1. code_generater

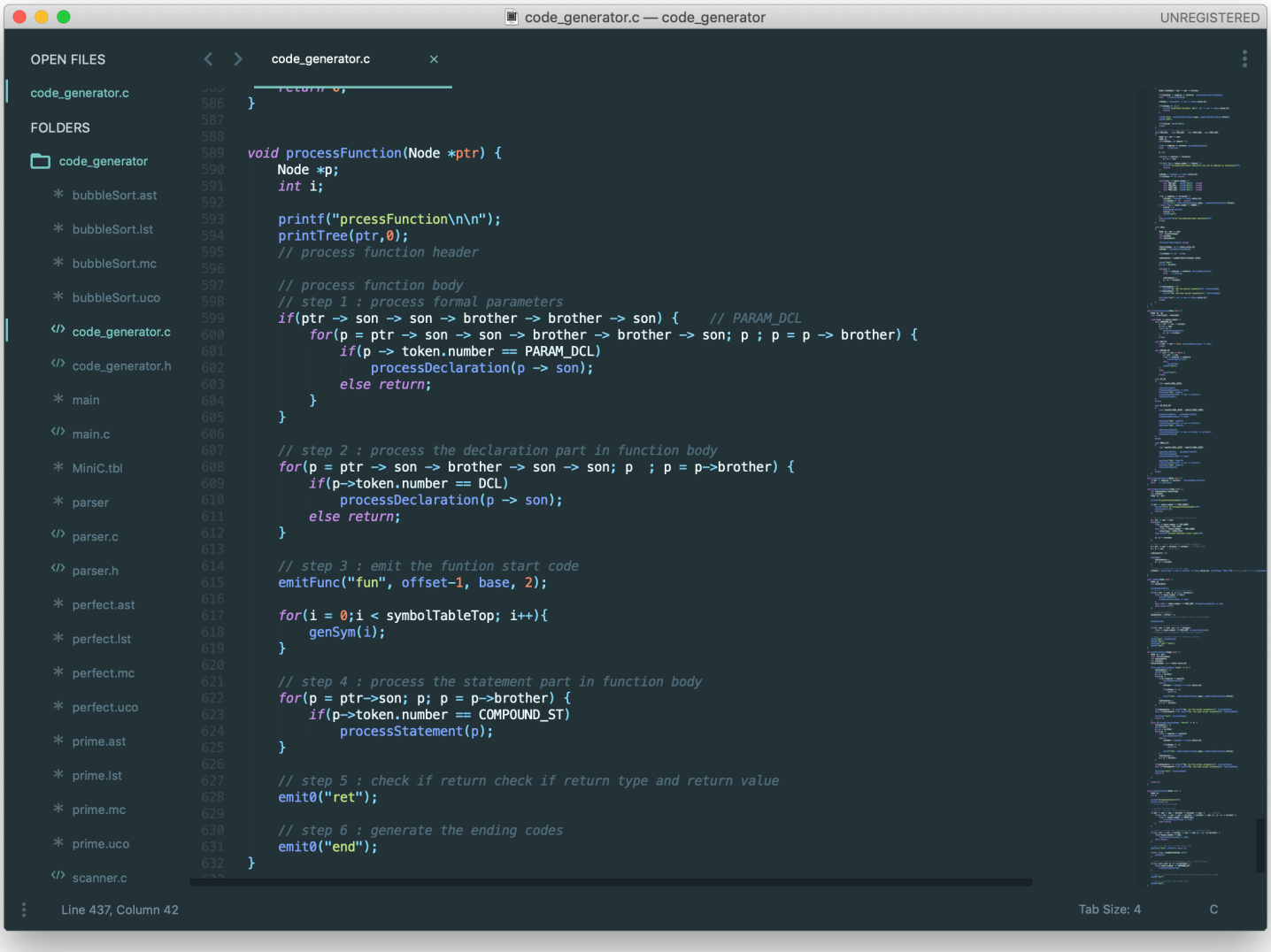

code_generator.c — code_generator
UNREGISTERED
OPEN FILES
code_generator.c
FOLDERS
code_generator
bubbleSort.ast
bubbleSort.lst
bubbleSort.mc
bubbleSort.uco
code_generator.c
code_generator.h
main
main.c
MiniC.tbl
parser
parser.c
parser.h
perfect.ast
perfect.lst
perfect.mc
perfect.uco
prime.ast
prime.lst
prime.mc
prime.uco
scanner.c
}
}
void processArrayVariable(Node *ptr, int typeSpecifier, int typeQualifier) {
Node *p = ptr -> son;
int stIndex,size;
if(ptr -> token.number != ARRAY_VAR) {
printf("error in ARRAY_VAR\n");
return;
}
if(p -> brother == NULL) {
printf("array size must be typeSpecifierified\n");
return;
}
else
size = p -> brother -> token.value.num;
stIndex = insert(p -> token.value.id, typeSpecifier, typeQualifier, base, offset, size, 0);
offset += size;
}
void processOperator(Node *ptr) {
int stIndex;
switch(ptr -> token.number) {
// assignment operator
case ASSIGN_OP:
{
Node *lhs = ptr -> son;
Node *rhs = ptr -> son -> brother;
if(lhs -> token.number == ASSIGN_OP && rhs -> token.number == ASSIGN_OP)
flag = 1;
// step 1 : generate instructions for left-hand side if INDEX node.
if(lhs -> noderep == nonterm) {
lvalue = 1;
processOperator(lhs);
lvalue = 0;
}
// step 2 : generate instructions for right-hand side
if(rhs -> noderep == nonterm) {
rvalue = 1;
processOperator(rhs);
rvalue = 0;
}
//step 3 : generate a store instruction
if(lhs -> noderep == terminal) {
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;
Node *rhs = ptr -> son -> brother;
int nodeNumber = ptr -> token.number;
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 repeatng 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);
8 lines, 141 characters selected
Tab Size: 4
C


