

설계(프로젝트) 보고서

나는 송실대학교 컴퓨터학부의 일원으로 명예를 지키면서 생활하고 있습니다.
나는 보고서를 작성하면서 다음과 같은 사항을 준수하였음을 엄숙히 서약합니다.

1. 나는 자력으로 보고서를 작성하였습니다.
2. 나는 보고서에서 참조한 문헌의 출처를 밝혔으며 표절하지 않았습니다.
3. 나는 보고서의 내용을 조작하거나 날조하지 않았습니다.

교과목	시스템프로그래밍 2020
프로젝트 명	JAVA언어로 파서 구현하기
교과목 교수	최 재 영
제출인	전자정보공학부 학번: 20160433 성명: 김민정 (출석번호: 108)
제출일	2020년 5월 13 일

차 례

1장 프로젝트 개요

1.1 개발 배경 및 목적

2장 배경 지식

2.1 주제에 관한 배경지식

2.2 기술적 배경지식

3장 시스템 설계 내용

3.1 전체 시스템 설계 내용

3.2 모듈별 설계 내용

4장 시스템 구현 내용 (구현 화면 포함)

4.1 전체 시스템 구현 내용

4.2 모듈별 구현 내용

4.3 기존에 생성해놓은 inst.data 파일 내용

4.4 디버깅 과정

4.5 수행결과

5장 기대효과 및 결론

첨부 프로그램 소스파일

1장 프로젝트 개요

1.1 개발 배경 및 목적

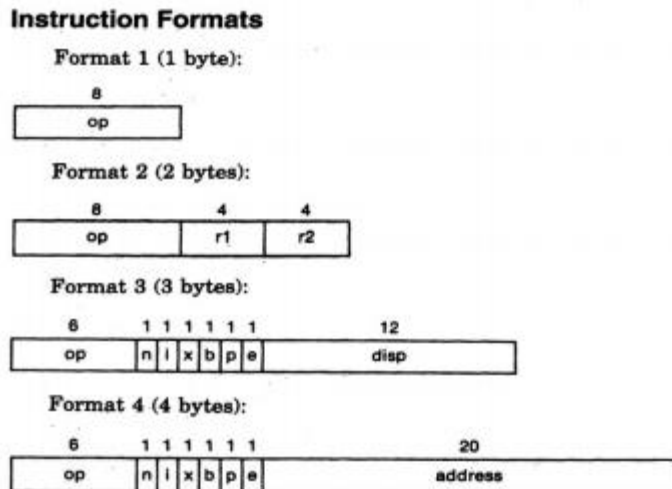
- 주어진 input파일을 이용하여 기본적인 SIC/XE 어셈블러를 구현해보는 프로젝트를 진행해보았다. 이번 프로젝트는 지난번 프로젝트처럼 input 파일을 원하는 object program으로 작성해보는 프로젝트이지만 구현언어가 c에서 java로 바뀌었다. input파일을 라인별로 읽어들이고 후 각각의 명령어들을 원하는 대로 파싱한 후 파싱한 토큰들을 각각 분석하여 어셈블러가 어떠한 규칙으로 기계어들을 변환하는지를 이해할 수 있다. 또한 object program을 작성하기 위해 input파일의 프로그램을 변환해보면서 그 과정들에 대해 좀 더 명확한 이해가 가능하다. 이를 통해 시스템프로그래밍의 2장 Assemblers를 더 확실하게 공부할 수 있다. 또한 C로 한번 설계해보았던 프로젝트를 JAVA로 다시 프로그래밍해보면서 다시 한번 프로그램에 대해 이해해볼 수 있었고 JAVA에 대해서도 다시한 번 공부할 수 있다.

2장 배경 지식

2.1 주제에 관한 배경지식

1) opcode와 명령어 형식

- SIC/XE의 머신은 4종류의 형식을 지원한다.



이 때 비트 e의 값으로 3형식과 4형식을 구분한다. e=0이면 3형식을 따르고 e=1이면 4형식을 따른다.

2) 주소지정방식

- 주소지정방식에는 Absolute addressing, Indirect addressing, Immediate addressing, simple addressing, relative addressing이 존재한다.

3) assembler program

- pass1과 pass2의 방식으로 이루어지는데 pass1에서는 각 라인별로 loc를 지정하고 symbol_table과 literal_table을 작성한다. 그 후 pass2에서는 pass1의 결과물을 이용하여 각 라인별로 object code를 작성하고 이 object code를 이용하여 추후 object program을 만들어낸다.

4) object program format

- object program은 Header record, Define record, Refer record, Text record, Modification record, End record의 형식으로 되어 있는데 각각의 record들의 형식은 다음과 같

다.

□ Header record

- ◆ Col. 1 H
- ◆ Col. 2~7 Program name
- ◆ Col. 8~13 Starting address of object program (Hex)
- ◆ Col. 14~19 Length of object program in bytes (Hex)

◆ Define record

- Col. 1 D
- Col. 2-7 Defined external symbol name
- Col. 8-13 Relative address of symbol within this control section
- Col. 14-73 Repeat information in Col. 2-13 for other external symbols

◆ Refer record

- Col. 1 R
- Col. 2-7 Referred external symbol name
- Col. 8-73 Names of other external reference symbols

□ Text record

- ◆ Col. 1 T
- ◆ Col. 2~7 Starting address for object code in this record (Hex)
- ◆ Col. 8~9 Length of object code in this record in bytes (Hex)
- ◆ Col. 10~69 Object code in Hex (2 column per byte)

◆ Modification record (revised)

- Col. 1 M
- Col. 2-7 Starting address of the field to be modified
- Col. 8-9 Length of the field to be modified
- Col. 10 Modification flag (+ or -)
- Col. 11-16 External symbol whose value is to be added to or subtracted from the indicated field

□ End record

- ◆ Col. 1 E
- ◆ Col. 2~7 Address of first executable instruction in object (hex)

2.2 기술적 배경지식

1) 파일입출력

자바에서 입출력과 관련된 클래스들이 모여있는 패키지는 java.io 패키지이다. 따라서 입출력 프로그램을 작성하려면 java.io 패키지를 import 시켜준 다음 프로그램을 작성해야 한다. 파일을 읽고 쓰는 방법에는 다음과 같이 여러 방법이 존재한다. 먼저 파일을 읽는 방법이다. FileReader를 이용하여 파일을 읽을 수 있고 BufferedReader를 이용하여 파일을 읽을 수 있다. 그리고 Scanner를 이용하여 파일을 읽을 수 있고 DataInputStream을 이용하여 파일을 읽을 수 있다. 마찬가지로 FileWriter를 이용하여 파일에 데이터를 쓸 수 있고 BufferedWriter를 이용하여 파일에 데이터를 쓸 수 있다. 그리고 PrintWriter를 이용하여 파일에 데이터를 쓸 수 있고 FileOutputStream을 이용하여 파일에 데이터를 쓸 수 있다. 또한 자바에서 File I/O를 하기 위해서는 try-catch 구문을 사용해야 한다.

2) 토큰분리

문자열들을 가공을 하기 위해서는 원하는 문자열만 잘라 보관한 후에 사용하는 경우가 많다. 이번 프로젝트에서도 필수적으로 사용해야 했던 기술이 토큰분리인데 이를 위해 StringTokenizer 클래스와 String 클래스의 split()를 사용하였다. StringTokenizer는 특정 구분자를 기준으로 token 단위로 끊어서 읽을 수 있게 해주고 split()는 구분자를 기준으로 문자열을 분리하여 배열로 리턴해주는 함수이다. StringTokenizer 클래스의 생성자와 주요 메소드는 다음과 같다.

