숭실대학교 Soongsil University

시스템프로그래밍 2021 보고서

보고서 제출서약서

나는 숭실대학교 컴퓨터학부의 일원으로 명예를 지키면서 생활하고 있습니다.

나는 보고서를 작성하면서 다음과 같은 사항을 준수하였음을 엄숙히 서약합니다.

1. 나는 자력으로 보고서를 작성하였습니다.
   1.1. 나는 동료의 보고서를 베끼지 않았습니다.
   1.2. 나는 비공식적으로 얻은 해답/해설을 기초로 보고서를 작성하지 않았습니다.
2. 나는 보고서에서 참조한 문헌의 출처를 밝혔으며 표절하지 않았습니다. (나는 특히 인터넷에서 다운로드한 내용을 보고서에 거의 그대로 복사하여 사용하지 않았습니다.)
3. 나는 보고서를 제출하기 전에 동료에게 보여주지 않았습니다.
4. 나는 보고서의 내용을 조작하거나 날조하지 않았습니다.

과목	시스템프로그래밍 2021 <가반>
과제명	어셈블러 구현 (Program Project #1b)
담당교수	최 재 영 교 수
제출인	컴퓨터학부 20180637 홍길동 (출석번호 101번)
제출일	2021년 05월 17일

차 례

1장 프로젝트 동기/목적

control section 방식의 어셈블리 코드를 object code로 이루어진 object program으로 바꾸는 assembler를 제작한다. 프로젝트 1과 다르게 JAVA로 구현한다.

2장 설계/구현 아이디어

assembler 프로그램은 총 6개의 class로 구성되어있다.

1. Assembler.class : 프로그램 main 루틴을 실행하는 class
2. Instruction.class : assembly 기계어 명령어의 구체적 정보를 담은 class
3. InstTable.class : assembly 기계어 명령의 정보를 관리하는 class
4. LabelTable.class : symbol table, label table을 이루는 class
5. Token.class : assembly 프로그램의 라인을 token 단위로 분할해 저장하는 class
6. TokenTable.class : assembly 프로그램의 라인을 token 단위로 분할하고 object code로 변환을 총괄하는 class

프로그램 main 루틴은 다음과 같다.

```
Assembler assembler = new Assembler("inst.data");
assembler.loadInputFile("input.txt");
assembler.pass1();

assembler.printSymbolTable("symtab_20180637");
assembler.printLiteralTable("literaltab_20180637");
assembler.pass2();
assembler.printObjectCode("output_20180637");
```

프로그램 main 루틴이 하는 일은 Project 1과 같다. main 루틴의 함수는 Project 1과 이번 프로젝트에서의 차이가 있는 부분 위주로 설명하겠다.

1) `public Assembler(String instFile)`

프로그램에 필요한 각종 클래스 객체를 초기화 및 생성한다. InstTable class의 객체 instTable은 생성자로 맴버 변수 세팅까지 한다. lineList는 어셈블리 프로그램의 명령을 한 줄씩 저장하는 ArrayList<String> class의 객체이다.

프로그램 section별 symbol table과 literal table을 저장하는 ArrayList<LableTable> class 객체를 생성한다. 프로그램 section별 TokenTable을 저장하는 ArrayList<TokenTable> class 객체를 생성한다. object code를 object 프로그램에 사용되는 최종 출력형태로 저장하는 ArrayList<String> class 객체 codeList를 생성한다.

주어진 ArrayList 이외에 한 어셈블리 프로그램 길이를 저장하는 ArrayList<Integer> progLength 객체생성을 하였다.

2) `private void loadInputFile(String inputFile)`

java.io.BufferedReader class를 사용해 inputfile을 읽어 한 줄씩 lineList에 저장한다.

3) `private void pass1()`

pass1에서 token parsing을 하는데, 이 parsing은 TokenTable class에서 이루어지고 TokenTable에 저장된다. parsing 이후 location 계산이 이뤄진다. symTable과 literalTable은 section별로 나뉘어 저장된다. 처음 parsing 하기 전 LabelTable 객체로 생성하고 label은 symTable에, '='으로 시작하는 literal은 literalTable에 저장한다. Project 1에서는 table이 각각 한 개여서 section별 구분을 짓은 index를 여러개 사용하느라 가독성이

떨어지는 경향이 있었지만 java의 ArrayList collection을 활용해 section간 구분을 간단하게, 가독성좋게 구현하였다.
"LTORG" operator와 "END"operator를 만날 때 literalTable의 address가 정의된다. LTORG와 END뒤에 원래 literal이 정의된 명령이 있는 것처럼 tokenTable에 label이 "*", operator가 literal인 명령어를 parsing후 저장한다. 원래 lineList에 없던 명령이기 때문에 tokenNum을 1늘린다.
"CSECT" operator를 만나면 새로운 프로그램을 만들어야 하므로 이전 프로그램의 정보를 List에 저장한다. 저장하는 정보는 progLength, symbolTable, literalTable, tokenTable이다. 그리고 새로운 프로그램을 위해 위 정보를 가지는 class 객체를 다시 생성한다. 이 정보의 저장은 "END" operator를 만날 때도 일어난다.

4)**private void** printSymbolTable(String fileName)
java.io.PrintWriter class를 사용해 fileName에 symtable 의 label과 address를 순서대로 매칭해 출력한다.

5)**private void** printLiteralTable(String fileName)
java.io.PrintWriter class를 사용해 fileName에 literaltable 의 label과 address를 순서대로 매칭해 출력한다.

6)**private void** pass2()
object code를 만들어 codeList에 저장하는 함수이다. object code가 존재하지 않는 RESW와 같은 operator는 object code를 생성하지 않는다. 예외로 object code가 없지만 bject program 제작시 라인변경을 하지 않는 LTORG와 END operator는 objectcode를 생성한다.(단 결과는 빈칸이다.)

7)**private void** printObjectCode(String fileName)
codeList와 TokenTable로 object program을 만든다. H,D,R Record는 section 시작마다 출력한다. T Record는 record 첫 문자가 'T'가 아니면 T Record로 초기화하는 작업을 한다. startAddress는 Token class에 저장되어 있지만 record의 길이는 알수 없으므로 "XX"로 두고 나중에 길이가 정해지면 변경한다.
record를 출력하는 기준은 record의 길이가 70 이상이거나, null objectcode(null codeList)를 만났을 때, section이 끝날 때이다. record의 길이를 XX 자리에 넣고 출력한 뒤 record를 'X'로 바꿔 T Record 초기화 작업을 자연스럽게 수행하도록 한다.
M Record는 tokenTable에 저장해둔 것을 가져와 출력한다.

Instruction class는 기계어 명령, opcode, operand 개수, format의 맴버변수를 가진다. class의 생성자는 parsing 메서드를 호출한다. parsing 메서드는 파라미터인 String line을 ','를 기준으로 나눠서 순서대로 맴버변수에 저장한다. 즉 instruction data file엔 각 정보가 ','를 기준으로 저장돼있다.

InstTable class는 HashMap<String, Instruction> 타입의 instMap 맴버변수를 가진다. String타입의 기계어 명령 이름을 key로하여 그 명령어의 정보를 가진 Instruction class 객체를 쉽게 찾을 수 있다. Instruction 객체는 생성자가 호출하는 openFile 메서드에서 이뤄진다. openFile 메서드는 기계어 명령 정보가 들어있는 파라미터인 fileName의 파일을 java.io.BufferedReader class로 입력받아 입력파일을 라인별로 Instruction class 객체를 만들고 instMap에 순서대로 입력한다.
InstMap에 파라미터로 주어진 명령어가 있는지 search하는 containInst와 instMap에서 파라미터인 명령어에 대응하는 Instruction 객체를 얻는 get 메서드를 구현하였다.

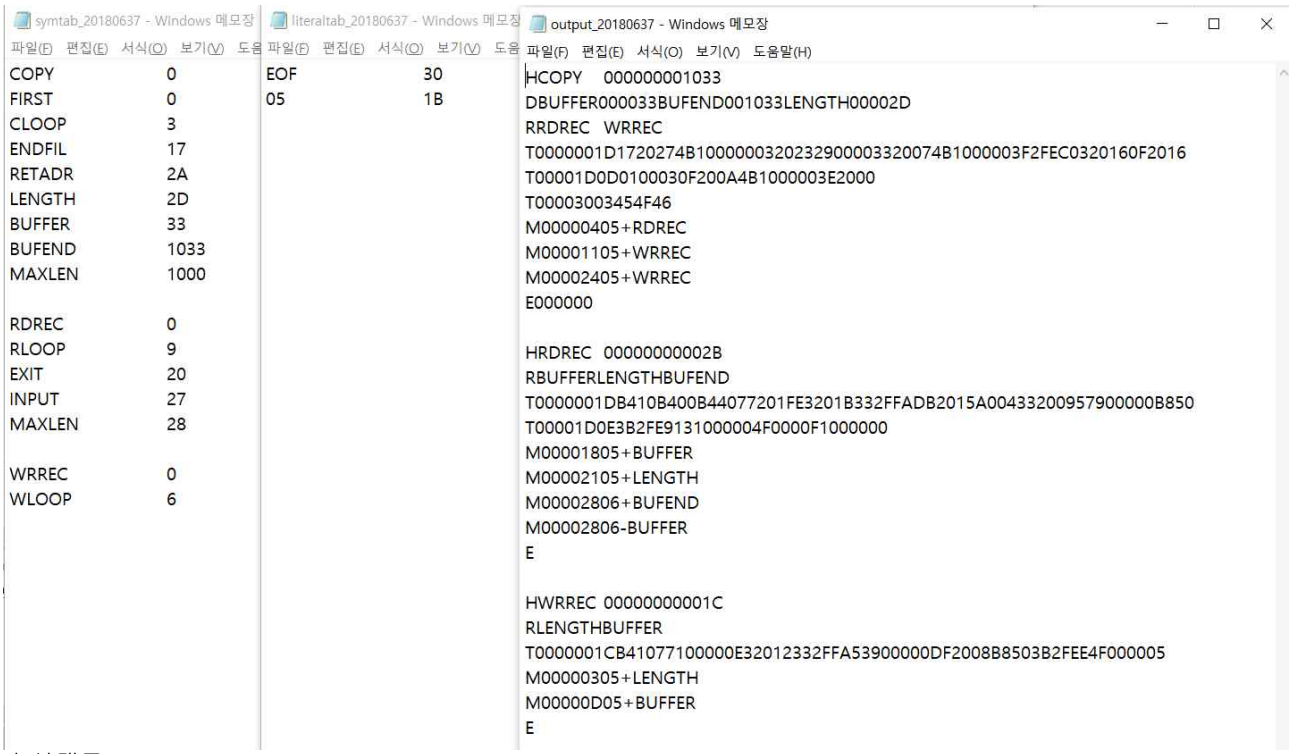
LabelTable class는 맴버함수로 4개의 ArraryList를 갖는다. 각각 label의 이름, label의 주소, extdef label 이름, extref label 이름을 저장한다. putName 메서드는 파라미터로 label과 주소를 입력받아 맴버함수에 저장한다. symbol table과 label table은 같은 label이 두 번 이상 입력되면 안된다. 따라서 중복된 label의 입력은

putName 메서드가 호출되면 안되고, modifyName 메서드에서 주소값을 변경하는 것으로 한다.
파라미터로 주어지는 label이 table에 존재하면 address를 리턴하는 search 메서드로 putName을 사용할지, modufyName을 사용할지 확일할수 있다. LabelTable 생성자를 추가로 구현해 맴버함수 ArraryList 객체를 모두 생성한다.