code_generator.c — code_generator
UNREGISTERED
OPEN FILES
code_generator.c
FOLDERS
code_generator
bubbleSort.ast
bubbleSort.lst
bubbleSort.mc
bubbleSort.uco
code_generator.c
code_generator.h
main
main.c
MiniC.tbl
parser
parser.c
parser.h
perfect.ast
perfect.lst
perfect.mc
perfect.uco
prime.ast
prime.lst
prime.mc
prime.uco
scanner.c
// step 4 : emit the correspoinding 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) return;
emit2("str", symbolTable[stIndex].base , symbolTable[stIndex].offset);
}
else
emit0("sti");
break;
}
// binary(arithmetic / realational / logical) operations
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;
Node *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;
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;
case NE: emit0("ne"); break;
case GT: emit0("gt"); break;
case GE: emit0("ge"); break;
case LE: emit0("le"); break;
case LOGICAL_AND: emit0("and"); break;
case LOGICAL_OR: emit0("or"); break;
}
if(flag == 1){
emit0("dup");
flag = 0;
}
break;
}
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;
}
case INDEX:
{
Node *indexExp = ptr -> son -> brother;
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");
break;
}
8 lines, 141 characters selected
Tab Size: 4
C


FOLDERS
code_generator
bubbleSort.ast
bubbleSort.lst
bubbleSort.mc
bubbleSort.uco
code_generator.c
code_generator.h
main
main.c
MiniC.tbl
parser
parser.c
parser.h
perfect.ast
perfect.lst
perfect.mc
perfect.uco
prime.ast
prime.lst
prime.mc
prime.uco
code_generator
bubbleSort.ast
bubbleSort.lst
bubbleSort.mc
bubbleSort.uco
code_generator.c
code_generator.h
main
main.c
MiniC.tbl
parser
parser.c
parser.h
perfect.ast
perfect.lst
perfect.mc
perfect.uco
prime.ast
// increment / decrement operatiors
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);
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"); break;
case PRE_DEC: emit0("dec"); break;
case POST_INC: emit0("inc"); break;
case POST_DEC: 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) {
lvalue = 1;
processOperator(p);
lvalue = 0;
emit0("sti");
}
else printf("error increase/decrease operators\n");
break;
}
case CALL:
{
Node *p = ptr -> son;
char *functionName;
int stIndex;
int noArguments;
if(checkPredefined(p)) break;
functionName = p -> token.value.id;
stIndex = lookup(functionName);
if(stIndex == -1) break;
noArguments = symbolTable[stIndex].width;
emit0("ldp");
p = p -> brother;
while(p) {
if(p -> noderep == nonterm) processOperator(p);
else rv_emit(p);
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", ptr -> son -> token.value.id);
break;
}
}
}


code_generator.c — code_generator
UNREGISTERED
OPEN FILES
code_generator.c
FOLDERS
code_generator
bubbleSort.ast
bubbleSort.lst
bubbleSort.mc
bubbleSort.uco
code_generator.c
code_generator.h
main
main.c
MiniC.tbl
parser
parser.c
parser.h
perfect.ast
perfect.lst
perfect.mc
perfect.uco
prime.ast
prime.lst
void processStatement(Node *ptr) {
Node *p, *q;
char *startLabel, *endLabel;
switch(ptr -> token.number) {
case COMPOUND_ST:
p = ptr -> son -> brother;
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) {
p = ptr -> son;
if(p -> noderep == nonterm)
processOperator(p);
else
rv_emit(p);
emit0("retv");
}
else
emit0("ret");
break;
case IF_ST:
{
char label[LABEL_SIZE];
genLabel(label);
processCondition(ptr -> son);
emitJump("fjp", label);
processStatement(ptr -> son -> brother);
emitLabel(label);
}
break;
case IF_ELSE_ST:
{
char label1[LABEL_SIZE], label2[LABEL_SIZE];
genLabel(label1); genLabel(label2);
processCondition(ptr -> son);
emitJump("fjp", label1);
processStatement(ptr -> son -> brother);
emitJump("ujp", label2);
emitLabel(label1);
processStatement(ptr -> son -> brother -> brother);
emitLabel(label2);
}
break;
case WHILE_ST:
{
char label1[LABEL_SIZE], label2[LABEL_SIZE];
genLabel(label1); genLabel(label2);
emitLabel(label1);
processCondition(ptr -> son);
emitJump("fjp", label2);
processStatement(ptr -> son -> brother);
emitJump("ujp", label1);
emitLabel(label2);
}
break;
}
}
void processCondition(Node *ptr) {
if(ptr -> noderep == nonterm) processOperator(ptr);
else rv_emit(ptr);
}
void processFuncHeader(Node *ptr) {
int noArguments,returnType;
int stIndex;
Node *p = ptr;
printf("processFunctionHeader\n\n");
if(ptr -> token.number != FUNC_HEAD){
printf("error in processFunctionHeader!\n");
printTree(ptr,0);
exit(1);
}


```
    // 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*/,noArgument

}

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){
            printf("\n\n\n");
            processDeclaration(p -> son);
        }
        else if(p -> token.number == FUNC_DEF) processFuncHeader(p -> son);

    // 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 codes for starting routine
    emit1("bgn", globalSize);
    emit0("ldp");
    emitJump("call","main");
    emit0("end");
}

int checkPredefined(Node *ptr) {
    Node *p = ptr;
    char *functionName;
    int noArguments;
    int stIndex;
    functionName = p -> token.value.id;

    if(strcmp(functionName,"read") == 0) {
        noArguments = 1;
        emit0("ldp");
        p = p -> brother;
        while(p) {
            if(p->noderep == nonterm)
                processOperator(p);
            else {
                stIndex = lookup(p -> token.value.id);

                if(stIndex == -1)
                    return 0;

                emit2("lda", symbolTable[stIndex].base, symbolTable[stIndex].offset);
            }
            noArguments--;
            p = p -> brother;
        }

        if(noArguments > 0) printf("%s: too few actual arguments\n", functionName);
        else if(noArguments < 0) printf("%s: too many actual arguments\n", functionName);

        emitJump("call", functionName);
       return 1;

    }

    return 0;
}
```