StringTokenizer(String str)	delimiter를 인자로 받지 않는 생성자로 디폴트 구획문자는 공백문자들이다.
StringTokenizer(String str,String delim)	delimiter를 인자로 받는 생성자이다. 또한 delimiter는 2자리 이상으로 설정하여 여러개의 delimiter를 가질 수 있다.
StringTokenizer(String str), String delim, boolean returnDelims)	delimiter를 인자로 받는 생성자로 returnDelims가 true면 구획문자도 토큰으로 간주된다.
int countTokens()	현재 남아있는 token의 개수를 반환한다.
boolean hasMoreTokens()	토큰이 더 남아있는지를 확인한다.
String nextToken()	다음 토큰을 불러온다.

3) 비트연산

- 비트연산자에는 다음과 같은 연산자들이 있다.

&	비트단위 AND 연산
	비트단위 OR 연산
^	비트단위 XOR 연산
~	비트단위 NOT 연산
>>	피연산자의 비트열을 오른쪽으로 이동
<<	피연산자의 비트열을 왼쪽으로 이동

연산자 '<<'와 '>>'의 경우 시프트연산자라고도 하는데 이 연산자들의 사용방법은 다음과 같다. 'a << 2'와 같이 사용한다면 이 연산의 의미는 다음과 같다. a의 비트열을 왼쪽으로 2칸 이동하라는 의미이다. 만일 'b >> 5'와 같이 사용했다면 이 연산의 의미는 b의 비트열을 오른쪽으로 5칸 이동하라는 의미이다.

4) 서식지정자를 활용한 입출력

- 다음과 같은 서식지정자들을 활용해서 입출력을 좀 더 편하게 할 수 있다. 자바에서는 System.out.printf(), String.format()등을 사용하여 서식지정자를 활용할 수 있다.

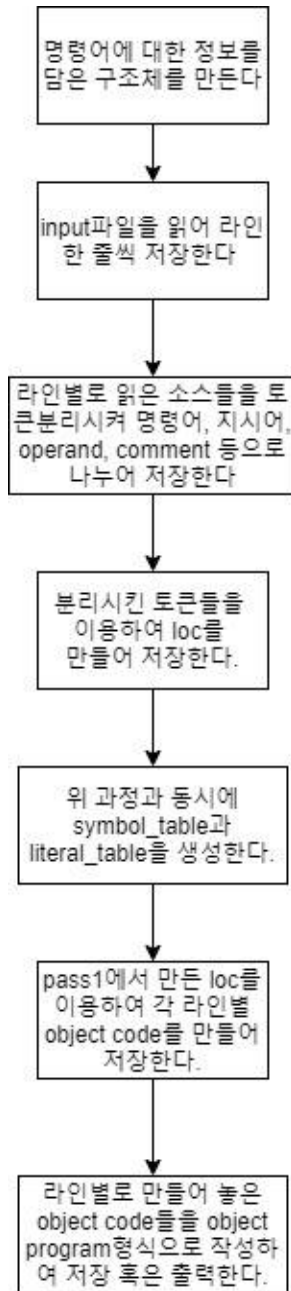
서식 지정자	설명	플래그	설명
d,i	부호 있는 10진 정수	-	왼쪽 정렬
u	부호 없는 10진 정수	공백	양수일 때는 공백, 음수일 때는 -
s	문자열	0	출력하는 폭의 남은 공간에 0으로 채움
X	부호 없는 16진 정수(대문자)		

5) JAVA

- JAVA란 자바로 지술된 프로그램 개발 및 실행을 할 수 있는 소프트웨어 모임의 총칭이다. 자바 프로그램은 운영체제나 하드웨어에 의존하지 않는 바이트코드인 추상적인 코드로 구현된다. 따라서, 자바 프로그램을 실행하기 위해서는 자바 가상머신(JVM)과 개발에 필요한 표준 라이브러리 세트와 컴파일러 환경만 맞추면 자바 프로그램은 모든 환경에서 동일하게 동작한다. 이러한 실행환경과 개발환경을 제공하는 것이 자바 플랫폼이다. JAVA는 처음부터 객체 지향 언어로 개발된 프로그래밍언어로 운영체제와는 독립적으로 실행할 수 있다. 또한 자동으로 메모리를 관리해주기 때문에 다른 언어에 비해 안정성이 높고 연산자 오버로딩을 금지하고 제네릭을 도입하여 코드의 가독성이 좋다. 그러나 자바는 자바 가상머신(JVM)에 의해 실행되기 때문에 다른 언어에 비해 실행속도가 느리고, 개발자가 일일이 예외처리를 지정해줘야한다는 불편함이 있는 언어이다.

3장 시스템 설계 내용

3.1 전체 시스템 설계 내용



3.2 모듈별 설계 내용

-기본적으로 주어진 메소드들을 이용하여 프로그램을 구현하였으나 필요에 의해 여러 메소드와 클래스변수들을 추가하였다.

1) InstTable.java

* 클래스 : InstTable

- 모든 instruction의 정보를 관리하는 클래스로 instruction data들을 저장한다. 또한 instruction 관련연산, 관련정보를 제공하는 함수 등을 제공한다.

① void openFile(String fileName)

- 입력받은 이름의 파일을 열고 해당 내용을 파싱하여 instMap에 저장하는 메소드이다.

② int getOpcode(String instruction)

- 인자로 받은 명령어의 opcode를 찾아 리턴하는 메소드이다.

③ int getNumberOfOperand(String instruction)

- 인자로 받은 명령어의 피연산자 개수를 찾아 리턴하는 메소드이다.

④ int getFormat(String instruction)

- 인자로 받은 명령어의 형식을 찾아 리턴하는 메소드이다.

⑤ int isInstruction(String str)

- 입력문자열이 명령어인지 검사하는 메소드이다.

⑥ int isDirective(String str)

- 입력문자열이 지시어인지 검사하는 메소드이다.

* 클래스 : Instruction

- 각각의 명령어에 대한 구체적인 정보는 Instruction클래스에 담긴다. 이 클래스에서는 instruction과 관련된 정보를 저장하고 기초적인 연산을 수행한다.

① void parsing(String line)

- 일반 문자열을 파싱하여 instruction 정보를 파악하고 저장하는 메소드이다.

2) TokenTable.java

* 클래스 : TokenTable

- 사용자가 작성한 프로그램 코드를 토큰별로 분할한 후 각 토큰의 의미를 분석하여 최종 object code로 변환하는 과정을 총괄하는 클래스이다. section마다 인스턴스가 하나씩 할당된다.

① void putToken(String line)

- 일반 문자열을 받아서 Token단위로 분리시켜 tokenList에 추가하는 메소드이다.

② Token getToken(int index)

- tokenList에서 index에 해당하는 Token을 리턴하는 메소드이다.

③ void makeObjectCode(int index)

- Pass2 과정에서 사용하는 메소드로 instruction table, symbol table, literal table 등을 참조하여 object code를 생성하고, 이를 저장한다.

④ String getObjectCode(int index)

- index번호에 해당하는 object code를 리턴하는 메소드이다.