Token class는 맴버변수로 assembly 명령의 주소, label, operator, operand, nixbpe, objectcode, objectcode의 bytesize를 가진다. 생성자는 parsing 메서드를 호출해 assembly 명령에서부터 각 맴버변수가 가지는 정보 단위로 parsing하고 저장한다.
어셈블리 명령은 '₩t'를 기준으로 구분되므로 파라미터 line을 '₩t'단위로 분할한다. 각 명령은 "lable₩toperator₩toperand1,2,3₩tcomment"로 되어있으므로 분할한 String 배열을 각각의 맴버변수에 저장한다. 어셈블리 주석의 경우 operator와 operand가 없다.
operand가 두 개 이상인데 두 번쨔 operand기 X register이고, 첫 번째 operand도 register일 경우 nixbpe의 xbit가 1인 것이다.
본격적으로 nixbpe값을 정하는데 n=1, i=1을 defalt로 두었다. indirect addressing 명령이면 i=0, immediate addressing 명령이면 n=0이다.
n=1인 경우 4형식 명령이면 e=1, operand가 존재하면 p=1이다. operand가 null이면 nullpoiner 예외가 발생하므로 try-catch를 사용하였다. n=0, i=1인 immediate addressing인데 operand가 label인 경우 p=1이다.
nixbpe를 설정하은 setFlag 메서드로 한다. 파라미터의 flag에 해당하는 nixbpe의 bit를 value값으로 바꾼다. nixbpe의 한 bit값 확인은 getFlag 메서드로 한다.

TokenTable class는 Token class에서도 bit연산에 쓰이는 nixbpe flag 상수를 가진다. 맴버변수로 object code를 만드는데 필요한 symbol table, literal table, instruction table을 가지고 어셈블리 명령을 parsing한 Token class 객체들의 ArrayList와 M Record출력 내용을 저장한 ArrayList도 가진다.
TokenTable의 생성자에서 현재 section의 symTab, literalTab과 instTab을 파라미터로 입력받아 맴버함수인 symTab, literalTab, instTab이 각각의 객체를 가리키도록 하였다. 또한 tokenList와 mRecod 객체를 생성한다.
어셈블리 명령어를 parising하여 tokenList에 저장하는 메서드는 putToken, 파라미터로 입력되는 index의 token 객체를 리턴하는 메서드인 getToken이 있다.
TokenTable class에서 object code를 만드는 메서드는 makeObjectCode 이다. 파라미터로 입력받은 index의 TokenList token의 object code를 생성한다. object code가 생성되는 경우는 operator가 "WORD", "BYTE", literal, 기계어 명령인 경우이다.
"WORD"의 경우 '-'를 기준으로 operand1-operand2 연산을 요하는데, 둘다 extern 이여야 한다.(현재 section의 symbol이면 EQU로 연산을 한다) symTab.extref 에서 operand를 찾고 있으면 M Record를 세팅한다. (operand1은 +, operand2는 -이다) object code는 000000이다.
BYTE와 literal의 object code 생성 원리는 같다. Character이면 "'" 사이의 각 문자를 아스키코드로 바꾼 것이 object code이고 16진수면 그 자체가 object code이다. byteSize는 Character이면 "'"사이 문자열의 길이이고 16진수면 (문자열 길이+1)/2이다.
기계어 명령의 object code는 format에 따라 다르다. 1형식이면 opcode가 object code이다. 2형식이면 opcode+operand1의 register번호+ operand2의 register번호이다. 3형식의 경우 opcode 앞 6자리+ nixbpe + disp 12자리 총 24자리 bit이다. disp은 addresing 방식에 따라 정해진다.
simple, indirect addressing : operand 정의된 address - (현재 location+byteSize)
immediate addressing : operand 정의된 address - (현재 location+byteSize) or operand
4형식 명령은 operand가 extref에 있고 앞 12bit는 3형섹과 같으면 disp은 0으로 이루어진 20자리 bit이다.
각 형식의 byteSize는 1,2,3,4이다.

3장 수행결과



symtab_20180637 - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움

COPY	0
FIRST	0
CLOOP	3
ENDFIL	17
RETADR	2A
LENGTH	2D
BUFFER	33
BUFEND	1033
MAXLEN	1000
RDREC	0
RLOOP	9
EXIT	20
INPUT	27
MAXLEN	28
WRREC	0
WLOOP	6

literaltab_20180637 - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움

EOF	30
05	1B

output_20180637 - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

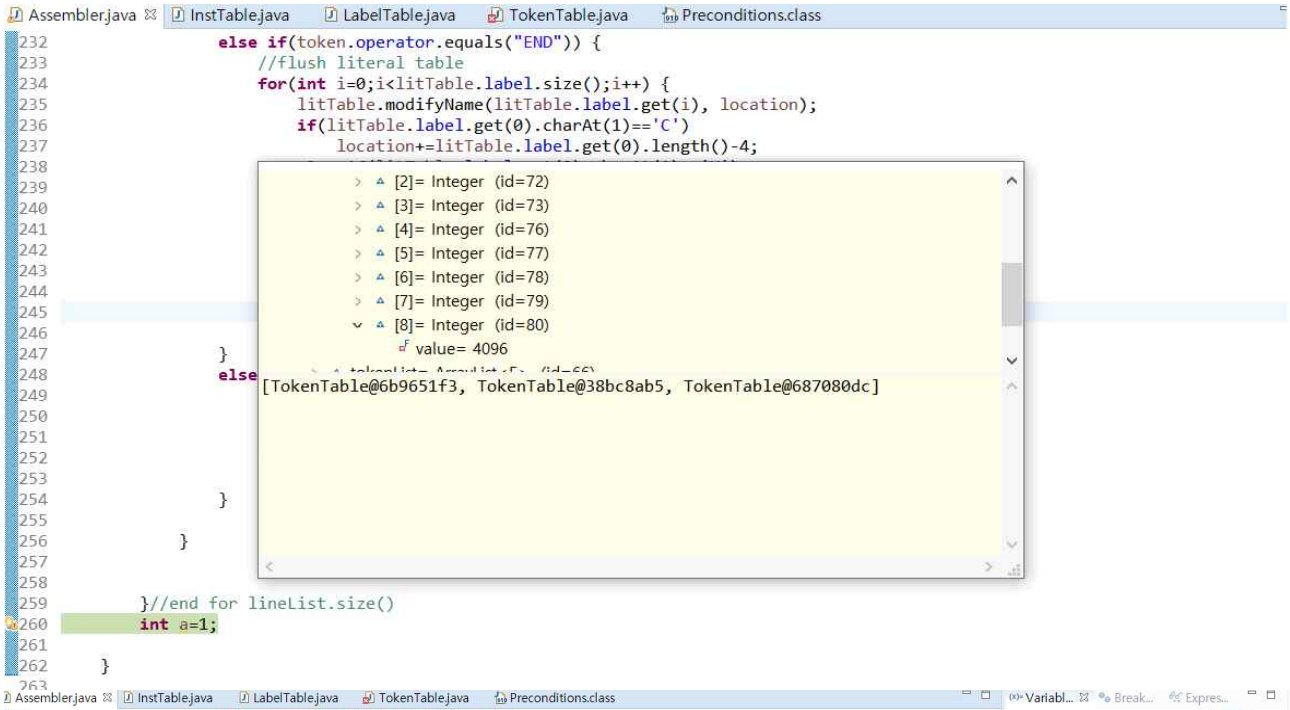
```
HCOPY   000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC  WRREC
T0000001D1720274B10000003202329000003320074B1000003F2FEC0320160F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000

HRDREC  000000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER
E

HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB41077100000E32012332FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E
```

