REPORT

“ 중간 코드 생성기 구현 ”

과 목 명	컴파일러 설계 및 구축 (월25,26)
담당교수	이양선 교수님
학 과	컴퓨터공학과
학 번	2015305084
이 름	홍송희
제 출 일	2018.12.13

<main.c 코드>

main.c

1212 — opcodeEnum

```c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include "parser.h"

#define SYMTAB_SIZE 200
#define LABEL_SIZE 12

int base = 1, offset = 1, width = 1;
int lvalue;
int symlevel = 0;

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

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
};
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"
};
typedef enum { NON_SPECIFIER, VOID_TYPE, INT_TYPE } TypeSpecifier;
typedef enum { NON_QUALIFIER, FUNC_TYPE, PARAM_TYPE, CONST_TYPE, VAR_TYPE} TypeQualifier;
typedef enum { ZERO_DIMENSION, ONE_DIMENSION } Dimension;
char *typeName[] = { "none",   "void",   "int" };
char *qualifierName[] = { "NONE", "FUNC_TYPE",  "PARAM_TYPE",  "CONST_TYPE", "VAR_TYPE" };
typedef struct tableType {
    char symbolName[ID_LENGTH];
    int typeSpecifier;
    int typeQualifier;
    int base;
    int offset;
    int width;                  // size
    int initialValue;           // initial value
    int nextIndex;              // link to next entry.
} SymbolEntry;
SymbolEntry symbolTable[SYMTAB_SIZE];       // symbol table
int symbolTableTop;

void rv_emit(Node* ptr);
void processOperator(Node* ptr);
void processArrayVariable(Node *ptr, int typeSpecifier, int typeQaulifier);
void processSimpleVariable(Node *ptr, int typeSpecifier, int typeQualifier);
void processSimpleParamVariable(Node *ptr, int typeSpecifier, int typeQualifier);
void processArrayParamVariable(Node *ptr, int typeSpecifier, int typeQualifier);
void processDeclaration(Node *ptr);
void processParamDeclaration(Node *ptr);
void processStatement(Node *ptr);
void codeGen(Node *ptr);
void icg_error(int err);
void emitLabel(char* label);
void genLabel(char* label);
void emitFunc(char *value, int p, int q, int r);
void emit0(int opcode);
void emit1(int opcode, int operand);
void emit2(int opcode, int operand1, int operand2);
void emitSym(int operand1, int operand2, int operand3);
void emitJump(int value, char* label);
void processCondition(Node* ptr);
void processFuncHeader(Node *ptr);
void processFunction(Node* ptr);
int typeSize(int typeSpecifier);
```

```
81  int checkPredefined(Node *ptr);
82  void initSymbolTable( );
83  int lookup(char *symbol);
84  int insert(char *symbol, int specifier, int qualifier, int base, int offset, int width, int initialValue);
85
86  void initSymbolTable( ) {
87      symbolTableTop = 0;
88  }
89  int lookup(char *symbol) {
90      int stIndex;
91      for (stIndex = 0; stIndex < symbolTableTop; stIndex++)
92          if ((strcmp(symbol, symbolTable[stIndex].symbolName) == 0) && (symbolTable[stIndex].nextIndex == symlevel))
93              return stIndex;
94      return -1;
95  }
96  int insert(char *symbol, int specifier, int qualifier, int base, int offset,
97      int width, int initialValue) {
98      strcpy(symbolTable[symbolTableTop].symbolName, symbol);
99      symbolTable[symbolTableTop].typeSpecifier = specifier;
100     symbolTable[symbolTableTop].typeQualifier = qualifier;
101     symbolTable[symbolTableTop].base = base;
102     symbolTable[symbolTableTop].offset = offset;
103     symbolTable[symbolTableTop].width = width;
104     symbolTable[symbolTableTop].initialValue = initialValue;
105     symbolTable[symbolTableTop].nextIndex = symlevel;
106     return ++symbolTableTop;
107 }
108
109 void genSym(int base) {
110     int stIndex;
111     //  fprintf(ucodeFile, "// Information for Symbol Table₩n");
112     for (stIndex = 0; stIndex <= symbolTableTop; stIndex++) {
113         if ((symbolTable[stIndex].typeQualifier == FUNC_TYPE) || (symbolTable[stIndex].typeQualifier == CONST_TYPE))
114             continue;
115         if (base == symbolTable[stIndex].base)
116             emitSym(symbolTable[stIndex].base, symbolTable[stIndex].offset, symbolTable[stIndex].width);
117     }
118 }
```

```
119 void codeGen(Node *ptr) {
120     Node *p;
121     int globalSize;
122     initSymbolTable( );
123     //1.process the declaration part
124     for (p = ptr->son; p; p = p->brother) {