* 클래스 : Token

- 각 라인별로 저장된 코드를 단어 단위로 분할한 후 의미를 해석하기 위해 사용되는 변수와 연산을 정의하는 클래스이다.

① void setLocation(int loc)

- loc값을 인자로 받아와 저장하는 메소드이다.

② void setObjectCode(String objcode)

- object code을 인자로 받아와 저장하는 메소드이다.

③ void setByteSize(int bSize)

- byteSize을 인자로 받아와 저장하는 메소드이다.

④ void parsing(String line)

- line의 실질적인 분석을 수행하는 메소드로 Token의 각 변수에 분석한 결과를 저장한다.

⑤ void setFlag(int flag, int value)

- n,i,x,b,p,e flag를 설정해주는 메소드이다.

⑥ int getFlag(int flags)

- 원하는 flag들의 값을 얻어오는 메소드이다. flag의 조합을 통해 동시에 여러개의 플래그를 얻을 수 있다.

3) SymbolTable.java

* 클래스 : SymbolTable

- symbol과 관련된 데이터와 연산을 가지고 있는 클래스로 section별로 하나씩 인스턴스를 할당한다.

① void putSymbol(String symbol, int location)

- 새로운 Symbol을 table에 추가하는 메소드이다.

② void modifySymbol(String symbol, int newLocation)

- 기존에 존재하는 symbol 값에 대해서 가리키는 주소값을 변경해주는 메소드이다.

③ void modSymbol(String symbol, int location, int size)

- modification record에 작성할 symbol들을 table에 추가해주는 메소드이다.

④ void setdefSymbol(String symbol)

- 지시어 EXTDEF가 나왔을 때 symbol을 저장하는 메소드이다.

⑤ void setrefSymbol(String symbol)

- 지시어 EXTREF가 나왔을 때 symbol을 저장하는 메소드이다.

⑥ int searchdefSymbol(String symbol)

- 주어진 symbol이 definition된 symbol인지 확인하는 메소드이다.

⑦ int searchrefSymbol(String symbol)

- 주어진 symbol이 reference된 symbol인지 확인하는 메소드이다.

⑧ int search(String symbol)

- 인자로 전달된 symbol이 어떤 주소를 지칭하는지 알려주는 메소드이다.

⑨ String getSymbol(int index)

- index에 해당하는 symbol을 리턴해주는 메소드이다.

⑩ int getAddress(int index)

- index를 이용하여 symbol을 찾은 후 그 symbol에 해당하는 address를 리턴한다.

⑪ int getSize()

- symboltable의 크기를 리턴해주는 메소드이다.

4) LiteralTable.java

* 클래스 : LiteralTable

- literal과 관련된 데이터와 연산을 소유하는 메소드로 section별로 하나씩 인스턴스를 할당한다.

① void putLiteral(String literal, int location)

- 새로운 Literal을 table에 추가해주는 메소드이다.

② void modifyLiteral(String literal, int newLocation)

- 기존에 존재하는 literal 값에 대해서 가리키는 주소값을 변경해주는

메소드이다.

③ int search(String literal)

- 인자로 전달된 literal이 어떤 주소를 지칭하는지 알려주는 메소드이다.

④ int addrLiteralTab(int locctr)

- locctr를 이용하여 literal들의 주소를 넣어주는 메소드이다.

⑤ String getLiteral(int index)

- index에 해당하는 literal을 리턴해주는 메소드이다.

⑥ int getaddress(int index)

- index를 이용하여 literal을 찾은 후 해당하는 literal의 address를 리

턴해주는 메소드이다.

⑦ int getSize()

- literaltable의 크기를 리턴해주는 메소드이다.

5) Assembler.java

* 클래스 : Assembler

- SIC/XE 머신을 위한 Assembler 프로그램의 메인 루틴에 해당하는 클래스이다.

① void loadInputFile(String inputFile)

- inputFile을 읽어들여서 lineList에 저장하는 메소드이다.

② void pass1()

- pass1 과정을 수행하는 메소드이다.

③ void printSymbolTable(String fileName)

- 작성된 SymbolTable들을 출력형태에 맞게 출력해주는 메소드이다.

④ void printLiteralTable(String fileName)

- 작성된 LiteralTable들을 출력형태에 맞게 출력해주는 메소드이다.

⑤ void pass2()

- pass2 과정을 수행하는 메소드이다.

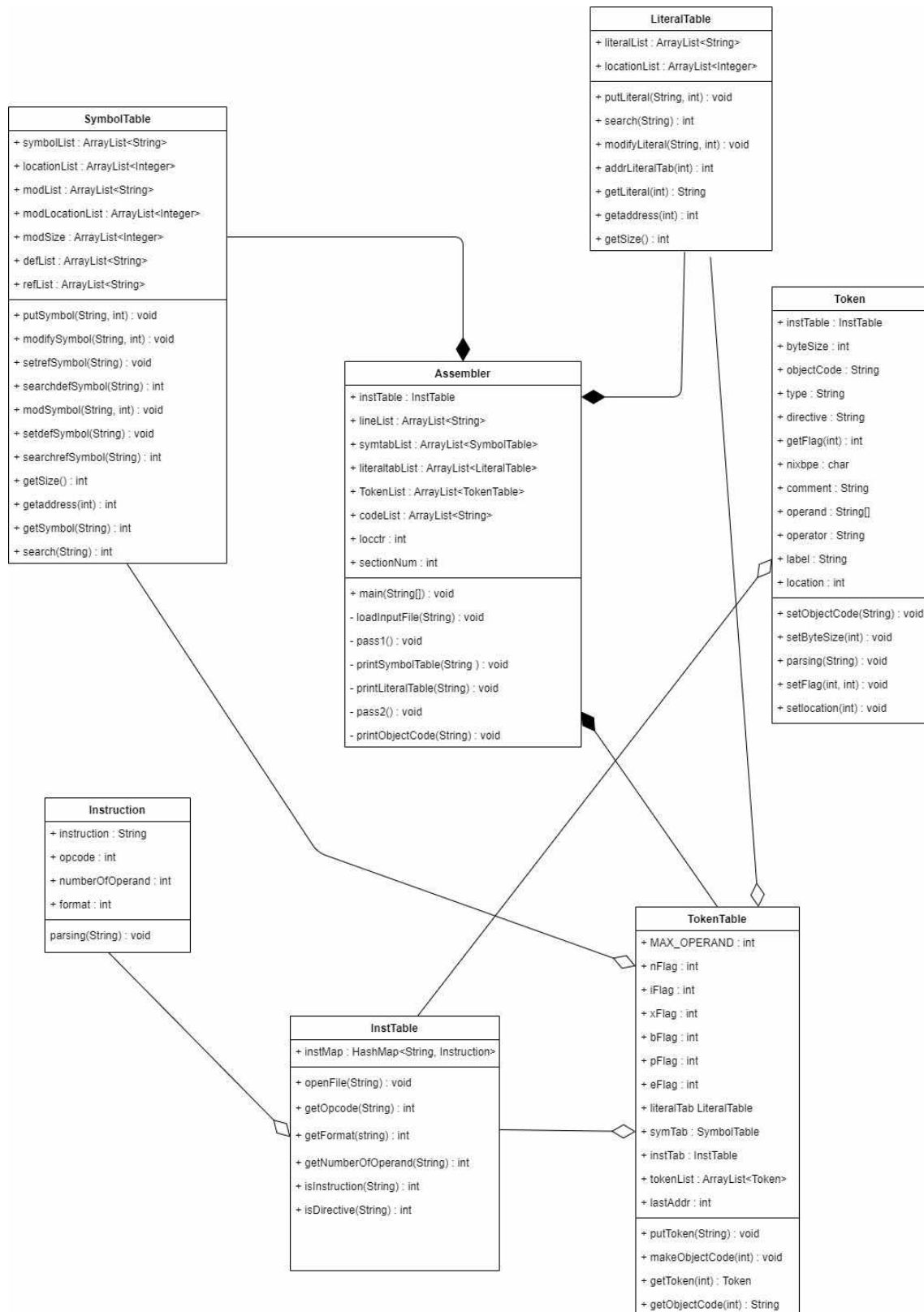
⑥ void printObjectCode(String fileName)

- 이전 과정들에서 작성된 codeList를 출력형태에 맞게 출력해주는

메소드이다.