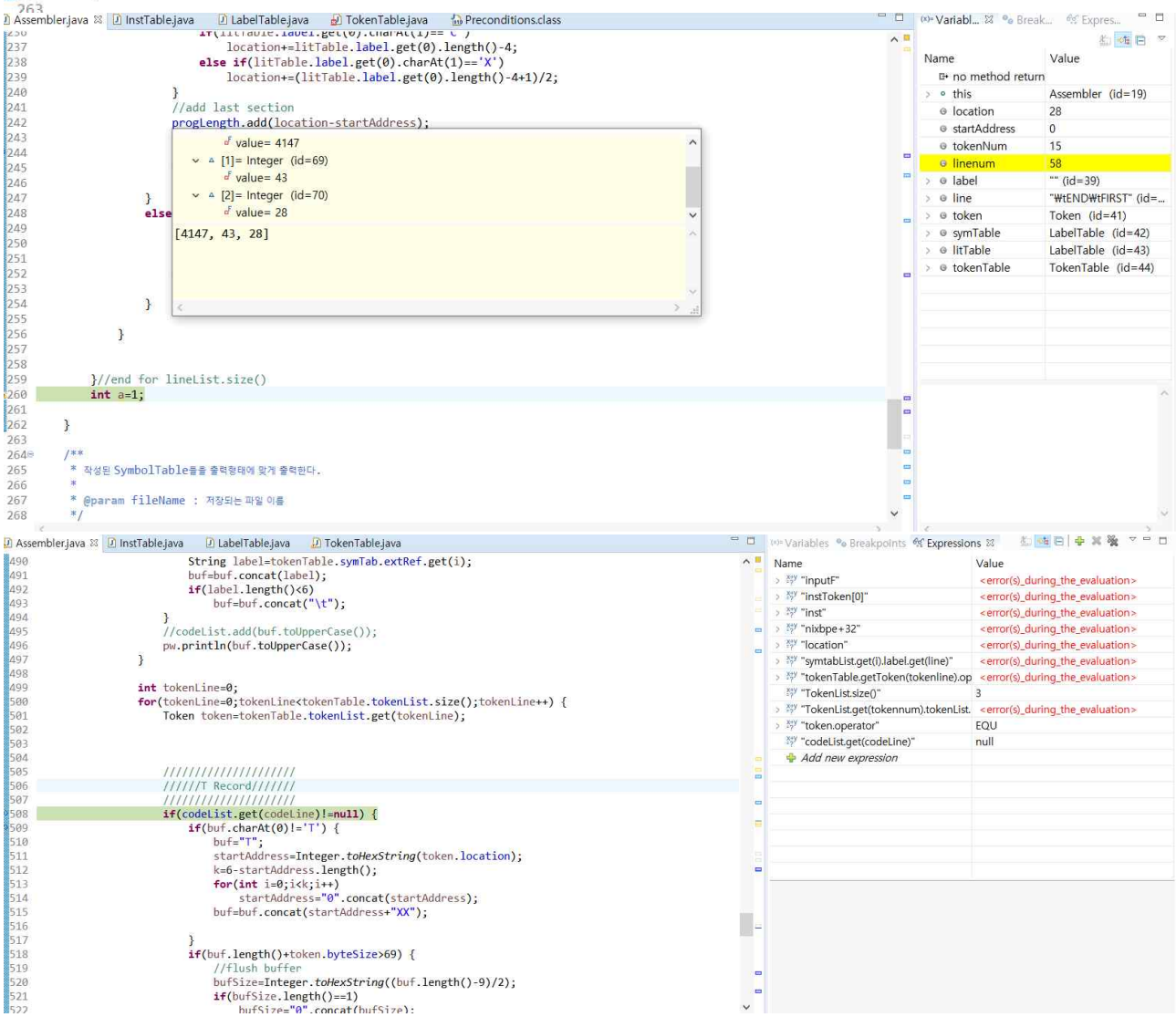
순서대로 symtab, literaltab, output

4장 결론 및 보충할 점

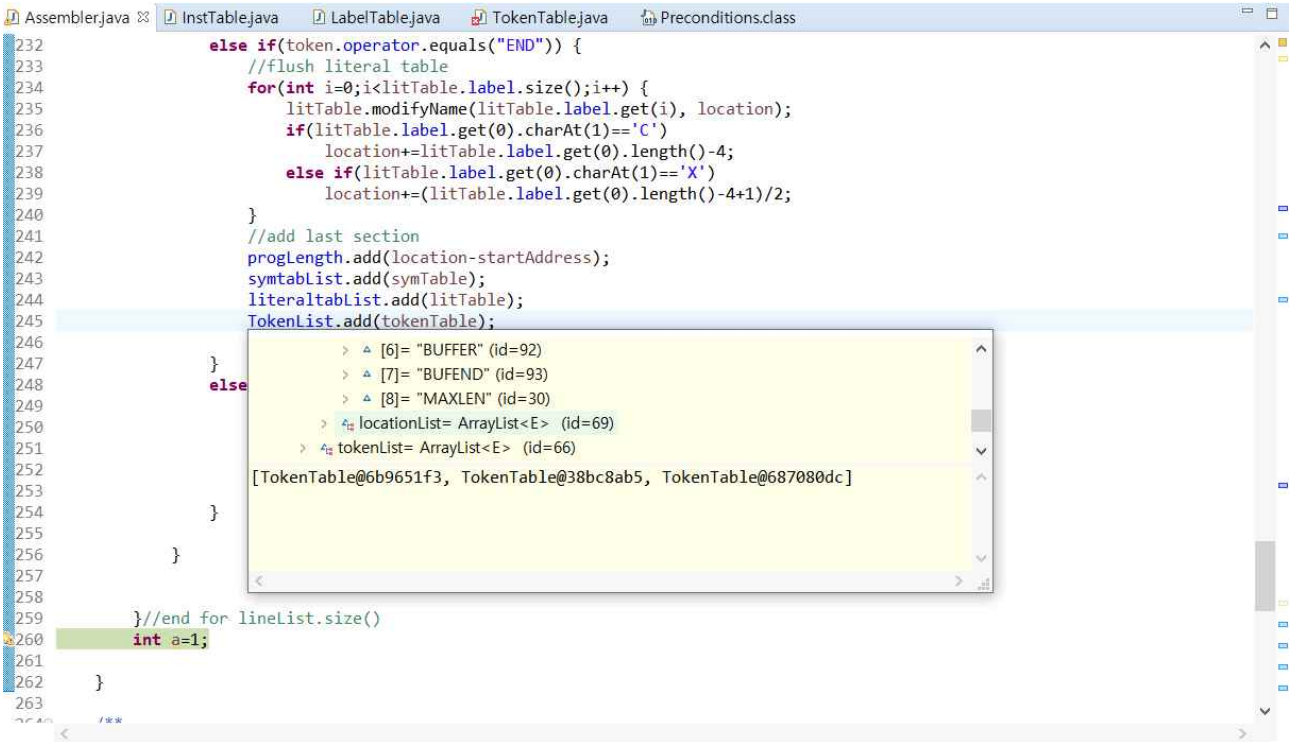
C언어로 구현했던 controll section 방식의 어셈블러를 자바로 구현하였다. Project 1때 미처 생각하지 못했던 부분을 새로운 언어의 형식으로 구현하다 버그가 있음을 알 수 있었다. 또한 Java의 문법과 객체지향 프로그래밍을 오랜만에 하여 그 원리를 상기할 수 있었다. C에서는 데이터를 저장하는 자료구조를 생성하는데 부담스러움이 있었는데, Java의 객체지향적인 프로그래밍 기법 덕에 부담이 없었다. 그러나 여전히 M Record를 깔끔하게 proccessing하는 법을 찾지 못해 다소 지저분하게 M Record를 만들고 출력하여 이부분에 보강을 해야한다.