125         if (p->token.number == DCL) processDeclaration(p->son);
126         else if (p->token.number == FUNC_DEF) processFuncHeader(p->son);
127         else icg_error(3);
128     }
129     //dumpSymbolTable( );//
130     globalSize = offset - 1;
131     //printf("size of global variables = %d₩n", globalSize);
132     genSym(base);
133     //2.process the function part
134     for (p = ptr->son; p; p = p->brother)
135         if (p->token.number == FUNC_DEF) processFunction(p);
136     //if (!mainExist) warningmsg("main does not exist");
137     //3.generate codes for starting routine
138     // bgn  globalSize
139     // ldp
140     // call main
141     // end
142     emit1(bgn, globalSize);
143     emit0(ldp);
144     emitJump(call, "main");
145     emit0(endop);
146 }
147 void processDeclaration(Node *ptr) {
148     int typeSpecifier, typeQualifier;
149     Node *p, *q;
150     if (ptr->token.number != DCL_SPEC) icg_error(4);
151     //printf("processDeclaration₩n");
152     //1. process DCL_SPEC
153     typeSpecifier = INT_TYPE;
154     typeQualifier = VAR_TYPE;
155     p = ptr->son;
```

```
156         while (p) {
157             if (p->token.number == INT_NODE) typeSpecifier = INT_TYPE;
158             else if (p->token.number == CONST_NODE)
159                 typeQualifier = CONST_TYPE;
160             else { //AUTO, EXTERN, REGISTER, FLOAT, DOUBLE, SIGNED, UNSIGNED
161                 printf("not yet implemented₩n");
162                 return;
163             }
164             p = p->brother;
165         }
166         //2. process DCL_ITEM
167         p = ptr->brother;
168         if (p->token.number != DCL_ITEM) icg_error(5);
169         while (p) {
170             q = p->son; //SIMPLE_VAR OR ARRAY_VAR
171             switch (q->token.number) {
172             case SIMPLE_VAR:
173                 processSimpleVariable(q, typeSpecifier, typeQualifier);
174                 break;
175             case ARRAY_VAR:
176                 processArrayVariable(q, typeSpecifier, typeQualifier);
177                 break;
178             default:
179                 printf("error in SIMPLE_VAR or ARRAY_VAR₩n");
180                 break;
181             }
182             p = p->brother;
183         }
184     }
185     void processSimpleVariable(Node *ptr, int typeSpecifier, int typeQualifier) {
186         Node *p = ptr->son;         //variable name(=> indentifier)
187         Node *q = ptr->brother;     //initial value part
188         int stIndex, size;
189         int initialValue;
190         int sign = 1;
191
192         if (ptr->token.number != SIMPLE_VAR) printf("error in SIMPLE_VAR₩n");
193         if (typeQualifier == CONST_TYPE) {
194             if (q == NULL) {
195                 printf("%s must have a constant value₩n", ptr->son->token.value.id);
196                 return;
197             }
198             if (q->token.number == UNARY_MINUS) {
199                 sign = -1;
200                 q = q->son;
201             }
202             initialValue = sign * q->token.value.num;
203             stIndex = insert(p->token.value.id, typeSpecifier, typeQualifier,
204                 0/*base*/, 0/*offset*/, 0/*width*/, initialValue);
205         }
206         else {
207             size = typeSize(typeSpecifier);
208             stIndex = insert(p->token.value.id, typeSpecifier, typeQualifier,
209                 base, offset, width, 0);
210             offset += size;
211         }
212     }
213     void processArrayVariable(Node *ptr, int typeSpecifier, int typeQaulifier) {
214         Node *p = ptr->son;
215         int stIndex, size;
216
217         if (ptr->token.number != ARRAY_VAR) {
218             printf("error in ARRAY_VAR₩n");
219             return;
220         }
221         if (p->brother == NULL)
222             printf("array size must be specified₩n");
223         else size = p->brother->token.value.num;
224         size *= typeSize(typeSpecifier);
225         stIndex = insert(p->token.value.id, typeSpecifier, typeQaulifier,
226             base, offset, size, 0);
227         offset += size;
228     }
229     void processSimpleParamVariable(Node *ptr, int typeSpecifier, int typeQualifier) {