4장 시스템 구현 내용 (구현 화면 포함)

4.1 전체 시스템 구현 내용



4.2 모듈별 구현 내용

1) InstTable.java

- InstTable 클래스에 HashMap을 이용하여 명령어들에 대한 정보를 저장하였다. 명령어에 대한 정보를 담은 파일을 기존에 만들어놓고 그 파일을 한 줄씩 읽어들었다. 그런다음 Instruction 클래스에서 명령어들에 대한 정보를 각각 토큰분리하여 각 정보에 해당하는 변수들에 저장하였다.

2) TokenTable.java

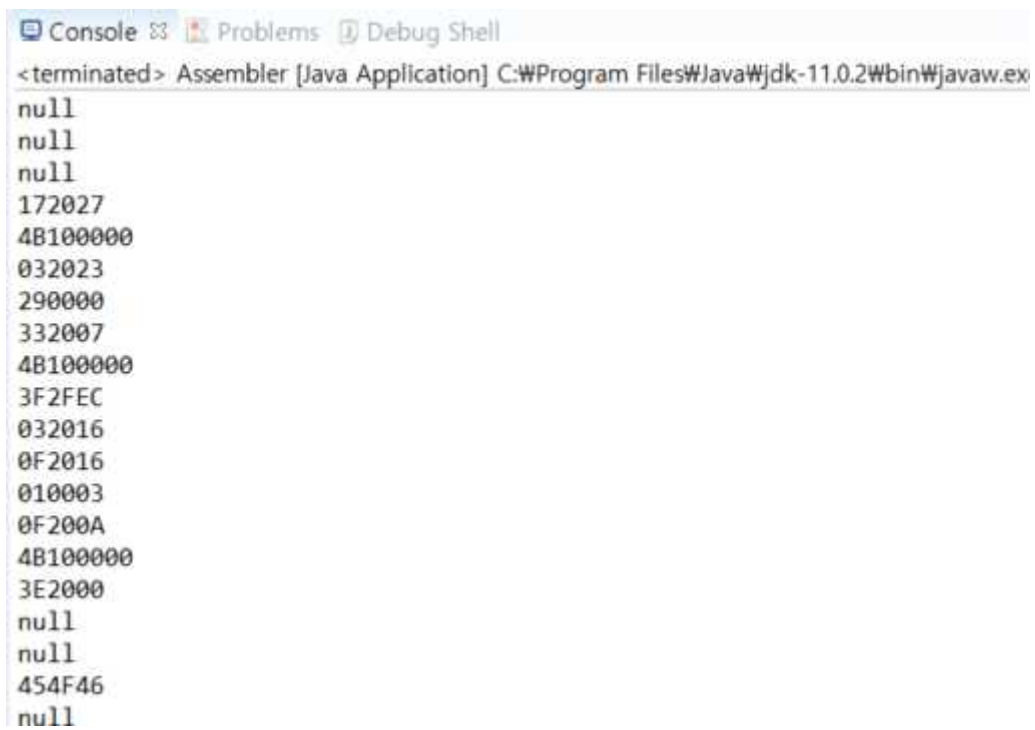
- TokenTable을 초기화하면서 literalTable과 instTable을 링크시켰다. Assembler클래스의 pass1과정을 진행하면서 한줄씩 코드를 읽어온다. 그리고 각 줄마다 Token 클래스의 인스턴스를 생성한 후 의미를 분석하며 토큰분리를 진행시켰다. pass1 과정이 끝난 다음 pass2 과정을 진행하면서 TokenTable클래스에서는 각 줄에 해당하는 object code를 만들어 Token 클래스에 object Code과 byteSize를 저장하였다.

3) Assembler.java

- 메인 루틴이 진행되는 부분이다. Assembler 클래스에서 클래스를 초기화하면서 필요한 여러 인스턴스를 생성한다. 그리고 메인루틴을 진행하는데 메인루틴은 크게 pass1과 pass2로 나뉘어진다. pass1을 진행한 후에는 input파일이 의미에 맞게 토큰분리되어 저장이 되고 각 라인에 맞는 loc가 할당되며 symbol table과 literal table이 만들어진다. 그 후 이 정보들을 이용하여 pass2를 진행하며 각 라인에 맞는 object code를 만든다. 그리고 최종적으로 각 라인에 맞는 object code들을 object program의 형식에 따라 출력하여 저장하였다.



그림 7 pass1을 진행한 후 생성된 loc 중간출력



The screenshot shows a Java IDE's console window with the title bar "Console Problems Debug Shell". The text in the console is as follows:

```
<terminated> Assembler [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\javaw.exe
null
null
null
172027
4B100000
032023
290000
332007
4B100000
3F2FEC
032016
0F2016
010003
0F200A
4B100000
3E2000
null
null
454F46
null
```

그림 8 pass2를 진행하면서 생성된 각 라인들의 object code 중간출력

4.3 기존에 생성해놓은 inst.data 파일 내용

각 명령어들마다 명령어이름, operand의 개수, format, opcode의 순서로 appendix를 참고하여 inst.data파일을 생성하였다. 파일의 내용은 다음과 같다.

inst.data	input.txt	my_assembler_20160433.c
1	ADD 1 3	18
2	ADDF 1 3	58
3	ADDR 2 2	90
4	AND 1 3	40
5	CLEAR 1 2	B4
6	COMP 1 3	28
7	COMPF 1 3	88
8	COMPR 2 2	A0
9	DIV 1 3	24
10	DIVF 1 3	64
11	DIVR 2 2	9C
12	FIX 0 1	C4
13	FLOAT 0 1	C0
14	HIO 0 1	F4
15	J 1 3	3C
16	JEQ 1 3	30
17	JGT 1 3	34
18	JLT 1 3	38
19	JSUB 1 3	48
20	LDA 1 3	00
21	LDB 1 3	68
22	LDCH 1 3	50
23	LDF 1 3	70
24	LDL 1 3	08
25	LDS 1 3	6C
26	LDT 1 3	74
27	LDX 1 3	04
28	LPS 1 3	D0
29	MUL 1 3	20
30	MULF 1 3	60
31	MULR 2 2	98
32	NORM 0 1	C8
33	OR 1 3	44

그림 9 inst.data의 내용(1)

inst.data	input.txt	my_assembler_20160433.c
33	OR 1 3	44
34	RD 1 3	D8
35	RMO 2 2	AC
36	RSUB 0 3	4C
37	SHIFTL 2 2	A4
38	SHIFTR 2 2	A8
39	SIO 0 1	F0
40	SSK 1 3	EC
41	STA 1 3	0C
42	STB 1 3	78
43	STCH 1 3	54
44	STF 1 3	80
45	STI 1 3	D4
46	STL 1 3	14
47	STS 1 3	7C
48	STSW 1 3	E8
49	STT 1 3	84
50	STX 1 3	10
51	SUB 1 3	1C
52	SUBF 1 3	5C
53	SUBR 2 2	94
54	SVC 1 2	B0
55	TD 1 3	E0
56	TIO 0 1	F8
57	TIX 1 3	2C
58	TIXR 1 2	B8
59	WD 1 3	DC
60		

그림 10 inst.data의 내용(2)

4.4 디버깅 과정

-프로그램을 구현하면서 디버깅했던 내용들 중 일부화면들을 첨부하였다.