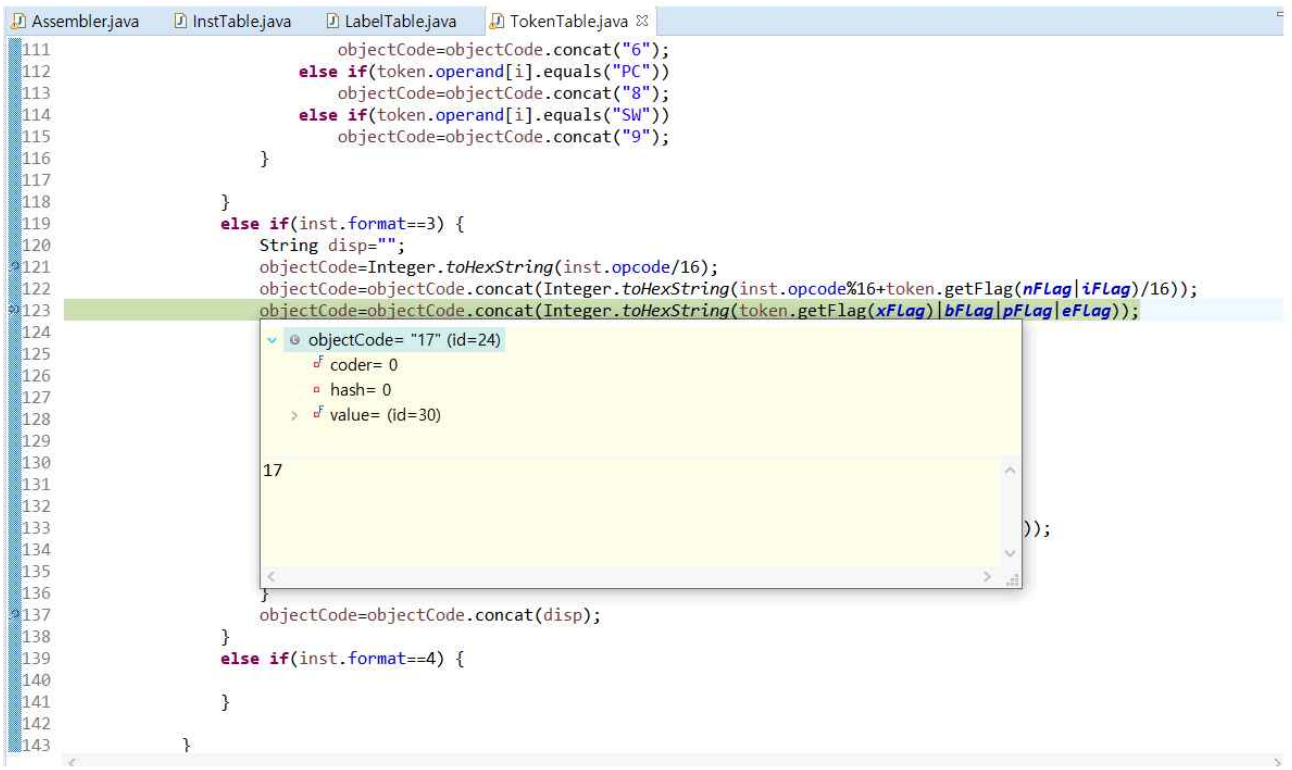
5장 디버깅

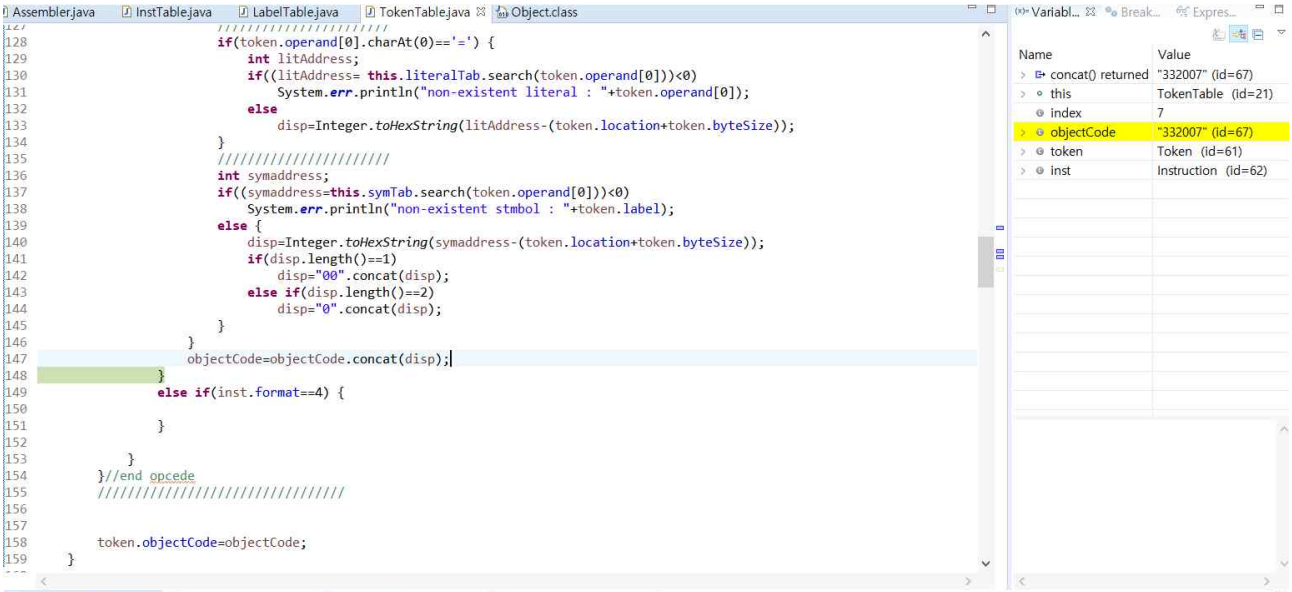

Assembler.java
InstTable.java
LabelTable.java
TokenTable.java
Preconditions.class
[2]= Integer (id=72)
[3]= Integer (id=73)
[4]= Integer (id=76)
[5]= Integer (id=77)
[6]= Integer (id=78)
[7]= Integer (id=79)
[8]= Integer (id=80)
value= 4096
[TokenTable@6b9651f3, TokenTable@38bc8ab5, TokenTable@687080dc]


value= 4147
[1]= Integer (id=69)
value= 43
[2]= Integer (id=70)
value= 28
[4147, 43, 28]
Variabl...
Name
Value
no method return
this
Assembler (id=19)
location
28
startAddress
0
tokenNum
15
linenum
58
label
"" (id=39)
line
"WtENDWtFIRST" (id=...
token
Token (id=41)
symTable
LabelTable (id=42)
litTable
LabelTable (id=43)
tokenTable
TokenTable (id=44)

Variables
Breakpoints
Expressions
"inputF"
"instToken[0]"
"inst"
"nixbpe+32"
"location"
"symtabList.get(i).label.get(line)"
"tokenTable.getToken(tokenline).op
"TokenList.size()"
3
"TokenList.get(tokennum).tokenList.
"token.operator"
EQU
"codeList.get(codeLine)"
null
Add new expression
<error(s)_during_the_evaluation>


Assembler.java
InstTable.java
LabelTable.java
TokenTable.java
Preconditions.class
else if(token.operator.equals("END")) {
//flush literal table
for(int i=0;i<litTable.label.size();i++) {
litTable.modifyName(litTable.label.get(i), location);
if(litTable.label.get(0).charAt(1)=='C')
location+=litTable.label.get(0).length()-4;
else if(litTable.label.get(0).charAt(1)=='X')
location+=(litTable.label.get(0).length()-4+1)/2;
}
//add last section
progLength.add(location-startAddress);
symtabList.add(symTable);
literaltabList.add(litTable);
TokenList.add(tokenTable);
[6]= "BUFFER" (id=92)
[7]= "BUFEND" (id=93)
[8]= "MAXLEN" (id=30)
locationList= ArrayList<E> (id=69)
tokenList= ArrayList<E> (id=66)
[TokenTable@6b9651f3, TokenTable@38bc8ab5, TokenTable@687080dc]
}//end for lineList.size()
int a=1;

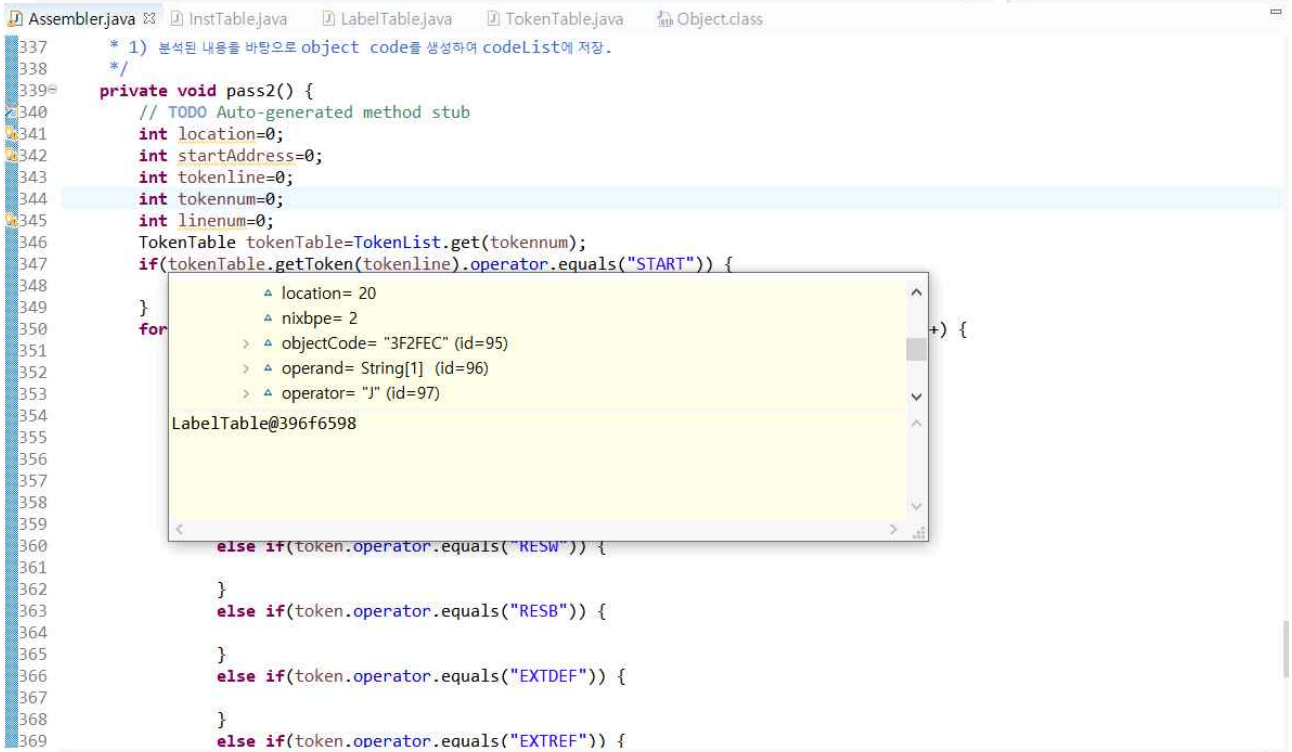

Assembler.java
InstTable.java
LabelTable.java
TokenTable.java
objectCode=objectCode.concat("6");
else if(token.operand[i].equals("PC"))
objectCode=objectCode.concat("8");
else if(token.operand[i].equals("SW"))
objectCode=objectCode.concat("9");
}
}
else if(inst.format==3) {
String disp="";
objectCode=Integer.toHexString(inst.opcode/16);
objectCode=objectCode.concat(Integer.toHexString(inst.opcode%16+token.getFlag(nFlag|iFlag)/16));
objectCode=objectCode.concat(Integer.toHexString(token.getFlag(xFlag)|bFlag|pFlag|eFlag));
objectCode= "17" (id=24)
coder= 0
hash= 0
value= (id=30)
17
));
}
objectCode=objectCode.concat(disp);
}
else if(inst.format==4) {
}
}



```
Assembler.java  InstTable.java  LabelTable.java  TokenTable.java  Object.class

                    //////////////////////
                if(token.operand[0].charAt(0)=='=') {
                    int litAddress;
                    if((litAddress= this.literalTab.search(token.operand[0]))<0)
                        System.err.println("non-existent literal : "+token.operand[0]);
                    else
                        disp=Integer.toHexString(litAddress-(token.location+token.byteSize));
                }
                //////////////////////
                int symaddress;
                if((symaddress=this.symTab.search(token.operand[0]))<0)
                    System.err.println("non-existent stmbol : "+token.label);
                else {
                    disp=Integer.toHexString(symaddress-(token.location+token.byteSize));
                    if(disp.length()==1)
                        disp="00".concat(disp);
                    else if(disp.length()==2)
                        disp="0".concat(disp);
                }
            }
            objectCode=objectCode.concat(disp);
        }
        else if(inst.format==4) {

        }

    }
}//end opcede
////////////////////////////////


token.objectCode=objectCode;
}
```

Variabl... | Break... | Expres...