230         Node *p = ptr->son;
231         int stIndex, size;
232
```

```
233         if (ptr->token.number != SIMPLE_VAR) printf("error in SIMPLE_VAR₩n");
234         size = typeSize(typeSpecifier);
235         stIndex = insert(p->token.value.id, typeSpecifier, typeQualifier,
236             base, offset, 0, 0);
237         offset += size;
238     }
239 void processArrayParamVariable(Node *ptr, int typeSpecifier, int typeQualifier) {
240         Node *p = ptr->son; // variable name(=> identifier)
241         int stIndex, size;
242 
243         if (ptr->token.number != ARRAY_VAR) { printf("error in ARRAY_VAR₩n"); return; }
244         size = typeSize(typeSpecifier);
245         stIndex = insert(p->token.value.id, typeSpecifier, typeQualifier,
246             base, offset, width, 0);
247         offset += size;
248     }
249 void processParamDeclaration(Node *ptr) {
250         int typeSpecifier, typeQualifier;
251         Node *p, *q;
252         
253         if (ptr->token.number != DCL_SPEC) icg_error(4);
254         typeSpecifier = INT_TYPE;
255         typeQualifier = VAR_TYPE;
256         p = ptr->son;
257         while (p) {
258             if (p->token.number == INT_NODE) typeSpecifier = INT_TYPE;
259             else if (p->token.number == CONST_NODE) typeQualifier = CONST_TYPE;
260             else {  printf("not yet implemented₩n"); return;  }
261             p = p->brother;
262         }
263         p = ptr->brother;
264         switch (p->token.number) {
265         case SIMPLE_VAR:
266             processSimpleParamVariable(p, typeSpecifier, typeQualifier);
267             break;
268         case ARRAY_VAR:
269             processArrayParamVariable(p, typeSpecifier, typeQualifier);
270             break;
271         default:
272             printf("error in SIMPLE_VAR or ARRAY_VAR₩n");
273             break;
274         }
275     }
276 void processOperator(Node* ptr) {
277     switch (ptr->token.number) { // assignment operator
278     case ASSIGN_OP: {
279         Node* lhs = ptr->son, *rhs = ptr->son->brother;
280         int stIndex;
281         //1. generate instructions for left-hand side if INDEX node
282         if (lhs->noderep == nonterm) {
283             lvalue = 1;
284             processOperator(lhs);
285             lvalue = 0;
286         }
287         //2.generate instructions for right-hand side
288         if (rhs->noderep == nonterm) processOperator(rhs);
289         else rv_emit(rhs);
290         //3.generate a store instruction
291         if (lhs->noderep == terminal) {
292             stIndex = lookup(lhs->token.value.id);
293             if (stIndex == -1) {
294                 printf("undefined varriable : %s₩n", lhs->token.value.id);
295                 return;
296             }
297             emit2(str, symbolTable[stIndex].base, symbolTable[stIndex].offset);
298         }
299         else emit0(sti);
300         break;
301     }
302     case ADD_ASSIGN: case SUB_ASSIGN: case MUL_ASSIGN: case DIV_ASSIGN: case MOD_ASSIGN:
303     {   //complex assignment operators
304         Node* lhs = ptr->son;
305         Node* rhs = ptr->son->brother;
306         int nodeNumber = ptr->token.number;
307         int stIndex;
308         ptr->token.number = ASSIGN_OP;
309         //1. code generation for left hand side
310         if (lhs->noderep == nonterm) {
311             lvalue = 1;
312             processOperator(lhs);
313             lvalue = 0;
314         }
```

```
315             ptr->token.number = nodeNumber;
316             //2. code generation for repeating part
317             if (lhs->noderep == nonterm)
318                 processOperator(lhs);
319             else rv_emit(lhs);
320             //3. code generation for right hand side
321             if (rhs->noderep == nonterm) processOperator(rhs);
322             else rv_emit(rhs);
323             //4. emit the corresponding operation code
324             switch (ptr->token.number) {
325             case ADD_ASSIGN: emitO(add); break;
326             case SUB_ASSIGN: emitO(sub); break;
327             case MUL_ASSIGN: emitO(mult); break;
328             case DIV_ASSIGN: emitO(divop); break;