code_generator.c — code_generator
UNREGISTERED
OPEN FILES
code_generator.c
FOLDERS
code_generator
bubbleSort.ast
bubbleSort.lst
bubbleSort.mc
bubbleSort.uco
code_generator.c
code_generator.h
main
main.c
MiniC.tbl
parser
parser.c
parser.h
perfect.ast
perfect.lst
perfect.mc
perfect.uco
prime.ast
prime.lst
prime.mc
prime.uco
scanner.c
code_generator.c
}
void processFunction(Node *ptr) {
Node *p;
int i;
printf("prcessFunction\n\n");
printTree(ptr,0);
// process function header
// process function body
// step 1 : process formal parameters
if(ptr -> son -> son -> brother -> brother -> son) { // PARAM_DCL
for(p = ptr -> son -> son -> brother -> brother -> son; p ; p = p -> brother) {
if(p -> token.number == PARAM_DCL)
processDeclaration(p -> son);
else return;
}
}
// step 2 : process the declaration part in function body
for(p = ptr -> son -> brother -> son -> son; p ; p = p->brother) {
if(p->token.number == DCL)
processDeclaration(p -> son);
else return;
}
// step 3 : emit the funtion start code
emitFunc("fun", offset-1, base, 2);
for(i = 0;i < symbolTableTop; i++){
genSym(i);
}
// step 4 : process the statement part in function body
for(p = ptr->son; p; p = p->brother) {
if(p->token.number == COMPOUND_ST)
processStatement(p);
}
// step 5 : check if return check if return type and return value
emit0("ret");
// step 6 : generate the ending codes
emit0("end");
}
Line 437, Column 42
Tab Size: 4
C

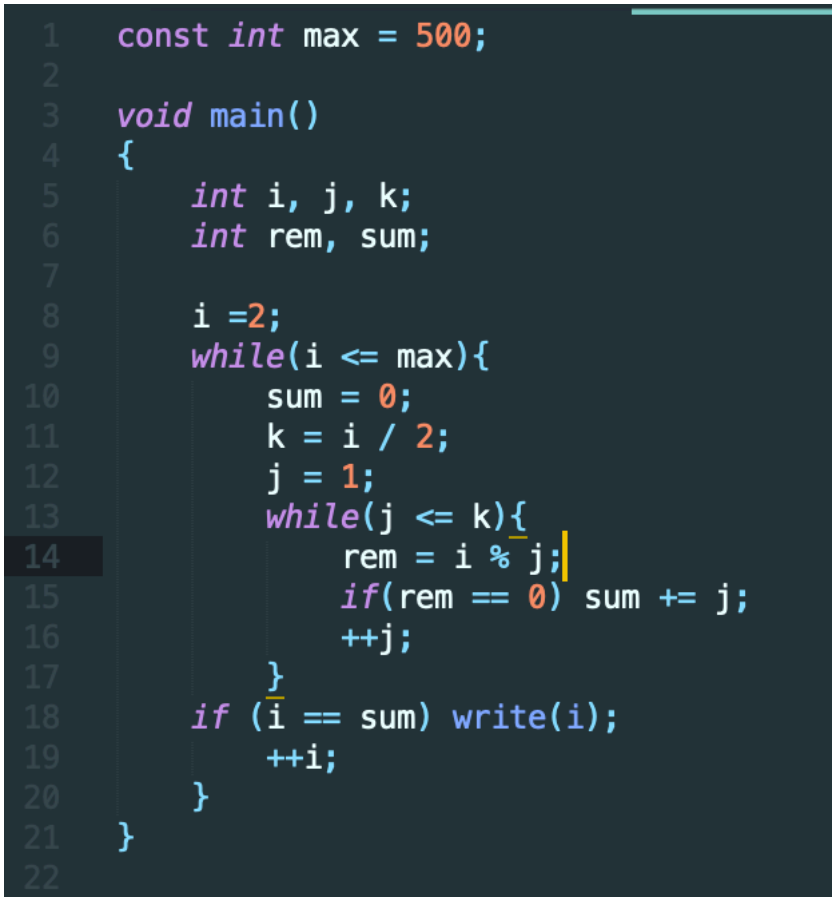
2. main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "parser.c"
#include "scanner.c"
#include "code_generator.c"