Name	Value
concat() returned	"332007" (id=67)
this	TokenTable (id=21)
index	7
objectCode	"332007" (id=67)
token	Token (id=61)
inst	Instruction (id=62)



```
Assembler.java  InstTable.java  LabelTable.java  TokenTable.java  Object.class

 * 1) 분석된 내용을 바탕으로 object code를 생성하여 codeList에 저장.
 */
private void pass2() {
    // TODO Auto-generated method stub
    int location=0;
    int startAddress=0;
    int tokenline=0;
    int tokennum=0;
    int linenum=0;
    TokenTable tokenTable=TokenList.get(tokennum);
    if(tokenTable.getToken(tokenline).operator.equals("START")) {

    }
    for                                                    +) {

            else if(token.operator.equals("RESW")) {

            }
            else if(token.operator.equals("RESB")) {

            }
            else if(token.operator.equals("EXTDEF")) {

            }
            else if(token.operator.equals("EXTREF")) {
```

location= 20
nixbpe= 2
objectCode= "3F2FEC" (id=95)
operand= String[1] (id=96)
operator= "J" (id=97)

LabelTable@396f6598

6장 소스코드(+주석)

//Assembler.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.PrintWriter;
/**
 * Assembler: 이 프로그램은 SIC/XE 머신을 위한 Assembler 프로그램의 메인루틴이다. 프로그램의 수행 작업은 다음과 같다.
 * 1) 처음 시작하면 Instruction 명세를 읽어들여서 assembler를 세팅한다.
 *
 * 2) 사용자가 작성한 input 파일을 읽어들인 후 저장한다
 *
 * 3) input 파일의 문장들을 단어별로 분할하고 의미를 파악해서 정리한다. (pass1)
 *
 * 4) 분석된 내용을바탕으로 컴퓨터가 사용할 수 있는 object code를 생성한다. (pass2)
 *
 *
 * 작성중의 유의사항:
 *
 * 1) 새로운 클래스, 새로운 변수, 새로운 함수 선언은 얼마든지 허용됨. 단, 기존의 변수와 함수들을 삭제하거나 완전히 대체하는 것은
 * 안된다.
 *
 * 2) 마찬가지로 작성된 코드를 삭제하지 않으면 필요에 따라 예외처리, 인터페이스 또는 상속 사용 또한 허용됨
 *
 * 3) 모든 void 타입의 리턴값은 유저의 필요에 따라 다른 리턴 타입으로 변경 가능.
 *
 * 4) 파일, 또는 콘솔창에 한글을 출력시키지말 것. (채점상의 이유. 주석에 포함된 한글은 상관없음)
 *
 * + 제공하는 프로그램 구조의 개선방법을 제안하고 싶은 분들은 보고서의 결론 뒷부분에 첨부 바랍니다. 내용에 따라 가산점이 있을 수
 * 있습니다.
 */

public class Assembler {
	/** instruction 명세를 저장한 공간 */
	InstTable instTable;
	/** 읽어들인 input 파일의 내용을 한 줄 씩 저장하는 공간. */
	ArrayList<String> lineList;
	/** 프로그램의 section별로 symbol table을 저장하는 공간 */
	ArrayList<LabelTable> symtabList;
	/** 프로그램의 section별로 literal table을 저장하는 공간 */
	ArrayList<LabelTable> literaltabList;
	/** 프로그램의 section별로 프로그램을 저장하는 공간 */
	ArrayList<TokenTable> TokenList;
	/**
	 * Token, 또는 지시어에 따라 만들어진 오브젝트 코드들을 출력 형태로 저장하는 공간. 필요한 경우 String 대신 별도의 클래스를
	 * 선언하여 ArrayList를 교체해도 무방함.
	 */
	ArrayList<String> codeList;

	/** 프로그램 section별로 프로그램 크기를 저장하는 공간*/
	ArrayList<Integer> progLength;

	/**
	 * 클래스 초기화. instruction Table을 초기화와 동시에 세팅한다.
	 *
	 * @param instFile : instruction 명세를 작성한 파일 이름.
	 */
	public Assembler(String instFile) {
		instTable = new InstTable(instFile);
		lineList = new ArrayList<String>();
		symtabList = new ArrayList<LabelTable>();
		literaltabList = new ArrayList<LabelTable>();
		TokenList = new ArrayList<TokenTable>();
		codeList = new ArrayList<String>();
		//...
		progLength=new ArrayList<Integer>();

	}

	/**
	 * 어셈블러의 메인 루틴
	 */
	public static void main(String[] args) {
		Assembler assembler = new Assembler("inst.data");

assembler.loadInputFile("input.txt");
		assembler.pass1();


assembler.printSymbolTable("symtab_20180637");

assembler.printLiteralTable("literaltab_20180637");
		assembler.pass2();

assembler.printObjectCode("output_20180637");

	}

	/**
	 * inputFile을 읽어들여서 lineList에 저장한다.
	 *
	 * @param inputFile : input 파일 이름.
	 */
```

```
	private void loadInputFile(String inputFile) {
		// TODO Auto-generated method stub
		String inputF="./";
		inputF= inputF.concat(inputFile);

		try {
		BufferedReader br= new BufferedReader(new FileReader(inputF));
		String line;

while((line=br.readLine())!=null)
		lineList.add(line);

		br.close();
		}
		catch(Exception e){

System.err.println(inputF);
		}

	}

	/**
	 * pass1 과정을 수행한다.
	 *
	 * 1) 프로그램 소스를 스캔하여 토큰 단위로 분리한 뒤 토큰 테이블을 생성.
	 *
	 * 2) symbol, literal 들을 SymbolTable, LiteralTable에 정리.
	 *
	 * 주의사항: SymbolTable, LiteralTable, TokenTable은 프로그램의 section별로 하나씩 선언되어야 한다.
	 *
	 * @param inputFile : input 파일 이름.
	 */
	private void pass1() {
		// TODO Auto-generated method stub
		int location=0;
		int startAddress=0;
		int tokenNum=0;
		int linenum=0;
		String label;
		String line;
		Token token;
		LabelTable symTable = new LabelTable();
		LabelTable litTable = new LabelTable();
		TokenTable tokenTable = new TokenTable(symTable,litTable,instTable);
		////////////////////////////////
		///////parsing first line//////////
		////////////////////////////////
		line=lineList.get(linenum);
		tokenTable.putToken(line);

token=tokenTable.getToken(tokenNum++);
		label=token.label;

if(token.operator.equals("START")){

startAddress=Integer.parseInt(token.operand[0]);
		location=startAddress;

symTable.putName(label, location);
		}

for(linenum=1;linenum<lineList.size();linenum++) {


line=lineList.get(linenum);


tokenTable.putToken(line);

token=tokenTable.getToken(tokenNum++);
		label=token.label;

token.location=location;

		if(!label.equals(".")){//if not comment

if(token.operator.equals("CSECT")){
		//add last section

progLength.add(location-startAddress);

symtabList.add(symTable);

literaltabList.add(litTable);

TokenList.add(tokenTable);


//initial new section

symTable = new LabelTable();

litTable = new LabelTable();

tokenTable = new TokenTable(symTable,litTable,instTable);

tokenTable.putToken(line);

startAddress=0;

location=0;

tokenNum=1;

		}

////////////////////////////
		////////symbol table/////////

////////////////////////////

if(!label.equals("")){

if(symTable.search(label)<0)

	symTable.putName(label, location);
		else

	System.err.println("twice define symbol: "+label);
```

```
                                    }
//////////////////////////
                                    ////////literal
table////////
//////////////////////////
                                    try {
if(token.operand[0].charAt(0)=='='){


        if(litTable.search(token.operand[0])<0)

litTable.putName(token.operand[0], location);

        else

litTable.modifyName(token.operand[0],
location);
                                                }
                                    }
catch(Exception e) {

                                    }
//////////////////////////
///////////EXTDEF////////////
//////////////////////////
if(token.operator.equals("EXTDEF")){
                                                String
extDef[]=new String[3];
for(int i=0;i<token.operand.length;i++) {
        extDef[i]=token.operand[i];
        symTable.extDef.add(extDef[i]);
                                                }
                                    }
//////////////////////////
///////////EXTDEF////////////
//////////////////////////
                                    else
if(token.operator.equals("EXTREF")){
                                                String
extRef[]=new String[3];
for(int i=0;i<token.operand.length;i++) {
        extRef[i]=token.operand[i];
        symTable.extRef.add(extRef[i]);
                                                }
                                    }
//////////////////////////
                                    else
if(token.operator.equals("RESW")){
location+=3*Integer.parseInt(token.operand[0]
);
                                                }
                                    else
if(token.operator.equals("RESB")){
location+=Integer.parseInt(token.operand[0]);
                                                }
                                    else
if(token.operator.equals("EQU")){
if(!token.operand[0].equals("*")) {
        if(token.operand[0].contains("-")) {
                String
operand[]=token.operand[0].split("-");
                symTable.modifyName(label,
symTable.search(operand[0])-symTable.searc
h(operand[1]));
        }


                                                }
                                    }//end if EQU
                                    else
if(token.operator.equals("WORD")){
location+=3;
                                                }
                                    else
if(token.operator.equals("BYTE")){
if(token.operand[0].charAt(0)=='X')

location+=(token.operand[0].length()-3+1)/2;
                                                else
if(token.operand[0].charAt(0)=='C')

location+=token.operand[0].length()-3;
                                                }
                                    else
if(token.operator.equals("LTORG")){
for(int i=0;i<litTable.label.size();i++) {
        line="*₩t".concat(litTable.label.get(i));
        tokenTable.putToken(line);

litTable.modifyName(litTable.label.get(i),
location);
        if(litTable.label.get(0).charAt(1)=='C')

location+=litTable.label.get(0).length()-4;

        else
if(litTable.label.get(0).charAt(1)=='X')

location+=(litTable.label.get(0).length()-4+1)/
2;
```

```
            tokenNum++;
                                    }

                                }//LTORG
                                else
if(token.operator.equals("CSECT")){

                                }
                                else
if(token.operator.equals("END")) {

//flush literal table

for(int i=0;i<litTable.label.size();i++) {

            line="*₩t".concat(litTable.label.get(i));

            tokenTable.putToken(line);


litTable.modifyName(litTable.label.get(i),
location);

            if(litTable.label.get(0).charAt(1)=='C')


location+=litTable.label.get(0).length()-4;

            else
if(litTable.label.get(0).charAt(1)=='X')


location+=(litTable.label.get(0).length()-4+1)/
2;

            tokenNum++;
                                    }
                                    //add
last section

progLength.add(location-startAddress);

symtabList.add(symTable);

literaltabList.add(litTable);

TokenList.add(tokenTable);

                                }
                                else
{//opcode

Instruction inst;

if(instTable.containInst(token.operator))


location+=instTable.get(token.operator).format
;

                                    else
{

            if(!label.equals("*"))


System.err.println("Non-existent operator:
"+token.operator);
                                    }
                                }

                            }


                        }//end for lineList.size()
                        int a=1;

            }

            /**
             * 작성된 SymbolTable들을 출력형태에
맞게 출력한다.
             *
             * @param fileName : 저장되는 파일
이름
             */
            private void printSymbolTable(String
fileName) {
                        // TODO Auto-generated
method stub
                        String outputF="./";
                        outputF=
outputF.concat(fileName);