329             case MOD_ASSIGN: emitO(modop); break;
330             }
331             //5. code generation for stor code
332             if (lhs->noderep == terminal) {
333                 stIndex = lookup(lhs->token.value.id);
334                 if (stIndex == -1) {
335                     printf("undefined variable : %s₩n", lhs->son->token.value.id);
336                     return;
337                 }
338                 emit2(str, symbolTable[stIndex].base, symbolTable[stIndex].offset);
339             }
340             else emitO(sti);
341             break;
342         }
343         case ADD: case SUB: case MUL: case DIV: case MOD:
344         case EQ:  case NE:  case GT:  case LT:  case GE:  case LE:
345         case LOGICAL_AND:   case LOGICAL_OR:
346         {   //binary(arithmetic/relational/logical) operators
347             Node *lhs = ptr->son, *rhs = ptr->son->brother;
348             //1. visit left operand
349             if (lhs->noderep == nonterm) processOperator(lhs);
350             else rv_emit(lhs);
351             //2.visit right operand
352             if (rhs->noderep == nonterm) processOperator(rhs);
353             else rv_emit(rhs);
```

```
354             //3. visit root
355             switch (ptr->token.number) {
356             case ADD: emitO(add); break;
357             case SUB: emitO(sub); break;
358             case MUL: emitO(mult); break;
359             case DIV: emitO(divop); break;
360             case MOD: emitO(modop); break;
361             case EQ: emitO(eq); break;
362             case NE: emitO(ne); break;
363             case GT: emitO(gt); break;
364             case LT: emitO(lt); break;
365             case GE: emitO(ge); break;
366             case LE: emitO(le); break;
367             case LOGICAL_AND:   emitO(andop); break;
368             case LOGICAL_OR: emitO(orop); break;
369             }
370             break;
371         }
372         case UNARY_MINUS: case LOGICAL_NOT:
373         {   //unary operators
374             Node* p = ptr->son;
375             if (p->noderep == nonterm) processOperator(p);
376             else rv_emit(p);
377             switch (ptr->token.number) {
378             case UNARY_MINUS: emitO(neg); break;
379             case LOGICAL_NOT: emitO(notop); break;
380             }
381             break;
382         }
383         case PRE_INC: case PRE_DEC: case POST_INC: case POST_DEC:
384         {   //increment/decrement operators
385             Node *p = ptr->son; Node *q;
386             int stIndex; //int amount = 1;
387             if (p->noderep == nonterm) processOperator(p);
388             else rv_emit(p);
389             q = p;
390             while (q->noderep != terminal) q = q->son;
391             if (!q || (q->token.number != tident)) {
392                 printf("increment/decrement operators can not be applied in expression₩n");
393                 return;
394             }
```

```
395             stIndex = lookup(q->token.value.id);
396             if (stIndex == -1) return;
397             switch (ptr->token.number) {
398             case PRE_INC:  emitO(incop);
399                 // if(isOperation(ptr)) emitO(dup);
400                 break;
401             case PRE_DEC:  emitO(decop);
402                 // if(isOperation(ptr)) emitO(dup);
403                 break;
404             case POST_INC:  emitO(incop);
405                 // if(isOperation(ptr)) emitO(dup);
406                 break;
407             case POST_DEC: emitO(decop);
408                 // if(isOperation(ptr)) emitO(dup);
409                 break;
410             }
411             if (p->noderep == terminal) {
412                 stIndex = lookup(p->token.value.id);
413                 if (stIndex == -1) return;
414                 emit2(str, symbolTable[stIndex].base, symbolTable[stIndex].offset);
415             }
416             else if (p->token.number == INDEX) {
417                 lvalue = 1;
418                 processOperator(p);
419                 lvalue = 0;
420                 emitO(swp);
421                 emitO(sti);
422             }
423             else printf("error in increment/decrement operators\n");
424             break;