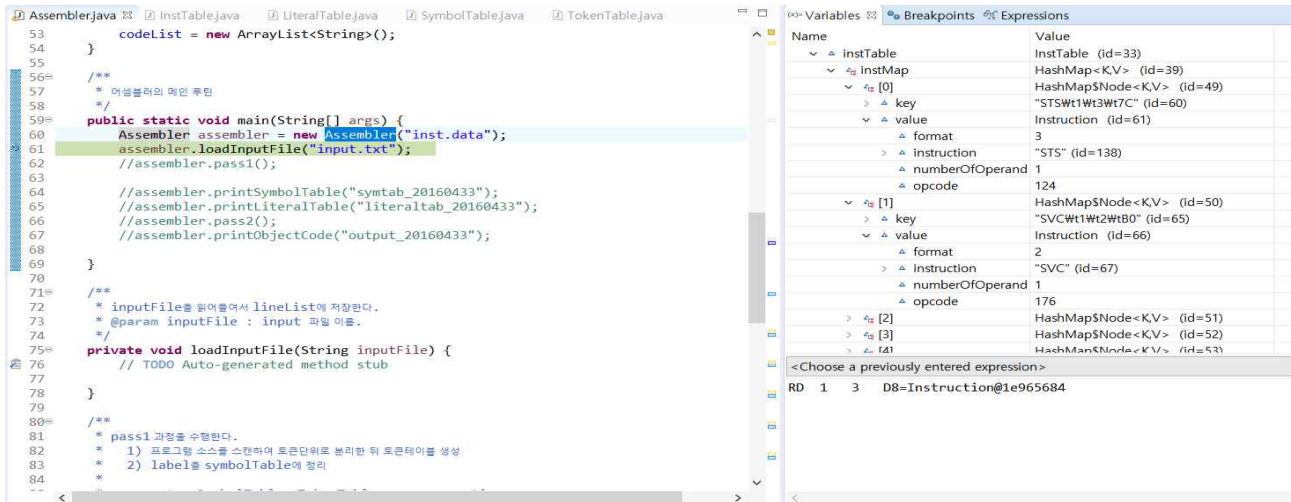


그림 11 기존에 만들어 놓은 inst.data 파일의 내용을 읽어와 inst_table에 저장하는 과정 디버깅

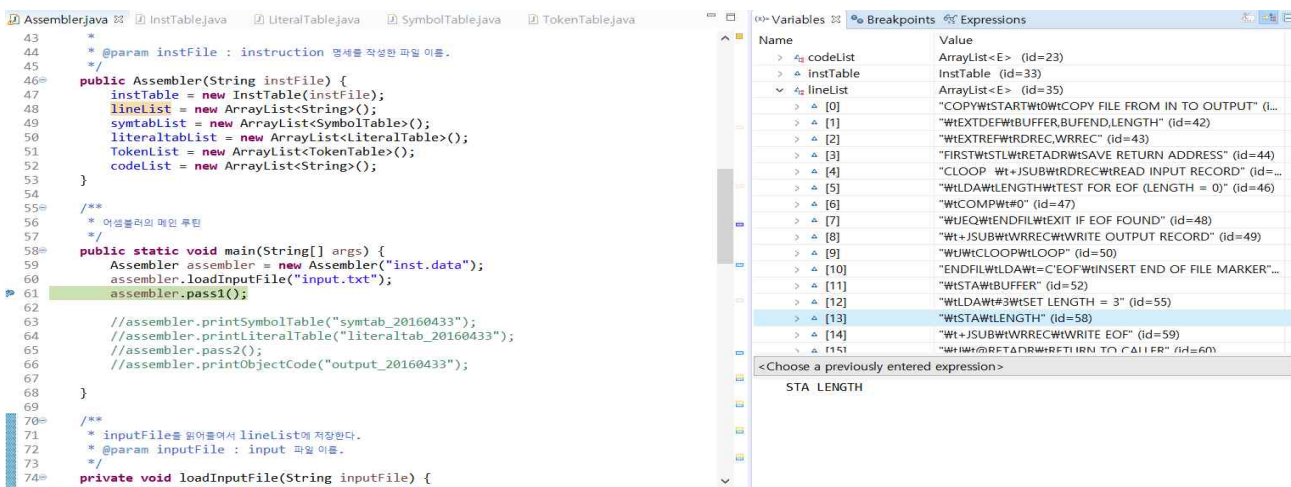


그림 12 input file의 내용을 라인단위로 저장하는 과정 디버깅

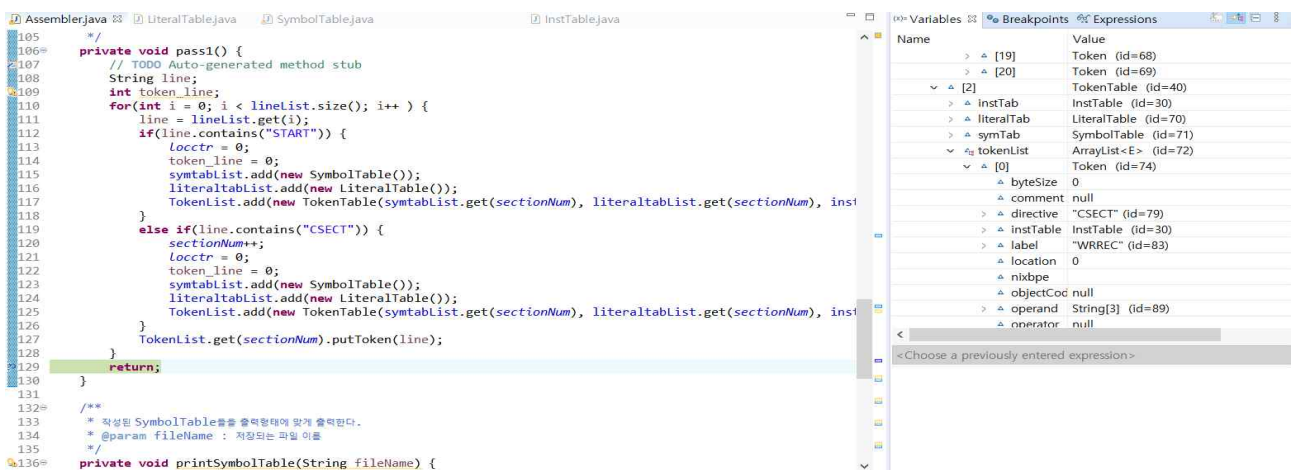


그림 13 소스 코드를 읽어와 토큰단위로 분리하여 토큰 테이블을 작성하는 과정 디버깅

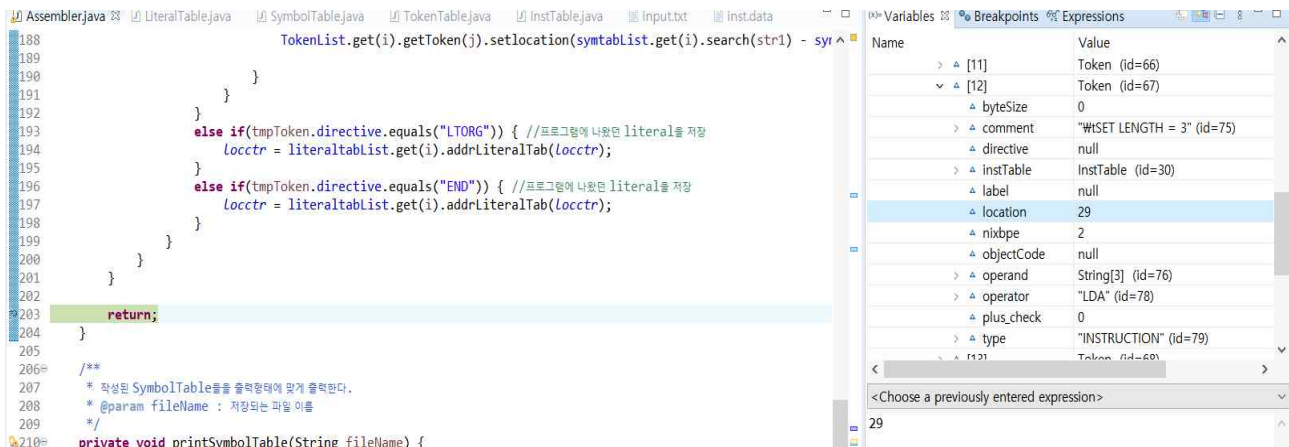


그림 14 loc가 제대로 저장되었는지 디버깅

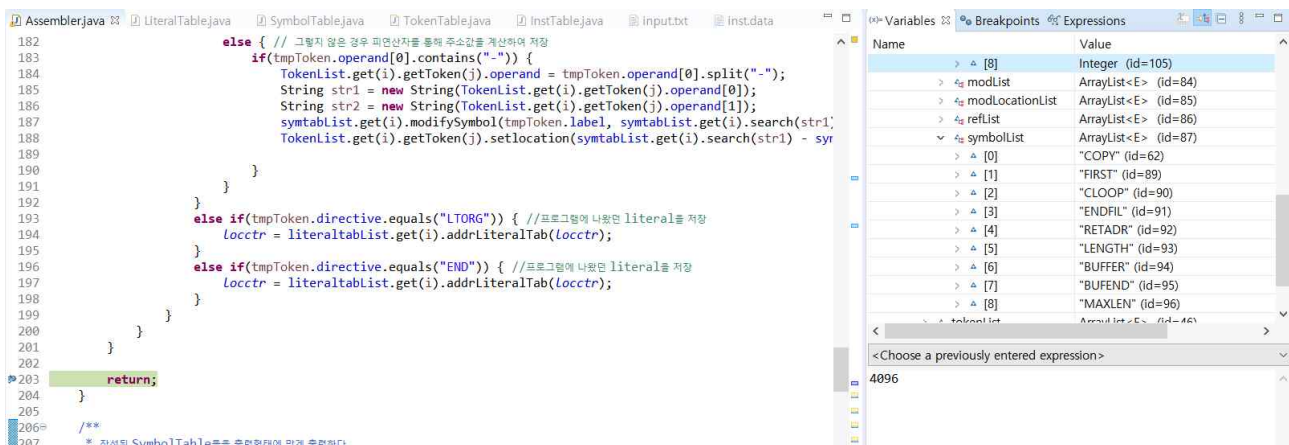


그림 15 symbol_table이 제대로 작성되었는지 디버깅

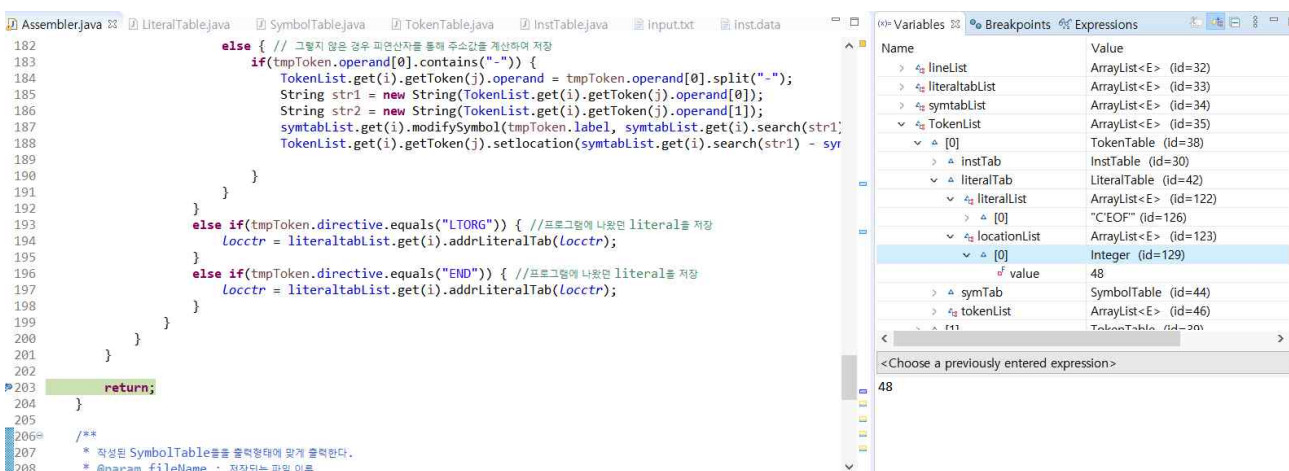


그림 16 literal_table이 제대로 작성되었는지 디버깅



그림 17 object code 만드는 과정 디버깅

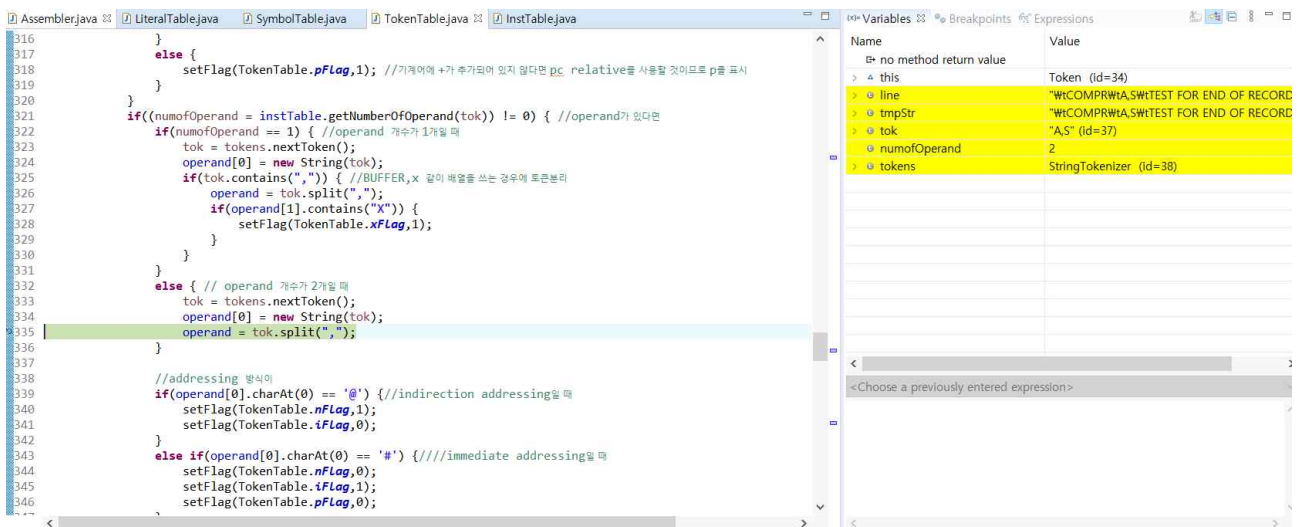


그림 18 object code 만드는 과정 디버깅

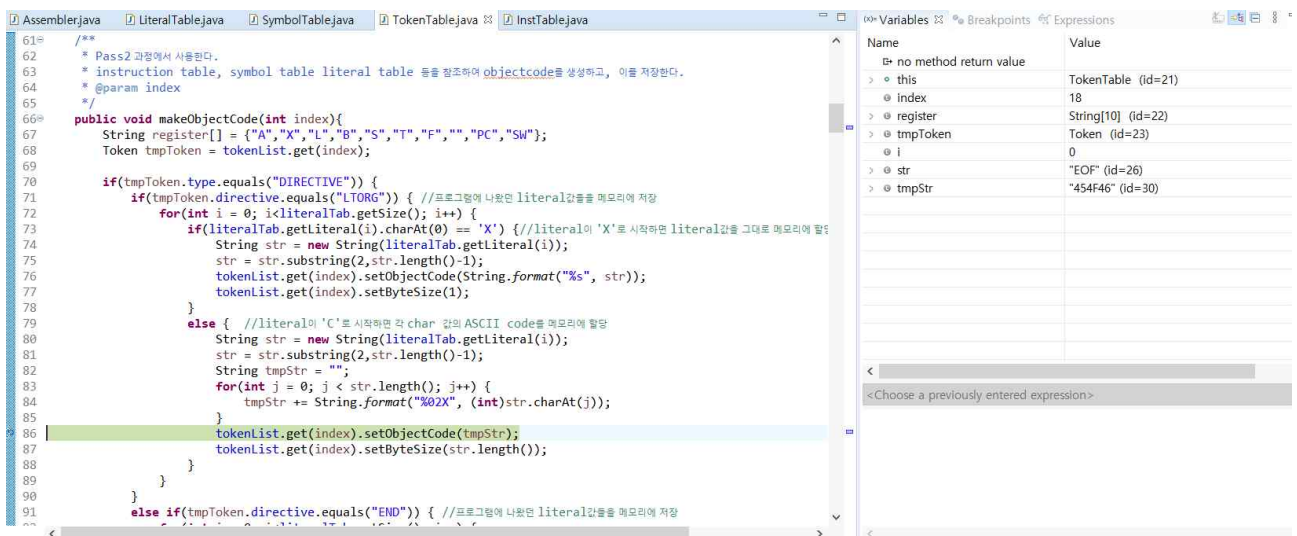