                        try {
                                    PrintWriter pw= new
PrintWriter(outputF);
                                    for(int
i=0;i<symtabList.size();i++) {
                                                for(int
line=0;line<symtabList.get(i).label.size();line++
) {

/*pw.print(symtabList.get(i).label.get(line)+"₩t
");

pw.println(symtabList.get(i).search(symtabList.
get(i).label.get(line)));
                                                */
                                                String
label = symtabList.get(i).label.get(line);

pw.print(label +"₩t₩t");
                                                int
address= symtabList.get(i).search(label);

pw.println(Integer.toHexString(address).toUppe
rCase());
                                                }
                                                pw.println("");

//System.out.println("");
                                    }


                                    pw.close();
                        }
                        catch(Exception e){
                                    System.err.println(e +
outputF);
                        }


            }

            /**
             * 작성된 LiteralTable들을 출력형태에
맞게 출력한다.
```

```
		 *
		 * @param fileName : 저장되는 파일
이름
		 */
		private void printLiteralTable(String
fileName) {
			// TODO Auto-generated
method stub
			String outputF="./";

outputF=outputF.concat(fileName);
			try {
				PrintWriter pw= new
PrintWriter(outputF);
				for(int
i=0;i<literaltabList.size();i++) {
					for(int
line=0;line<literaltabList.get(i).label.size();line+
+) {
						String
label=literaltabList.get(i).label.get(line);
						String
literal[]=label.split("'");

pw.print(literal[1]+"₩t₩t");

pw.println(Integer.toHexString(literaltabList.get
(i).search(label)).toUpperCase());
					}

				}
				pw.close();
			}
			catch(Exception e) {

			}

		}

		/**
		 * pass2 과정을 수행한다.
		 *
		 * 1) 분석된 내용을 바탕으로 object
code를 생성하여 codeList에 저장.
		 */
		private void pass2() {
			// TODO Auto-generated
method stub
			int startAddress=0;
			int tokenline=0;
			int tokennum=0;
			int linenum=0;
			TokenTable
tokenTable=TokenList.get(tokennum);

if(tokenTable.getToken(tokenline).operator.equ
als("START")) {

			}

for(tokenline=0;tokenline<tokenTable.tokenList
.size();tokenline++) {
				Token
token=tokenTable.getToken(tokenline);


if(!token.label.equals(".")) {

if(token.operator.equals("CSECT")) {

tokennum++;

tokenline=1;

tokenTable=TokenList.get(tokennum);
					}//if operator
= CESCT
					else
if(token.operator.equals("START")) {

					}
					else
if(token.operator.equals("RESW")) {

					}
					else
if(token.operator.equals("RESB")) {

					}
					else
if(token.operator.equals("EXTDEF")) {

					}
					else
if(token.operator.equals("EXTREF")) {

					}
					else
if(token.operator.equals("EQU")) {

					}
					//else
if(token.operator.equals("END")) {

					//}
					else {

tokenTable.makeObjectCode(tokenline);

//location+=token.byteSize;
					}


codeList.add(tokenTable.getObjectCode(tokenli
ne));
				}//not commend line

			}//for not end
			//////////////////////////
			/////flush literal table/////
			//////////////////////////
			try {

tokenTable.makeObjectCode(tokenline);

codeList.add(tokenTable.getObjectCode(tokenli
ne));
			}
			catch(Exception e) {

			}

		}

		/**
		 * 작성된 codeList를 출력형태에 맞게
출력한다.
		 *
		 * @param fileName : 저장되는 파일
이름
```

```
	*/
	private void printObjectCode(String fileName) {
		// TODO Auto-generated method stub
		try {
			fileName="./".concat(fileName);
			PrintWriter pw= new PrintWriter(fileName);
			int codeLine=1;
			for(int tokenNum=0;tokenNum<this.TokenList.size();tokenNum++) {
				String buf;
				String bufSize;
				TokenTable tokenTable=this.TokenList.get(tokenNum);
//////////////////
				//////H Record//////
//////////////////
buf="H"+tokenTable.symTab.label.get(0);
if(tokenTable.symTab.label.get(0).length()<6)
buf=buf.concat("₩t");
				/**start address**/
				String startAddress=Integer.toHexString(tokenTable.symTab.locationList.get(0));
				int k=6-startAddress.length();
				for(int i=0;i<k;i++)
startAddress="0".concat(startAddress);
buf=buf.concat(startAddress);
				/**program size;**/
				String progSize=Integer.toHexString(this.progLength.get(tokenNum));
k=6-progSize.length();
				for(int i=0;i<k;i++)
progSize="0".concat(progSize);
buf=buf.concat(progSize);

//codeList.add(buf.toUpperCase());
pw.println(buf.toUpperCase());

//////////////////
				//////D Record///////
//////////////////
					buf="D";
if(tokenTable.symTab.extDef.size()>0) {
for(int i=0;i<tokenTable.symTab.extDef.size();i++) {
		String label=tokenTable.symTab.extDef.get(i);
		buf=buf.concat(label);
		if(label.length()<6)
			buf=buf.concat("₩t");
		String address=Integer.toHexString(tokenTable.symTab.search(label));
		int m=6-address.length();
		for(int t=0;t<m;t++)
			address="0".concat(address);
		buf=buf.concat(address);

				}
//codeList.add(buf.toUpperCase());
pw.println(buf.toUpperCase());
				}
//////////////////
				//////R Record///////
//////////////////
					buf="R";
if(tokenTable.symTab.extRef.size()>0) {
for(int i=0;i<tokenTable.symTab.extRef.size();i++) {
		String label=tokenTable.symTab.extRef.get(i);
		buf=buf.concat(label);
		if(label.length()<6)
			buf=buf.concat("₩t");
				}
//codeList.add(buf.toUpperCase());
pw.println(buf.toUpperCase());
				}
				int tokenLine;
for(tokenLine=1;tokenLine<tokenTable.tokenList.size();tokenLine++) {
```

```
Token token=tokenTable.tokenList.get(tokenLine);


//////////////////////
//////T Record///////
//////////////////////
if(!token.label.equals(".")) {
        if(codeList.get(codeLine)!=null) {
                if(buf.charAt(0)!='T') {
                        buf="T";

startAddress=Integer.toHexString(token.location);

k=6-startAddress.length();
                        for(int i=0;i<k;i++)

startAddress="0".concat(startAddress);

buf=buf.concat(startAddress+"XX");


                }

if(buf.length()+token.byteSize>69) {
                        //flush buffer

bufSize=Integer.toHexString((buf.length()-9)/2);
                        if(bufSize.length()==1)

bufSize="0".concat(bufSize);

buf=buf.replaceFirst("XX",bufSize);

//codeList.add(buf.toUpperCase());

pw.println(buf.toUpperCase());


                        //initial buffer
                        buf="T";

startAddress=Integer.toHexString(token.location);

k=6-startAddress.length();
                        for(int i=0;i<k;i++)

startAddress="0".concat(startAddress);

buf=buf.concat(startAddress+"XX");
                }

buf=buf.concat(codeList.get(codeLine));
        }
        else {

if(buf.length()>9&&buf.charAt(0)=='T')
{//buffer에 objectcode 존재
                        //flush buffer

bufSize=Integer.toHexString((buf.length()-9)/2);
                        if(bufSize.length()==1)

bufSize="0".concat(bufSize);

buf=buf.replaceFirst("XX",bufSize);

//codeList.add(buf.toUpperCase());

pw.println(buf.toUpperCase());
                        buf="X";
                }

                else {//buffer에 objectcode 없음


                }

        }
        codeLine++;
                                }
                        }//for section
//////////////////////////
                        ////////flush
```

```
buffer////////
//////////////////////////
if((buf.length()-9)/2>0){
bufSize=Integer.toHexString((buf.length()-9)/2)
;
if(bufSize.length()==1)
        bufSize="0".concat(bufSize);
buf=buf.replaceFirst("XX",bufSize);
//codeList.add(buf.toUpperCase());
pw.println(buf.toUpperCase());
                                    }
//////////////////////////

//////////////////////
                                    //////M
Record///////
//////////////////////
                                    for(int
i=0;i<tokenTable.mRecord.size();i++ ) {
buf="M";
buf=buf.concat(tokenTable.mRecord.get(i));
//codeList.add(buf);
pw.println(buf.toUpperCase());
                                    }

//////////////////////
                                    //////E
Record///////
//////////////////////
                                    buf="E";
if(tokenNum==0) {
startAddress=Integer.toHexString(tokenTable.s
ymTab.locationList.get(0));
k=6-startAddress.length();
for(int  i=0;i<k;i++ )
        startAddress="0".concat(startAddress);
buf=buf.concat(startAddress);
                                    }
//codeList.add(buf.toUpperCase()+ "₩n");
pw.println(buf.toUpperCase()+ '₩n');

                        }//for  tokenList
                                        pw.close();
                                    }
                                    catch(Exception  e) {
System.err.println(fileName+ e);
                                    }

                }
}
```

```
//InstTable.java

import java.util.HashMap;
import java.io.FileReader;
import java.io.BufferedReader;
/**
 * 모든 instruction의 정보를 관리하는 클래스. instruction data들을 저장한다 또한 instruction 관련 연산,
 * 예를 들면 목록을 구축하는 함수, 관련 정보를 제공하는 함수 등을 제공 한다.
 */
public class InstTable {
	/**
	 * inst.data 파일을 불러와 저장하는 공간. 명령어의 이름을 집어넣으면 해당하는 Instruction의 정보들을 리턴할 수 있다.
	 */
	HashMap<String, Instruction> instMap;

	/**
	 * 클래스 초기화. 파싱을 동시에 처리한다.
	 *
	 * @param instFile : instuction에 대한 명세가 저장된 파일 이름
	 */

	public InstTable(String instFile) {
		instMap = new HashMap<String, Instruction>();
		openFile(instFile);
	}

	/**
	 * 입력받은 이름의 파일을 열고 해당 내용을 파싱하여 instMap에 저장한다.
	 */
	public void openFile(String fileName) {
		// ...
		String inputF="./";
		inputF= inputF.concat(fileName);
		try {
			BufferedReader br= new BufferedReader(new FileReader(inputF));
			String line;

while((line=br.readLine())!=null) {
				Instruction inst= new Instruction(line);

instMap.put(inst.instruction,inst);