425         }
426         case INDEX:
427         {
428             Node* indexExp = ptr->son->brother;
429             int stIndex;
430             if (indexExp->noderep == nonterm) processOperator(indexExp);
431             else rv_emit(indexExp);
432             stIndex = lookup(ptr->son->token.value.id);
433             if (stIndex == -1) {
434                 printf("undefined variable : %s\n", ptr->son->token.value.id);
435                 return;
436             }
437             emit2(lda, symbolTable[stIndex].base, symbolTable[stIndex].offset);
438             emitO(add);
439             if (!lvalue) emitO(ldi);
440             break;
441         }
442         case CALL:
443         {
444             Node* p = ptr->son;
445             char *functionName;
446             int stIndex;  int noArguments;
447             if (checkPredefined(p)) break;
448             //handle for user func
449             functionName = p->token.value.id;
450             stIndex = lookup(functionName);
451             if (stIndex == -1) break;
452             noArguments = symbolTable[stIndex].width;
453             emitO(ldp);
454             p = p->brother;
455             while (p) {
456                 if (p->noderep == nonterm) processOperator(p);
457                 else rv_emit(p);
458                 noArguments--;
459                 p = p->brother;
460             }
461             if (noArguments > 0) printf("%s: too few actual arguments", functionName);
462             if (noArguments < 0) printf("%s: too many actual arguments", functionName);
463             emitJump(call, ptr->son->token.value.id);
464             break;
465         }
466         }
467     }
468     void processStatement(Node *ptr) {
469         Node *p = NULL;
470         int returnWithValue;
471         switch (ptr->token.number) {
472         case COMPOUND_ST:
473             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) {
            returnWithValue = 1;
            p = ptr->son;
            if (p->noderep == nonterm) processOperator(p);
            else rv_emit(p);
            emitO(retv);
        }
        else  emitO(retv);
        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;
    default:
        printf("not yet implemented.\n");
        break;
    }
}
void processCondition(Node* ptr) {
    if (ptr->noderep == nonterm) processOperator(ptr);
    else rv_emit(ptr);
}
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;
    }
```

```
553     // step 2: process the declaration part in function body
554     p = ptr->son->brother->son->son;
555     while (p) {
556         if (p->token.number == DCL) {
557             processDeclaration(p->son);
558             q = p->son->brother;
559             while (q) {
560                 if (q->token.number == DCL_ITEM) {
561                     if (q->son->token.number == ARRAY_VAR) {
562                         sizeOfVar += q->son->son->brother->token.value.num;
563                     }
564                     else sizeOfVar += 1;
565                     numOfVar++;
566                 }
567                 q = q->brother;
568             }
569         }
570         p = p->brother;
571     }
572     // step 3: emit the function start code
573     p = ptr->son->son->brother; // IDENT
574     emitFunc(p->token.value.id, sizeOfVar, base, 2);
575     for (stIndex = symbolTableTop - numOfVar; stIndex < symbolTableTop; stIndex++)
576         emitSym(symbolTable[stIndex].base, symbolTable[stIndex].offset, symbolTable[stIndex].width);
577     // step 4: process the statement part in function body
578     p = ptr->son->brother;  // COMPOUND_ST
579     processStatement(p);
580     // step 5: check if return type and return value
581     p = ptr->son->son;  // DCL_SPEC
582     if (p->token.number == DCL_SPEC) {
583         p = p->son;
584         if (p->token.number == VOID_NODE)   emit0(ret);
585         else if (p->token.number == CONST_NODE) {
586             if (p->brother->token.number == VOID_NODE)  emit0(ret);
587         }
588     }
589     // step 6: generate the ending codes
590     emit0(endop);
591     base--;
592     symbolTable->nextIndex++;
593 }
```