FILE *sourceFile;
FILE *astFile;
FILE *ucodeFile;

void main (int argc, char *argv[]){
    char fileName[30];
    Node *root;
    printf(" *** start of Mini C Compiler\n");
    if(argc != 2){
        icg_error(1);
        exit(1);
    }

    strcpy(fileName, argv[1]);

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

    if((sourceFile = fopen(fileName , "r")) == NULL){
        icg_error(2);
        exit(1);
    }
    astFile = fopen(strcat(strtok(fileName, "."),".ast"), "w");
    ucodeFile = fopen(strcat(strtok(fileName, "."),".uco"), "w");

    printf(" === start of Parser\n");
    root = parser();
    printf(" === start of ICG\n");
    codeGen(root);
    printf(" *** end of Mini C Compiler\n");
} //end of main
```

3. bubble.mc

```
void main ()
{
    int list[100];
    int element;
    int total, i, top;
    int temp;
    i = 1;
    read(element);

    while(element != 0){
        list[i] = element;
        ++i;
        read(element);
    }

    top = total = i - 1;

    while (top > 1) {
        i = 1;
        while (i < top) {
            if(list[i] > list[i + 1]) {
                temp = list[i];
                list[i] = list[i + 1];
                list[i + 1] = temp;
            }
            ++i;
        }
        top--;
    }

    i = 1;

    while(i <= total) {
        write(list[i]);
        ++i;
    }
}
```

4. perfect.mc

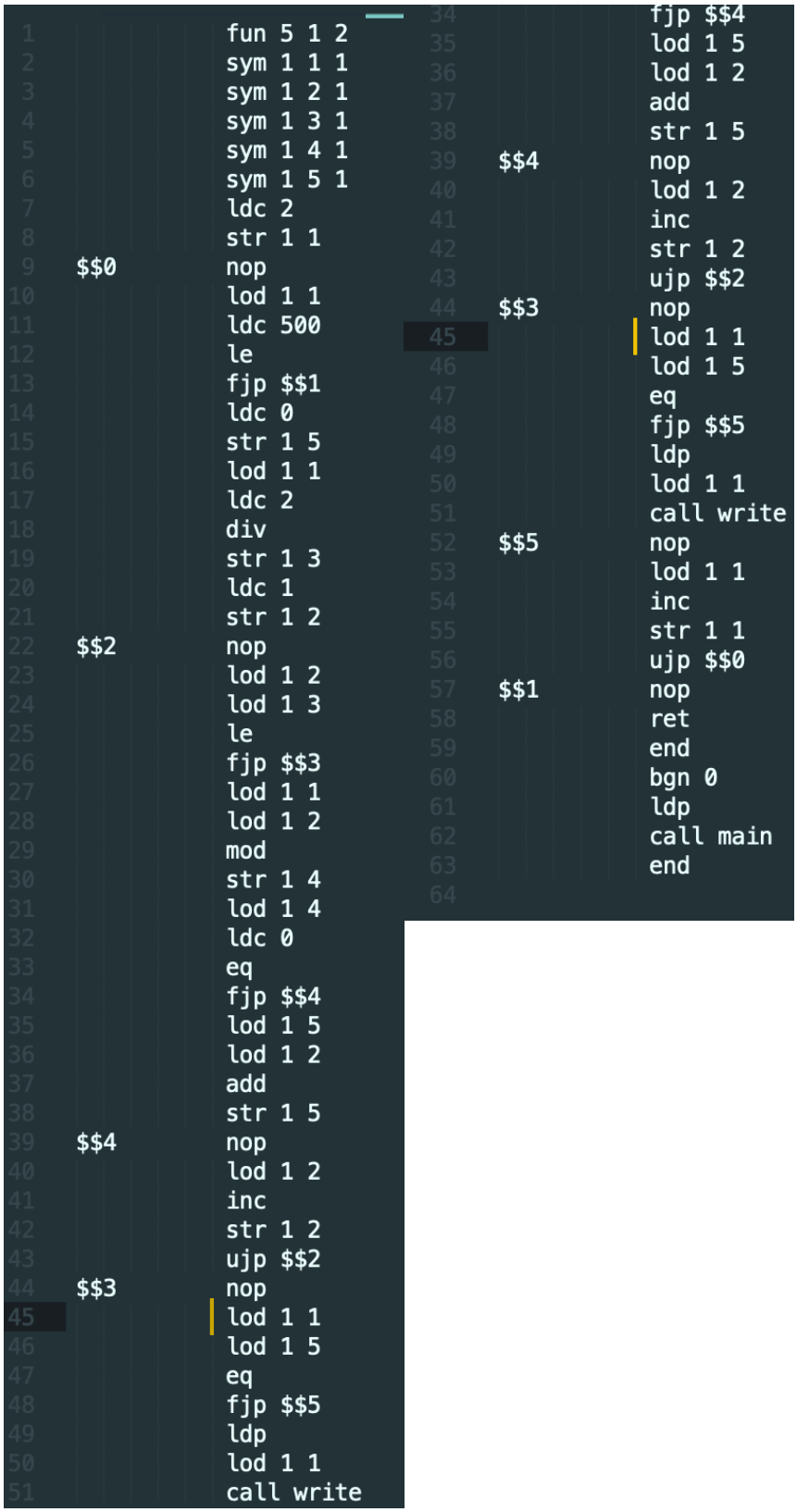


```
const int max = 500;

void main()
{
    int i, j, k;
    int rem, sum;

    i =2;
    while(i <= max){
        sum = 0;
        k = i / 2;
        j = 1;
        while(j <= k){
            rem = i % j;
            if(rem == 0) sum += j;
            ++j;
        }
    if (i == sum) write(i);
        ++i;
    }
}
```

5. bubbleSort.uco