			}

			br.close();
			}
			catch(Exception e){

System.err.println(inputF);
			}

	}

	// get, set, search 등의 함수는 자유 구현
	public boolean containInst(String instruction) {
		return instMap.containsKey(instruction);
	}
	public Instruction get(String instruction) {
		return instMap.get(instruction);
	}

}

/**
 * 명령어 하나하나의 구체적인 정보는 Instruction클래스에 담긴다. instruction과 관련된 정보를 저장하고 기초적인 연산을
 * 수행한다.
 */
class Instruction {

	/*
	 * 각자의 inst.data 파일에 맞게 저장하는 변수를 선언한다.
	 *
	 * ex) String instruction; int opcode; int numberOfOperand; String comment;
	 */
	String instruction; int opcode; int numberOfOperand;

	/** instruction이 몇 바이트 명령어인지 저장. 이후 편의성을 위함 */
	int format;

	/**
	 * 클래스를 선언하면서 일반문자열을 즉시 구조에 맞게 파싱한다.
	 *
	 * @param line : instruction 명세파일로부터 한줄씩 가져온 문자열
	 */
	public Instruction(String line) {
		parsing(line);
	}

	/**
	 * 일반 문자열을 파싱하여 instruction 정보를 파악하고 저장한다.
	 *
	 * @param line : instruction 명세파일로부터 한줄씩 가져온 문자열
	 */
	public void parsing(String line) {
		// TODO Auto-generated method stub
		String instToken[]=line.split(",");
		instruction=instToken[0];

format=Integer.parseInt(instToken[1]);

opcode=Integer.decode("0x".concat(instToken[2]));

numberOfOperand=Integer.parseInt(instToken[3]);
	}

	// 그 외 함수 자유 구현
}
```

```
//LableTable.java
import java.util.ArrayList;

/**
 * symbol, literal과 관련된 데이터와 연산을 소유한다. section 별로 하나씩 인스턴스를 할당한다.
 */
public class LabelTable {
	ArrayList<String> label;
	ArrayList<Integer> locationList;
	// external 선언 및 처리방법을 구현한다.
	ArrayList<String> extDef;
	ArrayList<String> extRef;


	/**
	 * 새로운 symbol과 literal을 table에 추가한다.
	 *
	 * @param label    : 새로 추가되는 symbol 혹은 literal의 lable
	 * @param location : 해당 symbol 혹은 literal이 가지는 주소값 주의 : 만약 중복된 symbol, literal이
	 *                 putName을 통해서 입력된다면 이는 프로그램 코드에 문제가 있음을 나타낸다. 매칭되는 주소값의 변경은
	 *                 modifylable()을 통해서 이루어져야 한다.
	 */
	public void putName(String label, int location) {

		this.label.add(label);
		this.locationList.add(location);
	}

	/**
	 * 기존에 존재하는 symbol, literal 값에 대해서 가리키는 주소값을 변경한다.
	 *
	 * @param lable       : 변경을 원하는 symbol, literal의 label
	 * @param newLocation : 새로 바꾸고자 하는 주소값
	 */
	public void modifyName(String lable, int newLocation) {
		int index = this.label.indexOf(lable);
		this.locationList.set(index, newLocation);
	}

	/**
	 * 인자로 전달된 symbol, literal이 어떤 주소를 지칭하는지 알려준다.
	 *
	 * @param label : 검색을 원하는 symbol 혹은 literal의 label
	 * @return address: 가지고 있는 주소값. 해당 symbol, literal이 없을 경우 -1 리턴
	 */
	public int search(String label) {
		int address = 0;
		// ...
		int index;
		if((index = this.label.indexOf(label))>=0) {
			Integer obj=this.locationList.get(index);

address=obj.intValue();
		}
		else
			address=-1;
		return address;
	}

	//추가 구현
	/**
	 *
	 * 맴버 변수 ArrayList의 객체를 생성한다.
	 *
	 */
	public LabelTable() {
		label = new ArrayList<String>() ;
		locationList = new ArrayList<Integer>() ;
		extDef = new ArrayList<String>();
		extRef = new ArrayList<String>();

	}

}
```

```
//TokenTable.java
import java.util.ArrayList;

/**
 * 사용자가 작성한 프로그램 코드를 단어별로 분할 한 후, 의미를 분석하고, 최종 코드로 변환하는 과정을 총괄하는 클래스이다.
 *
 * pass2에서 object code로 변환하는 과정은 혼자 해결할 수 없고 symbolTable과 instTable의 정보가 필요하므로
 * 이를 링크시킨다. section 마다 인스턴스가 하나씩 할당된다.
 *
 */
public class TokenTable {
	public static final int MAX_OPERAND = 3;

	/* bit 조작의 가독성을 위한 선언 */
	public static final int nFlag = 32;
	public static final int iFlag = 16;
	public static final int xFlag = 8;
	public static final int bFlag = 4;
	public static final int pFlag = 2;
	public static final int eFlag = 1;

	/* Token을 다룰 때 필요한 테이블들을 링크시킨다. */
	LabelTable symTab;
	LabelTable literalTab;
	InstTable instTab;
	/** 각 line을 의미별로 분할하고 분석하는 공간. */
	ArrayList<Token> tokenList;

	/**object program M Record를 저장하는 공간*/
	ArrayList<String> mRecord;


	/**
	 * 초기화하면서 symTable과 instTable을 링크시킨다.
	 *
	 * @param symTab     : 해당 section과 연결되어있는 symbol table
	 * @param literaTab : 해당 section과 연결되어있는 literal table
	 * @param instTab    : instruction 명세가 정의된 instTable
	 */
	public TokenTable(LabelTable symTab, LabelTable literalTab, InstTable instTab) {
		// ...
		this.symTab=symTab;
		this.literalTab=literalTab;
		this.instTab=instTab;
		tokenList= new ArrayList<Token>();
		this.mRecord=new ArrayList<String>();
	}

	/**
	 * 일반 문자열을 받아서 Token단위로 분리시켜 tokenList에 추가한다.
	 *
	 * @param line : 분리되지 않은 일반 문자열
	 */
	public void putToken(String line) {
		tokenList.add(new Token(line));
	}

	/**
	 * tokenList에서 index에 해당하는 Token을 리턴한다.
	 *
	 * @param index
	 * @return : index번호에 해당하는 코드를 분석한 Token 클래스
	 */
	public Token getToken(int index) {
		return tokenList.get(index);
	}

	/**
	 * Pass2 과정에서 사용한다. instruction table, symbol table 등을 참조하여 objectcode를 생성하고, 이를
	 * 저장한다.
	 *
	 * @param index
	 */
	public void makeObjectCode(int index) {
		// ...
		String objectCode="";
		Token token=getToken(index);

		//////////////////////////
		//////////WORD//////////
		//////////////////////////

if(token.operator.equals("WORD")) {

if(token.operand[0].contains("-")) {
		String operand[]=token.operand[0].split("-");
		for(int i=0;i<operand.length;i++) {

if(this.symTab.search(operand[i])<0) {

	//EXTREF찾기 구현

	int ext=0;


for(ext=0;ext<this.symTab.extRef.size();ext++) {


if(this.symTab.extRef.get(ext).equals(operand[i]))

			break;

	}

	if(ext==this.symTab.extRef.size()) {


System.err.println("non-exitent symbol : "+ operand[i]);
```

```
		}
		else {
			if(i==0) {
				String location=Integer.toHexString(token.location);
				int k=6-location.length();
				for(int a=0;a<k;a++)
location="0".concat(location);
this.mRecord.add(location+"06"+"+"+operand[i]);
			}
			else if(i==1) {
				String location=Integer.toHexString(token.location);
				int k=6-location.length();
				for(int a=0;a<k;a++)
location="0".concat(location);
this.mRecord.add(location+"06"+"-"+operand[i]);
			}


objectCode=objectCode.concat("000");
		}
					}
				}
			}

		}

		}//end WORD
		//////////////////////
		else if(token.operator.equals("BYTE")) {
			String llabel[]=token.operand[0].split("'");
if(token.operand[0].charAt(0)=='X') {
token.byteSize=(llabel[1].length()+1)%2;
objectCode=objectCode.concat(llabel[1]);
			}
			else if(token.operand[0].charAt(0)=='C') {
token.byteSize=llabel[1].length();
				for(int k=0;k<token.byteSize;k++)
objectCode=objectCode.concat(Integer.toHexString(llabel[1].charAt(k)));
			}
		}
		else if(token.label.equals("*")) {
			String label=token.operator;
			String llabel[]=label.split("'");
if(label.charAt(1)=='X') {
token.byteSize=(llabel[1].length()+1)%2;

objectCode=objectCode.concat(llabel[1]);
			}
			else if(label.charAt(1)=='C') {
token.byteSize=llabel[1].length();
for(int k=0;k<token.byteSize;k++)

objectCode=objectCode.concat(Integer.toHexString(llabel[1].charAt(k)));
			}

		}
		//////////////////////////////
		/////////////opcode////////////
		//////////////////////////////
		else {
if(this.instTab.containInst(token.operator)) {
			Instruction inst= this.instTab.get(token.operator);
token.byteSize=inst.format;

if(inst.format==1) {
objectCode=Integer.toHexString(inst.opcode);
			}
			else if(inst.format==2) {
objectCode=Integer.toHexString(inst.opcode);
				int i=0;
				try {
	for(i=0;i<2;i++) {
```

```
if(token.operand[i].equals("A"))

objectCode=objectCode.concat("0");

                    else
if(token.operand[i].equals("X"))

objectCode=objectCode.concat("1");

                    else
if(token.operand[i].equals("L"))

objectCode=objectCode.concat("2");

                    else
if(token.operand[i].equals("B"))

objectCode=objectCode.concat("3");

                    else
if(token.operand[i].equals("S"))

objectCode=objectCode.concat("4");

                    else
if(token.operand[i].equals("T"))

objectCode=objectCode.concat("5");

                    else
if(token.operand[i].equals("F"))

objectCode=objectCode.concat("6");

                    else
if(token.operand[i].equals("PC"))

objectCode=objectCode.concat("8");

                    else
if(token.operand[i].equals("SW"))

objectCode=objectCode.concat("9");