그림 19 object code 만드는 과정 디버깅

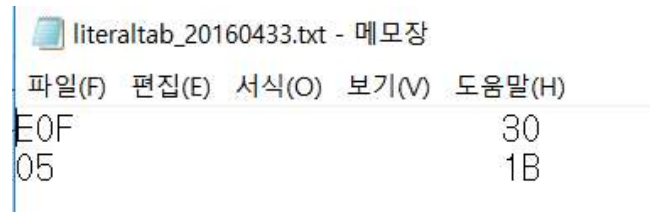
4.5 수행 결과

1) symtab_20160433.txt의 출력결과



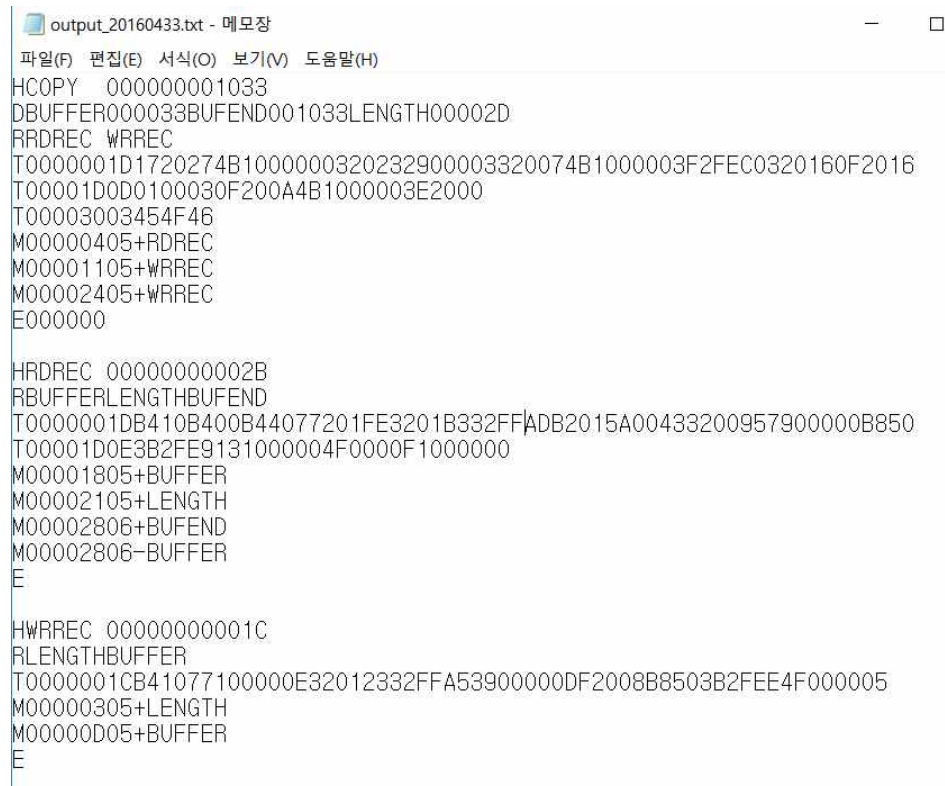
파일(F)	편집(E)	서식(O)	보기(V)	도움말(H)
COPY				0
FIRST				0
LOOP				3
ENDFIL				17
RETADR				2A
LENGTH				2D
BUFFER				33
BUFEND				1033
MAXLEN				1000
RDREC				0
LOOP				9
EXIT				20
INPUT				27
MAXLEN				28
WRREC				0
LOOP				6

2) literal_20160433.txt의 출력결과



파일(F)	편집(E)	서식(O)	보기(V)	도움말(H)
E0F				30
05				1B

3) output_20160433.txt의 출력결과



```
output_20160433.txt - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
HCOPY 000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
T0000001D1720274B1000000320232900003320074B1000003F2FEC0320160F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000
HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER
E
HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB41077100000E32012332FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E
```

5장 기대효과 및 결론

-이번 프로젝트는 Java를 이용하여 주어진 input파일을 파싱하여 토큰화하고 기본적인 SIC/XE 머신의 어셈블러를 구현해보는 프로젝트였다. Java는 익숙치 않은 언어라 프로젝트를 진행하면서 조금 어려움을 겪었지만 C로 한번 구현해본 프로젝트여서 그런지 좀 더 수월하게 프로젝트를 진행할 수 있었다. 지난번 프로젝트와 마찬가지로 이번 프로젝트에서도 input파일을 어떻게 토큰화할지, 토큰분리한 토큰들을 어떻게 활용하여 object code를 작성할지가 중요했다. C에서 한번 짜본 로직이라도 하더라도 java의 특성과 주어진 파일들을 이용해야 했기 때문에 좀 더 고민이 필요했던 부분이었다.