```
        fun 105 1 2
        sym 1 1 100
        sym 1 101 1
        sym 1 102 1
        sym 1 103 1
        sym 1 104 1
        sym 1 105 1
        ldc 1
        str 1 103
        ldp
        lda 1 101
        call read
$$0     nop
        lod 1 101
        ldc 0
        ne
        fjp $$1
        lod 1 103
        lda 1 1
        add
        lod 1 101
        sti
        lod 1 103
        inc
        str 1 103
        ldp
        lda 1 101
        call read
        ujp $$0
$$1     nop
        lod 1 103
        ldc 1
        sub
        dup
        str 1 102
        str 1 104
$$2     nop
        lod 1 104
        ldc 1
        gt
        fjp $$3
        ldc 1
        str 1 103
$$4     nop
        lod 1 103
        lod 1 104
        lt
        fjp $$5
        lod 1 103
        lda 1 1
        add
        ldi
        lod 1 103
        ldc 1
        add
        lda 1 1
        add
        ldi
        gt
        fjp $$6
        lod 1 103
        lda 1 1
        add
        ldi
        str 1 105
        lod 1 103
        lda 1 1
        add
        lod 1 103
        ldc 1
        add
        lda 1 1
        add
        ldi
        sti
        lod 1 103
        ldc 1
        add
        lda 1 1
        add
        lod 1 105
        sti
$$6     nop
        lod 1 103
        inc
        str 1 103
        ujp $$4
$$5     nop
        lod 1 104
        dec
        str 1 104
        ujp $$2
$$3     nop
        ldc 1
        str 1 103
$$7     nop
        lod 1 103
        lod 1 102
        le
        fjp $$8
        ldp
        lod 1 103
        lda 1 1
        add
        ldi
        call write
        lod 1 103
        inc
        str 1 103
        ujp $$7
$$8     nop
        ret
        end
        bgn 0
        ldp
        call main
        end
```

6. perfect.uco

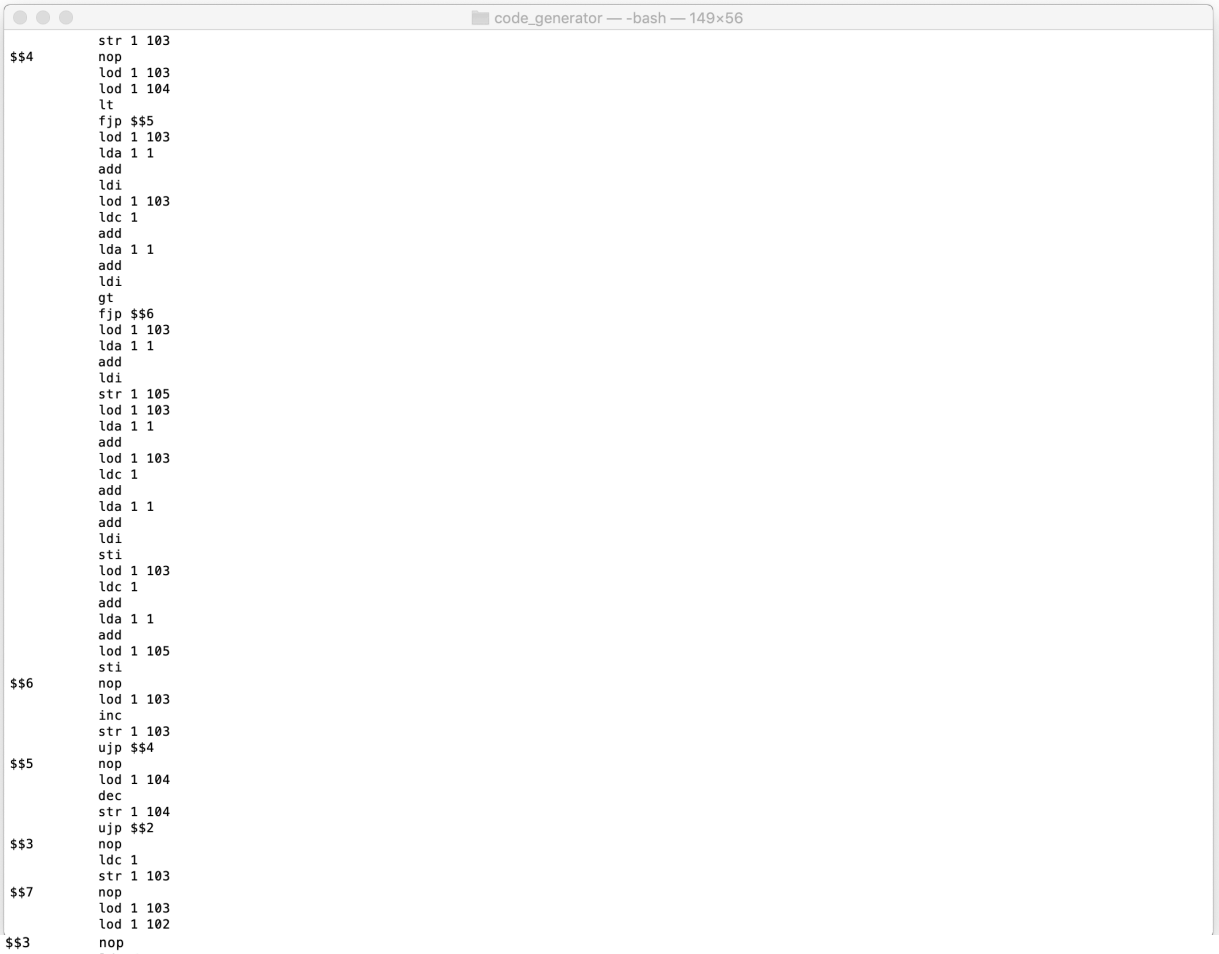