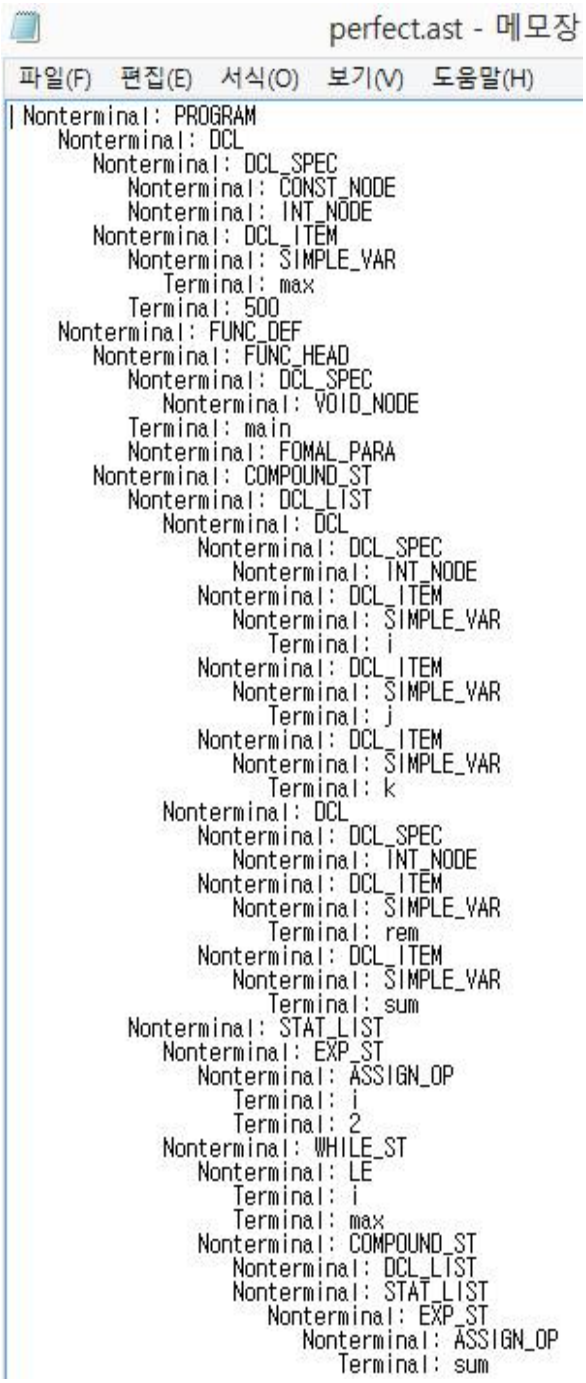
```
594 void processFuncHeader(Node *ptr) {
595     int noArguments, returnType;
596     int stIndex;
597     Node* p;
598
599     //printf("processFuncHeader₩n");
600     if (ptr->token.number != FUNC_HEAD)
601         printf("error in processFuncHeader₩n");
602     //1.process the function return type
603     p = ptr->son->son;
604     while (p) {
605         if (p->token.number == INT_NODE) returnType = INT_TYPE;
606         else if (p->token.number == VOID_NODE) returnType = VOID_TYPE;
607         else printf("invalid function return type₩n");
608         p = p->brother;
609     }
610     //2. count the number of formal parameters
611     p = ptr->son->brother->brother;
612     p = p->son;
613     noArguments = 0;
614     while (p) {
615         noArguments++;
616         p = p->brother;
617     }
618     //3.insert the function name
619     stIndex = insert(ptr->son->brother->token.value.id, returnType, FUNC_TYPE,
620         1/*base*/, 0/*width*/, noArguments/*width*/, 0/*initialValue*/);
621     //if(!strcmp("main",functionName)) mainExist=1;
622 }
623 void rv_emit(Node* ptr) {
624     int stIndex;
625
626     if (ptr->token.number == tnumber)
627         emit1(ldc, ptr->token.value.num);
628     else {
629         stIndex = lookup(ptr->token.value.id);
630         if (stIndex == -1) return;
631         if (symbolTable[stIndex].typeQualifier == CONST_TYPE)
632             emit1(ldc, symbolTable[stIndex].initialValue);
```

```
633             else if (symbolTable[stIndex].width > 1)
634                 emit2(lda, symbolTable[stIndex].base, symbolTable[stIndex].offset);
635             else
636                 emit2(lod, symbolTable[stIndex].base, symbolTable[stIndex].offset);
637         }
638     }
639     void emitSym(int operand1, int operand2, int operand3) {
640         fprintf(ucodeFile, "           sym  %d  %d  %d\n", operand1, operand2, operand3);
641         printf("           sym  %d  %d  %d\n", operand1, operand2, operand3);
642     }
643     void emitJump(int opcode, char *label) {
644         fprintf(ucodeFile, "           %s  %s\n", opcodeName[opcode], label);
645         printf("           %s  %s\n", opcodeName[opcode], label);
646     }
647     void emit0(int opcode) {
648         fprintf(ucodeFile, "           %s\n", opcodeName[opcode]);
649         printf("           %s\n", opcodeName[opcode]);
650     }
651     void emit1(int opcode, int operand) {
652         fprintf(ucodeFile, "           %s  %d\n", opcodeName[opcode], operand);
653         printf("           %s  %d\n", opcodeName[opcode], operand);
654     }
655     void emit2(int opcode, int operand1, int operand2) {
656         fprintf(ucodeFile, "           %s  %d  %d\n", opcodeName[opcode], operand1, operand2);
657         printf("           %s  %d  %d\n", opcodeName[opcode], operand1, operand2);
658     }
659     void emitFunc(char *value, int p, int q, int r) {
660         int label;
661         label = strlen(value);
662         fprintf(ucodeFile, "%s", value);
663         printf("%s", value);
664         for (; label < LABEL_SIZE - 1; label++) {
