGEND

printf("not yet implemented\n");

return;

}

p = p->brother;

}

// step 2: process SIMPLE\_VAR, ARRAY\_VAR

p = ptr->brother; // SIMPLE\_VAR or ARRAY\_VAR

switch (p->token.number) {

case SIMPLE\_VAR: // simple variable

processSimpleParamVariable(p, typeSpecifier, typeQualifier);

break;

case ARRAY\_VAR: // array variable

processArrayParamVariable(p, typeSpecifier, typeQualifier);

break;

default:

printf("error in SIMPLE\_VAR or ARRAY\_VAR\n");

break;

}

}

// 함수명을 처리하는 함수

void emitFunc(char\* FuncName, int operand1, int operand2, int operand3)

{

int label;

label = strlen(FuncName);

fprintf(ucodeFile, "%s", FuncName);

printf("%s", FuncName);

for (; label < LABEL\_SIZE - 1; label++) {

fprintf(ucodeFile, " ");

printf(" ");

}

fprintf(ucodeFile, "proc %d %d %d\n", operand1, operand2, operand3);

printf("proc %d %d %d\n", operand1, operand2, operand3);

}

// 함수헤더를 처리하는 함수

void processFuncHeader(NODE\* ptr)

{

int noArguments, returnType;

int stIndex;

NODE\* p;

// printf("processFuncHeader\n");

if (ptr->token.number != FUNC\_HEAD)

printf("error in processFuncHeader\n");

// step 1: process the function return type

p = ptr->son->son;

while (p) {

if (p->token.number == INT\_NODE) returnType = INT\_TYPE;

else if (p->token.number == VOID\_NODE) returnType = VOID\_TYPE;

else printf("invalid function return type\n");

p = p->brother;

}

// step 2: count the number of formal parameters

p = ptr->son->brother->brother; // FORMAL\_PARA

p = p->son; // PARAM\_DCL

noArguments = 0;

while (p) {

noArguments++;

p = p->brother;

}

// step 3: insert the function name

stIndex = insert(ptr->son->brother->token.value.id, returnType, FUNC\_TYPE,

1/\*base\*/, 0/\*offset\*/, noArguments/\*width\*/, 0/\*initialValue\*/);

// if(!strcmp("main", functionName)) mainExist = 1;

}

// 함수를 만드는 것을 처리하는 함수

void processFunction(NODE\* ptr)

{

NODE\* p, \* q;

int sizeOfVar = 0;

int numOfVar = 0;

int stIndex;

base++;

offset = 1;

if (ptr->token.number != FUNC\_DEF) icg\_error(4);

// step 1: process formal parameters

p = ptr->son->son->brother->brother;

p = p->son;

while (p) {

if (p->token.number == PARAM\_DCL) {

processParamDeclaration(p->son);

sizeOfVar++;

numOfVar++;

}

p = p->brother;

}

// step 2: process the declaration part in function body

p = ptr->son->brother->son->son;

while (p) {

if (p->token.number == DCL) {

processDeclaration(p->son);

q = p->son->brother;

while (q) {

if (q->token.number == DCL\_ITEM) {

if (q->son->token.number == ARRAY\_VAR) {

sizeOfVar += q->son->son->brother->token.value.num;

}

else sizeOfVar += 1;

numOfVar++;

}

q = q->brother;

}

}

p = p->brother;

}

// step 3: emit the function start code

p = ptr->son->son->brother; // IDENT

emitFunc(p->token.value.id, sizeOfVar, base, 2);

for (stIndex = symbolTableTop - numOfVar + 1; stIndex < symbolTableTop + 1; stIndex++)

emitSym(symbolTable[stIndex].base, symbolTable[stIndex].offset, symbolTable[stIndex].width);

// step 4: process the statement part in function body

p = ptr->son->brother; // COMPOUND\_ST

processStatement(p);

// step 5: check if return type and return value

p = ptr->son->son; // DCL\_SPEC

if (p->token.number == DCL\_SPEC) {

p = p->son;

if (p->token.number == VOID\_NODE) emit0("ret");

else if (p->token.number == CONST\_NODE) {

if (p->brother->token.number == VOID\_NODE) emit0("ret");

}

}

// step 6: generate the ending codes

emit0("end");

base--;

symbolTable->nextIndex++;

}

// 임시 코드를 생성하는 함수

void codeGen(NODE\* ptr)

{

NODE\* p;

int globalSize;

initSymbolTable();

// step 1: process the declaration part

for (p = ptr->son; p; p = p->brother) {

if (p->token.number == DCL) processDeclaration(p->son);

else if (p->token.number == FUNC\_DEF) processFuncHeader(p->son);

else icg\_error(3);

}

//dumpSymbolTable();

globalSize = offset - 1;

//printf("size of global variables = %d\n", globalSize);

genSym(base);

// step 2: process the function part

for (p = ptr->son; p; p = p->brother)

if (p->token.number == FUNC\_DEF) processFunction(p);

// if(!mainExist) warningmsg("main does not exist");

// step 3: generate code for starting routine

// bgn globalSize

// ldp

// call main

// end

emit1("bgn", globalSize);

emit0("ldp");

emitJump("call", "main");

emit0("end");

}

void main(int argc, char\* argv[])

{

char fileName[30] = "perfect.mc";

//char fileName[30] = "bubble.mc";

NODE\* root;

printf(" \*\*\* start of Mini C Compiler\n");

// 프롬프트(CMD)에서 실행할 시에 주석제거

//if (argc != 2) {

// icg\_error(1);

// exit(1);

//}

//strcpy(fileName, argv[1]);

//printf(" \* source file name: %s\n", fileName);

// 파일을 읽는데 없다면 종료

if ((source\_file = fopen(fileName, "r")) == NULL) {

icg\_error(2);

exit(1);

}

// AST와 U-code를 파일로 저장

astFile = fopen(strcat(strtok(fileName, "."), ".ast"), "w");

ucodeFile = fopen(strcat(strtok(fileName, "."), ".uco"), "w");

printf(" === start of Parser\n");

// 해당 파일을 파서가 파싱후 트리형태인 노드 반환

root = parser(source\_file);

// 트리형태를 출력

printTree(root, 1);

printf(" === start of ICG\n");

// 트리를 탐색하면서 코드생성

codeGen(root);

printf(" \*\*\* end of Mini C Compiler\n");

fclose(source\_file);

fclose(astFile);

fclose(ucodeFile);

}

**결과 출력)**

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

테이블이(가) 표시된 사진

자동 생성된 설명

프로그램을 실행한 후에 .ast와 .uco파일이 생성되었습니다. 그리고 이 U-code파일을 인터프리터에서 실행한다면

텍스트이(가) 표시된 사진

자동 생성된 설명

결과 화면과 같이 Mini-C로 작성한 완전수를 출력하는 프로그램이 정상적으로 실행되어 이것을 컴파일한 프로그램이 알맞은 U-code로 변환되는 것이 잘 작동되는 것을 알 수 있습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

테이블이(가) 표시된 사진

자동 생성된 설명

프로그램을 실행한 후에 .ast와 .uco파일이 생성되었습니다. 그리고 이 U-code파일을 인터프리터에서 실행한다면

텍스트이(가) 표시된 사진

자동 생성된 설명

결과 화면과 같이 Mini-C로 작성한 배열에 있는 수를 버블 소트하고 출력하는 프로그램이 정상적으로 실행되어 이것을 컴파일한 프로그램이 알맞은 U-code로 변환되는 것이 잘 작동되는 것을 알 수 있습니다.