```
            fun 5 1 2
            sym 1 1 1
            sym 1 2 1
            sym 1 3 1
            sym 1 4 1
            sym 1 5 1
            ldc 2
            str 1 1
$$0         nop
            lod 1 1
            ldc 500
            le
            fjp $$1
            ldc 0
            str 1 5
            lod 1 1
            ldc 2
            div
            str 1 3
            ldc 1
            str 1 2
$$2         nop
            lod 1 2
            lod 1 3
            le
            fjp $$3
            lod 1 1
            lod 1 2
            mod
            str 1 4
            lod 1 4
            ldc 0
            eq
            fjp $$4
            lod 1 5
            lod 1 2
            add
            str 1 5
$$4         nop
            lod 1 2
            inc
            str 1 2
            ujp $$2
$$3         nop
            lod 1 1
            lod 1 5
            eq
            fjp $$5
            ldp
            lod 1 1
            call write
            fjp $$4
            lod 1 5
            lod 1 2
            add
            str 1 5
$$4         nop
            lod 1 2
            inc
            str 1 2
            ujp $$2
$$3         nop
            lod 1 1
            lod 1 5
            eq
            fjp $$5
            ldp
            lod 1 1
            call write
$$5         nop
            lod 1 1
            inc
            str 1 1
            ujp $$0
$$1         nop
            ret
            end
            bgn 0
            ldp
            call main
            end
```

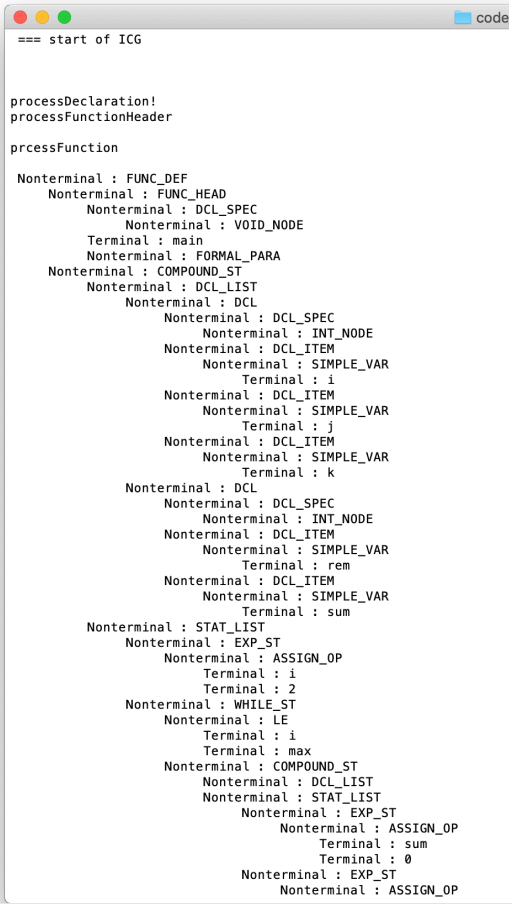
7. 실행결과

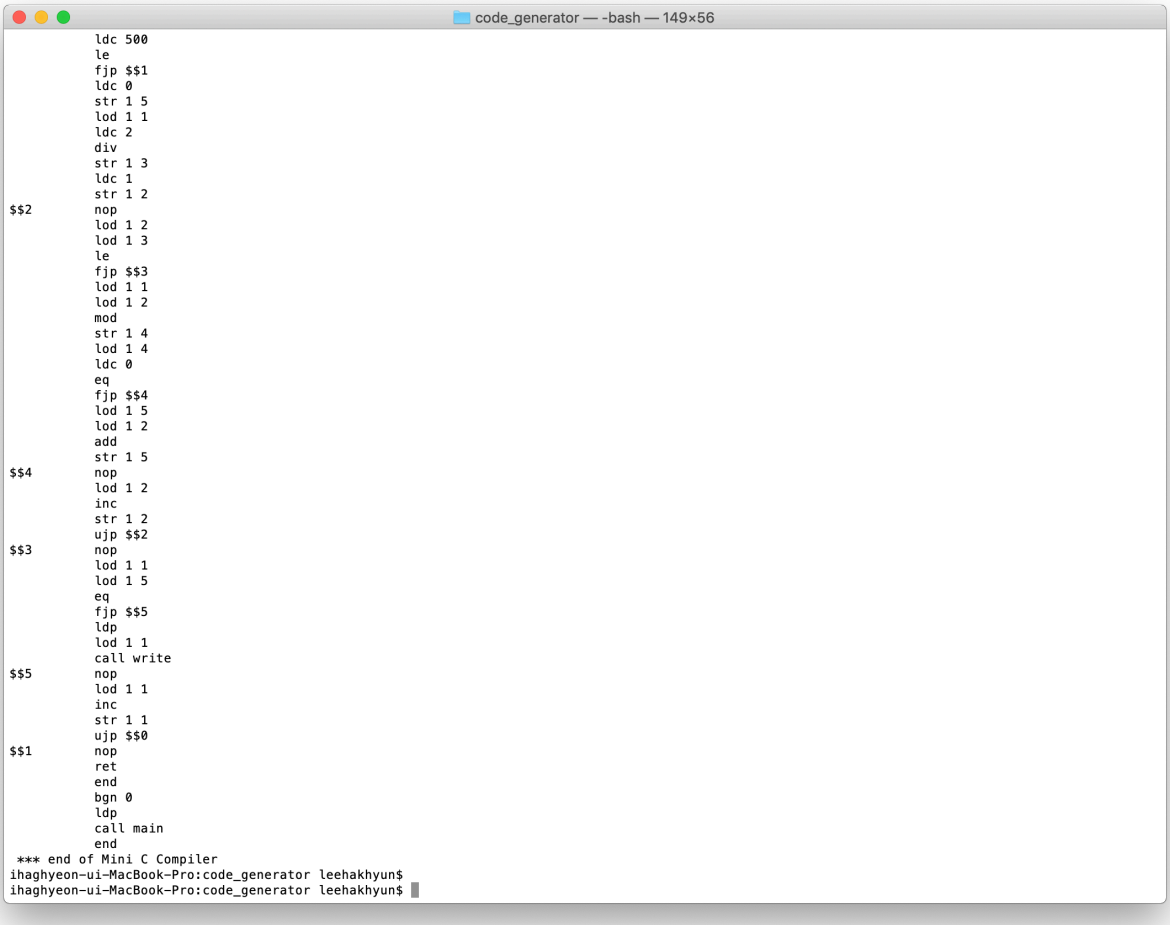
```
 === start of ICG
processFunctionHeader