          }
                                        }
catch(Exception e) {

          for(int ii=i;ii<2;ii++)

objectCode=objectCode.concat("0");
                                        }

                                  }
                                  else
if(inst.format==3) {
                                        String
disp="";

objectCode=Integer.toHexString(inst.opcode/16
);

objectCode=objectCode.concat(Integer.toHexSt
ring(inst.opcode%16+ token.getFlag(nFlag| iFla
g)/16));

objectCode=objectCode.concat(Integer.toHexSt
ring(token.getFlag(xFlag| bFlag| pFlag| eFlag)));

//////////////////////////////////////////

//////////simple addressing//////////////

//////////////////////////////////////////
                                                try {

          if(token.getFlag(nFlag| iFlag)==48) {

                    //////////////////////

                    ////////literal////////

                    //////////////////////



if(token.operand[0].charAt(0)=='=') {

                              int litAddress;

                              if((litAddress=
this.literalTab.search(token.operand[0]))<0)

System.err.println("non-existent literal :
"+ token.operand[0]);

                              else {

disp=Integer.toHexString(litAddress-(token.loc
ation+ token.byteSize));

if(disp.length()==1)

disp="00".concat(disp);

                                        else
if(disp.length()==2)

disp="0".concat(disp);

                                        else
if(disp.length()>3) {

disp=disp.substring(disp.length()-3,
disp.length());

                                        }

                              }
```

```
                }
                //////////////////////
                else {
                        int symaddress;

if((symaddress=this.symTab.search(token.oper
and[0]))<0)

System.err.println("non-existent symbol :
"+ token.operand[0]);
                                else {

disp=Integer.toHexString(symaddress-(token.l
ocation+ token.byteSize));

if(disp.length()==1)

disp="00".concat(disp);
                                        else
if(disp.length()==2)

disp="0".concat(disp);
                                        else
if(disp.length()>3) {

disp=disp.substring(disp.length()-3,
disp.length());
                                        }
                                }
                }
        }
        ////////////////////////////////////////
        //////////immediate addressing//////////
        ////////////////////////////////////////
        else
if(token.getFlag(nFlag| iFlag)==16) {
                int symaddress;

if(Character.isAlphabetic(token.operand[0].cha
rAt(1))) {

if((symaddress=this.symTab.search(token.oper
and[0]))<0)

System.err.println("non-existent symbol :
"+ token.operand[0]);
                                else {

disp=Integer.toHexString(symaddress-(token.l
ocation+ token.byteSize));

if(disp.length()==1)

disp="00".concat(disp);
                                        else
if(disp.length()==2)

disp="0".concat(disp);
                                        else
if(disp.length()>3) {

disp=disp.substring(disp.length()-3,
disp.length());
                                        }
                                }
                        }
                        else {
                                String
operand[]=token.operand[0].split("#");
                                disp=operand[1];
                                if(disp.length()==1)

disp="00".concat(disp);
                                        else
if(disp.length()==2)

disp="0".concat(disp);
                                        else
if(disp.length()>3) {

disp=disp.substring(disp.length()-3,
disp.length());
                                        }


                        }

        }
        ////////////////////////////////////////
```

```
//////////indirect addressing//////////

//////////////////////////////////////

else
if(token.getFlag(nFlag| iFlag)==32) {

int symaddress;

String
operand[]=token.operand[0].split("@");

if((symaddress=this.symTab.search(operand[1]
))<0)

System.err.println("non-existent symbol :
"+ token.operand[0]);

else {

disp=Integer.toHexString(symaddress-(token.l
ocation+ token.byteSize));

if(disp.length()==1)

disp="00".concat(disp);

else
if(disp.length()==2)

disp="0".concat(disp);

else
if(disp.length()>3) {

disp=disp.substring(disp.length()-3,
disp.length());

}

}

}

}

catch(StringIndexOutOfBoundsException e)
{//if no operand

disp="000";

}

objectCode=objectCode.concat(disp);
}
else
if(inst.format==4) {

String
disp="";

objectCode=Integer.toHexString(inst.opcode/16
);

objectCode=objectCode.concat(Integer.toHexSt
ring(inst.opcode%16+ token.getFlag(nFlag| iFla
g)/16));

objectCode=objectCode.concat(Integer.toHexSt
ring(token.getFlag(xFlag| bFlag| pFlag| eFlag)));

if(this.symTab.search(token.operand[0])<0) {

//EXTREF찾기 구현

String
location=Integer.toHexString(token.location+ 1)
;

int k=6-location.length();

for(int i=0;i<k;i++)

location="0".concat(location);

this.mRecord.add(location+ "05"+ "+ "+ token.op
erand[0]);

objectCode=objectCode.concat("00000");
}

}

}
}//end opcede
//////////////////////////////////


token.objectCode=objectCode.toUpperCase();
}

/**
 * index번호에 해당하는 object code를
리턴한다.
 *
 * @param index
 * @return : object code
 */
public String getObjectCode(int index)
{
return
tokenList.get(index).objectCode;
}

}

/**
 * 각 라인별로 저장된 코드를 단어 단위로
분할한 후 의미를 해석하는 데에 사용되는
변수와 연산을 정의한다. 의미 해석이 끝나면
pass2에서
 * object code로 변형되었을 때의 바이트 코드
역시 저장한다.
 */
class Token {
// 의미 분석 단계에서 사용되는 변수들
int location;
String label;
```

```
	String operator;
	String[] operand;
	String comment;
	char nixbpe;

	// object code 생성 단계에서 사용되는 변수들
	String objectCode;
	int byteSize;

	/**
	 * 클래스를 초기화 하면서 바로 line의 의미 분석을 수행한다.
	 *
	 * @param line 문장단위로 저장된 프로그램 코드
	 */
	public Token(String line) {
		// initialize ???
		parsing(line);
	}

	/**
	 * line의 실질적인 분석을 수행하는 함수. Token의 각 변수에 분석한 결과를 저장한다.
	 *
	 * @param line 문장단위로 저장된 프로그램 코드.
	 */
	public void parsing(String line) {
		String lineToken[]=line.split("₩t");
		if(lineToken[0].equals(".")) {//assembly 주석

this.label=lineToken[0];
			this.operator="";
			if(lineToken.length>1)

comment=lineToken[1];
		}
		else {
			label=lineToken[0];

operator=lineToken[1];
			if(lineToken.length>2) {

operand=lineToken[2].split(",");
			if(operand.length>1)

if(operand[1].equals("X")) {

this.setFlag(TokenTable.xFlag, 1);
				if(operand[0].equals("A")||
operand[0].equals("X")||
operand[0].equals("L")

		||operand[0].equals("B")||
operand[0].equals("S")||
operand[0].equals("T")

		||operand[0].equals("T")||
operand[0].equals("F")||
operand[0].equals("PC")

		||operand[0].equals("SW"))

this.setFlag(TokenTable.xFlag, 0);
			}
		}
		if(lineToken.length>3)

comment=lineToken[3];
		/**	nixbpe	*/

this.setFlag(TokenTable.nFlag,1);

this.setFlag(TokenTable.iFlag, 1);
		try {

if(operand[0].charAt(0)=='@') {

this.setFlag(TokenTable.iFlag, 0);
			}
			else
if(operand[0].charAt(0)=='#') {

this.setFlag(TokenTable.nFlag, 0);
			}
		}
		catch(Exception e) {

		}

		try {

if(this.getFlag(TokenTable.nFlag)==TokenTable.nFlag) {//n=1
				try{

	if(operator.charAt(0)=='+') {

		this.setFlag(TokenTable.eFlag, 1);

	}

	else if(!operand[0].equals(""))

	this.setFlag(TokenTable.pFlag, 1);
				}

catch(Exception e) {

				}

			}
			else
if(this.getFlag(TokenTable.iFlag)==TokenTable.iFlag){//n=0,i=1

if(Character.isAlphabetic(operand[0].charAt(1))) {

	this.setFlag(TokenTable.pFlag, 1);
				}
			}
		}
		catch(Exception e) {

		}

	}//end not comment line

	}
```

```
	/**
	 * n,i,x,b,p,e flag를 설정한다.
	 *
	 *
	 * 사용 예 : setFlag(nFlag, 1) 또는 setFlag(TokenTable.nFlag, 1)
	 *
	 * @param flag  : 원하는 비트 위치
	 * @param value : 집어넣고자 하는 값. 1또는 0으로 선언한다.
	 */
	public void setFlag(int flag, int value)
{
		// ...
		if(flag==TokenTable.nFlag)
{//n
			if (value==1)
this.nixbpe=(char)(nixbpe|flag);
			else
nixbpe=(char)(nixbpe&~flag);
		}
		else
if(flag==TokenTable.iFlag) {//i
			if (value==1)
nixbpe=(char)(nixbpe|flag);
			else
nixbpe=(char)(nixbpe&~flag);
		}
		else
if(flag==TokenTable.xFlag) {//x
			if (value==1)
this.nixbpe=(char)(nixbpe|flag);
			else
nixbpe=(char)(nixbpe&~flag);
		}
		else
if(flag==TokenTable.bFlag) {//b
			if (value==1)
this.nixbpe=(char)(nixbpe|flag);
			else
nixbpe=(char)(nixbpe&~flag);
		}
		else
if(flag==TokenTable.pFlag) {//p
			if (value==1)
this.nixbpe=(char)(nixbpe|flag);
			else
nixbpe=(char)(nixbpe&~flag);
		}
		else
if(flag==TokenTable.eFlag) {//e
			if (value==1)
this.nixbpe=(char)(nixbpe|flag);
			else
nixbpe=(char)(nixbpe&~flag);
		}

	}

	/**
	 * 원하는 flag들의 값을 얻어올 수 있다. flag의 조합을 통해 동시에 여러개의 플래그를 얻는 것 역시 가능하다.
	 *
	 * 사용 예 : getFlag(nFlag) 또는 getFlag(nFlag|iFlag)
	 *
	 * @param flags : 값을 확인하고자 하는 비트 위치
	 * @return : 비트위치에 들어가 있는 값. 플래그별로 각각 32, 16, 8, 4, 2, 1의 값을 리턴할 것임.
	 */

	public int getFlag(int flags) {
		return nixbpe & flags;
	}
}
```