665             fprintf(ucodeFile, " ");
666             printf(" ");
667         }
668         fprintf(ucodeFile, "fun  %d  %d  %d\n", p, q, r);
669         printf("fun  %d  %d  %d\n", p, q, r);
670     }
```

```
671     void genLabel(char* label) {
672         static int labelNum = 0;
673         sprintf(label, "$$%d", labelNum++);
674     }
675     void emitLabel(char* label) {
676         int length;
677         length = strlen(label);
678         fprintf(ucodeFile, "%s", label);
679         printf("%s", label);
680         for (; length < LABEL_SIZE + 1; length++) {
681             fprintf(ucodeFile, " ");
682             printf(" ");
683         }
684         fprintf(ucodeFile, "nop\n");
685         printf("nop\n");
686     }
687     void icg_error(int err) {
688         printf("error %d\n", err);
689     }
690     int checkPredefined(Node *ptr)
691     {
692         Node *p = NULL;
693         if (strcmp(ptr->token.value.id, "read") == 0) {
694             emit0(ldp);
695             p = ptr->brother;
696             while (p) {
697                 if (p->noderep == nonterm) processOperator(p);
698                 else rv_emit(p);
699                 p = p->brother;
700             }
701             emitJump(call, "read");
702             return 1;
703         }
704         else if (strcmp(ptr->token.value.id, "write") == 0) {
705             emit0(ldp);
706             p = ptr->brother;
707             while (p) {
708                 if (p->noderep == nonterm) processOperator(p);
709                 else rv_emit(p);
710                 p = p->brother;
711             }
```

```
712             emitJump(call, "write");
713             return 1;
714         }
715         else if (strcmp(ptr->token.value.id, "lf") == 0) {
716             emitJump(call, "lf");
717             return 1;
718         }
719         return 0;
720     }
721     int typeSize(int typeSpecifier)
722     {
723         if (typeSpecifier == INT_TYPE)
724             return 1;
725         else {
726             printf("not yet implemented₩n");
727             return 1;
728         }
729     }
730     int main(int argc, char* argv[]) {
731         char fileName[30];
732         Node *root;
733
734         printf(" *** start of Mini C Compiler₩n");
735         if (argc != 2) {
736             icg_error(1);
737             exit(1);
738         }
739         strcpy(fileName, argv[1]);
740         printf(" *source file name: %s₩n", fileName);
741         freopen(fileName, "r", stdin);
742         if ((sourceFile = fopen(fileName, "r")) == NULL) {
743             icg_error(2);
744             exit(1);
745         }
746         astFile = fopen(strcat(strtok(fileName, "."), ".ast"), "w");
747         ucodeFile = fopen(strcat(strtok(fileName, "."), ".uco"), "w");
748
749         printf(" === start of Parser₩n");
750         root = parser();
751         printTree(root, 0);
752         printf(" === start of ICG₩n");
753         codeGen(root);
754         printf(" *** end   of Mini C Compiler₩n");
755
756         fclose(sourceFile);
757         fclose(astFile);
758         fclose(ucodeFile);
759         return 0;
760     }
761
```