prcessFunction

 Nonterminal : FUNC_DEF
     Nonterminal : FUNC_HEAD
          Nonterminal : DCL_SPEC
               Nonterminal : VOID_NODE
          Terminal : main
          Nonterminal : FORMAL_PARA
     Nonterminal : COMPOUND_ST
          Nonterminal : DCL_LIST
               Nonterminal : DCL
                    Nonterminal : DCL_SPEC
                         Nonterminal : INT_NODE
                    Nonterminal : DCL_ITEM
                         Nonterminal : ARRAY_VAR
                              Terminal : list
                              Terminal : 100
               Nonterminal : DCL
                    Nonterminal : DCL_SPEC
                         Nonterminal : INT_NODE
                    Nonterminal : DCL_ITEM
                         Nonterminal : SIMPLE_VAR
                              Terminal : element
               Nonterminal : DCL
                    Nonterminal : DCL_SPEC
                         Nonterminal : INT_NODE
                    Nonterminal : DCL_ITEM
                         Nonterminal : SIMPLE_VAR
                              Terminal : total
                    Nonterminal : DCL_ITEM
                         Nonterminal : SIMPLE_VAR
                              Terminal : i
                    Nonterminal : DCL_ITEM
                         Nonterminal : SIMPLE_VAR
                              Terminal : top
               Nonterminal : DCL
                    Nonterminal : DCL_SPEC
                         Nonterminal : INT_NODE
                    Nonterminal : DCL_ITEM
                         Nonterminal : SIMPLE_VAR
                              Terminal : temp
          Nonterminal : STAT_LIST
               Nonterminal : EXP_ST
                    Nonterminal : ASSIGN_OP
                         Terminal : i
                         Terminal : 1
               Nonterminal : EXP_ST
                    Nonterminal : CALL
                         Terminal : read
                         Terminal : element
               Nonterminal : WHILE_ST
                    Nonterminal : NE
                         Terminal : element
```

```
code_generator — -bash — 149×56
                         Terminal : 1
Nonterminal : WHILE_ST
     Nonterminal : GT
          Terminal : top
          Terminal : 1
     Nonterminal : COMPOUND_ST
          Nonterminal : DCL_LIST
          Nonterminal : STAT_LIST
               Nonterminal : EXP_ST
                    Nonterminal : ASSIGN_OP
                         Terminal : i
                         Terminal : 1
               Nonterminal : WHILE_ST
                    Nonterminal : LT
                         Terminal : i
                         Terminal : top
                    Nonterminal : COMPOUND_ST
                         Nonterminal : DCL_LIST
                         Nonterminal : STAT_LIST
                              Nonterminal : IF_ST
                                   Nonterminal : GT
                                        Nonterminal : INDEX
                                             Terminal : list
                                             Terminal : i
                                        Nonterminal : INDEX
                                             Terminal : list
                                             Nonterminal : ADD
                                                  Terminal : i
                                                  Terminal : 1
                                   Nonterminal : COMPOUND_ST
                                        Nonterminal : DCL_LIST
                                        Nonterminal : STAT_LIST
                                             Nonterminal : EXP_ST
                                                  Nonterminal : ASSIGN_OP
                                                       Terminal : temp
                                                       Nonterminal : INDEX
                                                            Terminal : list
                                                            Terminal : i
                                             Nonterminal : EXP_ST
                                                  Nonterminal : ASSIGN_OP
                                                       Nonterminal : INDEX
                                                            Terminal : list
                                                            Terminal : i
                                                       Nonterminal : INDEX
                                                            Terminal : list
                                                            Nonterminal : ADD
                                                                 Terminal : i
                                                                 Terminal : 1
                                             Nonterminal : EXP_ST
                                                  Nonterminal : ASSIGN_OP
                                                       Nonterminal : INDEX
                                                            Terminal : list
                                                            Nonterminal : ADD
                                                                 Terminal : i
                                                                  Terminal : 1
                                                                Terminal : temp
                                                           Nonterminal : EXP_ST
                                                                Nonterminal : PRE_INC
                                                                     Terminal : i
                                                      Nonterminal : EXP_ST
                                                           Nonterminal : POST_DEC
                                                                Terminal : top
                                        Nonterminal : EXP_ST
                                             Nonterminal : ASSIGN_OP
                                                  Terminal : i
                                                  Terminal : 1
                                        Nonterminal : WHILE_ST
                                             Nonterminal : LE
                                                  Terminal : i
                                                  Terminal : total
                                             Nonterminal : COMPOUND_ST
                                                  Nonterminal : DCL_LIST
                                                  Nonterminal : STAT_LIST
                                                       Nonterminal : EXP_ST
                                                            Nonterminal : CALL
                                                                 Terminal : write
                                                                 Nonterminal : INDEX
                                                                      Terminal : list
                                                                      Terminal : i
                                                       Nonterminal : EXP_ST
                                                            Nonterminal : PRE_INC
                                                                 Terminal : i
```




```
            str 1 103
$$4         nop
            lod 1 103
            lod 1 104
            lt
            fjp $$5
            lod 1 103
            lda 1 1
            add
            ldi
            lod 1 103
            ldc 1
            add
            lda 1 1
            add
            ldi
            gt
            fjp $$6
            lod 1 103
            lda 1 1
            add
            ldi
            str 1 105
            lod 1 103
            lda 1 1
            add
            lod 1 103
            ldc 1
            add
            lda 1 1
            add
            ldi
            sti
            lod 1 103
            ldc 1
            add
            lda 1 1
            add
            lod 1 105
            sti
$$6         nop
            lod 1 103
            inc
            str 1 103
            ujp $$4
$$5         nop
            lod 1 104
            dec
            str 1 104
            ujp $$2
$$3         nop
            ldc 1
            str 1 103
$$7         nop
            lod 1 103
            lod 1 102
$$3         nop
            ldc 1
            str 1 103
$$7         nop
            lod 1 103
            lod 1 102
            le
            fjp $$8
            ldp
            lod 1 103
            lda 1 1
            add
            ldi
            call write
            lod 1 103
            inc
            str 1 103
            ujp $$7
$$8         nop
            ret
            end
            bgn 0
            ldp
            call main
            end
 *** end of Mini C Compiler
```


code
=== start of ICG
processDeclaration!
processFunctionHeader
prcessFunction
Nonterminal : FUNC_DEF
Nonterminal : FUNC_HEAD
Nonterminal : DCL_SPEC
Nonterminal : VOID_NODE
Terminal : main
Nonterminal : FORMAL_PARA
Nonterminal : COMPOUND_ST
Nonterminal : DCL_LIST
Nonterminal : DCL
Nonterminal : DCL_SPEC
Nonterminal : INT_NODE
Nonterminal : DCL_ITEM
Nonterminal : SIMPLE_VAR
Terminal : i
Nonterminal : DCL_ITEM
Nonterminal : SIMPLE_VAR
Terminal : j
Nonterminal : DCL_ITEM
Nonterminal : SIMPLE_VAR
Terminal : k
Nonterminal : DCL
Nonterminal : DCL_SPEC
Nonterminal : INT_NODE
Nonterminal : DCL_ITEM
Nonterminal : SIMPLE_VAR
Terminal : rem
Nonterminal : DCL_ITEM
Nonterminal : SIMPLE_VAR
Terminal : sum
Nonterminal : STAT_LIST
Nonterminal : EXP_ST
Nonterminal : ASSIGN_OP
Terminal : i
Terminal : 2
Nonterminal : WHILE_ST
Nonterminal : LE
Terminal : i
Terminal : max
Nonterminal : COMPOUND_ST
Nonterminal : DCL_LIST
Nonterminal : STAT_LIST
Nonterminal : EXP_ST
Nonterminal : ASSIGN_OP
Terminal : sum
Terminal : 0
Nonterminal : EXP_ST
Nonterminal : ASSIGN_OP
Terminal : j
Terminal : k
Nonterminal : COMPOUND_ST
Nonterminal : DCL_LIST
Nonterminal : STAT_LIST
Nonterminal : EXP_ST
Nonterminal : ASSIGN_OP
Terminal : rem
Nonterminal : MOD
Terminal : i
Terminal : j
Nonterminal : IF_ST
Nonterminal : EQ
Terminal : rem
Terminal : 0
Nonterminal : EXP_ST
Nonterminal : ADD_ASSIGN
Terminal : sum
Terminal : j
Nonterminal : EXP_ST
Nonterminal : PRE_INC
Terminal : j
Nonterminal : IF_ST
Nonterminal : EQ
Terminal : i
Terminal : sum
Nonterminal : EXP_ST
Nonterminal : CALL
Terminal : write
Terminal : i
Nonterminal : EXP_ST
Nonterminal : PRE_INC
Terminal : i




```
Z:\Users\leehakhyun\Documents\Compiler\Compiler\code_generator>ucodei.exe bubbleSort.uco bubbleSort.lst
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
2 7 8 5 8 5 8 4 4 5 3 45 3  35 4 3 21 54 7 0
 2 3 3 3 4 4 4 5 5 5 7 7 8 8 8 21 35 45 54

Z:\Users\leehakhyun\Documents\Compiler\Compiler\code_generator>ucodei.exe perfect.uco perfect.lst
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
 6 28 496
```

윈도우 실행파일과 unix용 실행파일 모두 첨부하였습니다!