<실행 결과 - perfect.ast>



perfect.ast - 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
Nonterminal: PROGRAM
   Nonterminal: DCL
      Nonterminal: DCL_SPEC
         Nonterminal: CONST_NODE
         Nonterminal: INT_NODE
      Nonterminal: DCL_ITEM
         Nonterminal: SIMPLE_VAR
            Terminal: max
         Terminal: 500
   Nonterminal: FUNC_DEF
      Nonterminal: FUNC_HEAD
         Nonterminal: DCL_SPEC
            Nonterminal: VOID_NODE
         Terminal: main
         Nonterminal: FOMAL_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
```

```
      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: k
                  Nonterminal: DIV
                     Terminal: i
                     Terminal: 2
            Nonterminal: EXP_ST
               Nonterminal: ASSIGN_OP
                  Terminal: j
                  Terminal: 1
            Nonterminal: WHILE_ST
               Nonterminal: LE
                  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
```

<실행 결과 - perfect.ast, 결과CMD창>

perfect.uco

파일(F) 편집(E) 서식(O) 보기(V) 도움

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

```
C:\Users\송희\Desktop\1212\Debug>1212.exe perfect.mc
 *** start of Mini C Compiler
 *source file name: perfect.mc
 === start of Parser
 === start of ICG
main    fun  5  2  2
        sym  2  1  1
        sym  2  2  1
        sym  2  3  1
        sym  2  4  1
        sym  2  5  1
        ldc  2
        str  2  1
$$0     nop
        lod  2  1
        ldc  500
        le
        fjp  $$1
        ldc  0
        str  2  5
        lod  2  1
        ldc  2
        div
        str  2  3
        ldc  1
        str  2  2
$$2     nop
        lod  2  2
        lod  2  3
        le
        fjp  $$3
        lod  2  1
        lod  2  2
        mod
        str  2  4
        lod  2  4
        ldc  0
        eq
        fjp  $$4
        lod  2  5
        lod  2  2
        add
        str  2  5
$$4     nop
        lod  2  2
        inc
        str  2  2
        ujp  $$2
$$3     nop
        lod  2  1
        lod  2  5
        eq
        fjp  $$5
        ldp
        lod  2  1
        call  write
$$5     nop
        lod  2  1
        inc
        str  2  1
        ujp  $$0
$$1     nop
        ret
        end
        bgn  0
        ldp
        call  main
        end
 *** end   of Mini C Compiler
```

```
C:₩Users₩송희₩Desktop₩1212₩Debug>ucodei perfect.uco perfect.lst
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
6 28 496
```

<실행 결과 - perfect.lst>

perfect.lst

```
   line      object          ucode  source program

    1   (35    5    2)     main       fun  5  2  2
    2   (38    2    1)                sym  2  1  1
    3   (38    2    2)                sym  2  2  1
    4   (38    2    3)                sym  2  3  1
    5   (38    2    4)                sym  2  4  1
    6   (38    2    5)                sym  2  5  1
    7   (20    2    )                 ldc  2
    8   (24    2    1)                str  2  1
    9   (34         )     $$0        nop
   10   (19    2    1)                lod  2  1
   11   (20  500    )                 ldc  500
   12   (16         )                 le
   13   (28   57    )                 fjp  $$1
   14   (20    0    )                 ldc  0
   15   (24    2    5)                str  2  5
   16   (19    2    1)                lod  2  1
   17   (20    2    )                 ldc  2
   18   ( 9         )                 div
   19   (24    2    3)                str  2  3
   20   (20    1    )                 ldc  1
   21   (24    2    2)                str  2  2
   22   (34         )     $$2        nop
   23   (19    2    2)                lod  2  2
   24   (19    2    3)                lod  2  3
   25   (16         )                 le
   26   (28   44    )                 fjp  $$3
   27   (19    2    1)                lod  2  1
   28   (19    2    2)                lod  2  2
   29   (10         )                 mod
   30   (24    2    4)                str  2  4
   31   (19    2    4)                lod  2  4
   32   (20    0    )                 ldc  0
   33   (17         )                 eq
   34   (28   39    )                 fjp  $$4
   35   (19    2    5)                lod  2  5
   36   (19    2    2)                lod  2  2
   37   ( 6         )                 add
   38   (24    2    5)                str  2  5
   39   (34         )     $$4        nop
   40   (19    2    2)                lod  2  2
   41   ( 2         )                 inc
   42   (24    2    2)                str  2  2
   43   (26   22    )                 ujp  $$2
   44   (34         )     $$3        nop
   45   (19    2    1)                lod  2  1
   46   (19    2    5)                lod  2  5
   47   (17         )                 eq
   48   (28   52    )                 fjp  $$5
   49   (23         )                 ldp
   50   (19    2    1)                lod  2  1
   51   (29   -2    )                 call  write
   52   (34         )     $$5        nop
   53   (19    2    1)                lod  2  1
   54   ( 2         )                 inc
   55   (24    2    1)                str  2  1
   56   (26    9    )                 ujp  $$0
   57   (34         )     $$1        nop
   58   (30         )                 ret
   59   (36         )                 end
   60   (37    0    )                 bgn  0
   61   (23         )                 ldp
   62   (29    1    )                 call  main
   63   (36         )                 end

  ****    Result    ****

6 28 496
```

```
            ##### Statistics #####

    ****  Static Instruction Counts  ****

inc    =  2       add    =  1       div    =  1       mod    =  1
le     =  2       eq     =  2       lod    =  14      ldc    =  6
ldp    =  2       str    =  8       ujp    =  2       fjp    =  4
call   =  2       ret    =  1       nop    =  6       fun    =  1
end    =  2       bgn    =  1       sym    =  5

    ****  Dynamic instruction counts  ****

inc    =  62999     add    =  2690      div    =  499       mod    =  62500
le     =  63499     eq     =  62999     lod    =  383877    ldc    =  64498
ldp    =  4         str    =  129687    ujp    =  62999     fjp    =  126498
call   =  4         ret    =  1         nop    =  126998    fun    =  1
end    =  1         bgn    =  1         sym    =  5

 Executable instruction count  =   1022755

 Total execution cycle         =   13705259
```