이번 프로젝트에서도 COPY프로그램을 변환하는 것이 목표였기 때문에 모든 문법들이나 모든 기능들이 포함되어 있는 어셈블러를 구현하지는 못했다는 것이 조금 아쉽기는 하지만 추후 이를 보완하여 더 나은 어셈블러를 구현해보려고 한다. 마찬가지로 이번 프로젝트 또한 assembler에 대한 이해를 높이고 java와 객체지향 프로그래밍에 대해서도 좀 더 익숙해질 수 있었던 프로젝트였다.

첨부 프로그램 소스파일

1) InstTable.java

```
import java.util.HashMap;
import java.io.*;
import java.util.StringTokenizer;

/**
 * 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 : instruction에 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) {
        try {
            // 파일 객체 생성
            File file = new File(fileName);
            // 입력 스트림 생성
            FileReader fileReader = new FileReader(file);
            // 입력 버퍼 생성
            BufferedReader bufReader = new
            BufferedReader(fileReader);

            String line = "";
            String inst;
            StringTokenizer tokens;

            while((line = bufReader.readLine()) != null){ //파
            일을 끝까지 읽기 전까지 한줄씩 읽음
                tokens = new StringTokenizer(line); // 읽
                어 들인 라인을 공백 기준으로 한 번 분리하여 instruction을 저장
                inst = tokens.nextToken(" \t");
            }
        }
    }
}
```

```

        instMap.put(inst, new Instruction(line));
//instruction 이름을 key로 저장
    }
    bufReader.close(); // 입력버퍼를 닫음
} catch (FileNotFoundException e) {
    System.out.println("not found");
} catch (IOException e) {
    System.out.println(e);
}
}

//get, set, search 등의 함수는 자유 구현
/**
 * 인자로 받은 명령어의 opcode를 찾아 리턴함
 * @param instruction: opcode를 알고싶은 명령어
 * @return: 해당 명령어의 opcode
 */
public int getOpcode(String instruction) {
    if (instMap.containsKey(instruction)) //주어진 명령어가
key값으로 존재한다면 해당하는 객체의 opcode를 찾아서 리턴
        return instMap.get(instruction).opcode;
    else //주어진 명령어가 key값으로 존재하지 않을 때 -1
리턴
        return -1;
}

/**
 * 인자로 받은 명령어의 피연산자 개수를 찾아 리턴함

```

```

 * @param instruction: 피연산자 개수를 알고싶은 명령어
 * @return: 해당 명령어의 피연산자 개수
 */
public int getNumberOfOperand(String instruction) {
    if (instMap.containsKey(instruction)) //주어진 명령어가
key값으로 존재한다면 해당하는 객체의 피연산자개수를 찾아서 리턴
        return instMap.get(instruction).numberOfOperand;
    else //주어진 명령어가 key값으로 존재하지 않을 때 -1
리턴
        return -1;
}

/**
 * 인자로 받은 명령어의 형식을 찾아 리턴함
 * @param instruction: 형식을 알고싶은 명령어
 * @return: 해당 명령어의 형식
 */
public int getFormat(String instruction) {
    if (instMap.containsKey(instruction)) //주어진 명령어가
key값으로 존재한다면 해당하는 객체의 형식을 찾아서 리턴
        return instMap.get(instruction).format;
    else //주어진 명령어가 key값으로 존재하지 않을 때 -1
리턴
        return -1;
}

/**

```

```

* 입력문자열이 명령어인지 검사.
*
* @param str: 검사하고싶은 문자열
* @return: 명령어가 맞다면 1, 아니라면 -1 리턴
*/
public int isInstruction(String str) {
    if(instMap.containsKey(str)) { // 입력된 문자열이 명령어
        라면(key값으로 존재한다면)
            return 1;
        }
        else {//주어진 명령어가 key값으로 존재하지 않을 때 -1
            리턴
                return -1;
        }
    }

/**
 * 입력문자열이 지시어인지 검사.
 *
 * @param str: 검사하고싶은 문자열
 * @return: 지시어가 맞다면 >=0, 아니라면 -1 리턴
 */
public int isDirective(String str) {
    String directiveTable[] =
{"START","END","BYTE","WORD","RESB","RESW","CSECT","EXTDEF","EX
TREF","EQU","ORG","LTORG"};
    int directive_num[] = {1,1,1,1,1,1,0,1,1,1,1,0};
    for(int i = 0; i < directiveTable.length; i++) {

```

```

        if(str.equals(directiveTable[i])) { // 입력된 문자
            열이 directive_table의 지시어와 일치한다면
                return directive_num[i]; // 뒤에 정보가
                포함되어 있는 지시어라면 1, 뒤에 정보가 포함되어 있지 않은 지시어라면
                0 리턴
        }
    }
    return -1; //입력된 문자열이 지시어가 아니라면 -1 리턴
}

/**
 * 명령어 하나하나의 구체적인 정보는 Instruction클래스에 담긴다.
 * instruction과 관련된 정보들을 저장하고 기초적인 연산을 수행한다.
 */
class Instruction {
    /*
     * 각자의 inst.data 파일에 맞게 저장하는 변수를 선언한다.
     *
     */
    String instruction; //명령어
    int opcode; //각 명령어에 해당하는 opcode
    int numberOfOperand; //각 명령어의 피연산자의 개수
    /** instruction이 몇 바이트 명령어인지 저장. 이후 편의성을 위함
    */
    int format;

```

열

```
/**
 * 클래스를 선언하면서 일반문자열을 즉시 구조에 맞게 파싱한다.
 * @param line : instruction 명세파일로부터 한줄씩 가져온 문자
```

열

```
 */
public Instruction(String line) {
    parsing(line);
}

/**
 * 일반 문자열을 파싱하여 instruction 정보를 파악하고 저장한다.
 * @param line : instruction 명세파일로부터 한줄씩 가져온 문자

 */
public void parsing(String line) {
    // TODO Auto-generated method stub
    StringTokenizer tokens = new StringTokenizer(line);
    instruction = tokens.nextToken(" \t");
    numberOfOperand = Integer.parseInt(tokens.nextToken("
\t"));

    format = Integer.parseInt(tokens.nextToken(" \t"));
    opcode = Integer.parseInt(tokens.nextToken(" \t"), 16);
}
//그 외 함수 자유 구현
```

}

2) TokenTable.java

```
import java.util.ArrayList;
import java.util.StringTokenizer;
```

```
/**
 * 사용자가 작성한 프로그램 코드를 단어별로 분할 한 후, 의미를 분석하
고, 최종 코드로 변환하는 과정을 총괄하는 클래스이다. <br>
 * pass2에서 object code로 변환하는 과정은 혼자 해결할 수 없고
symbolTable과 instTable의 정보가 필요하므로 이를 링크시킨다.<br>
 * 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. */
    SymbolTable symTab;
    LiteralTable literalTab;
    InstTable instTab;
```

```

/** 각 line을 의미별로 분할하고 분석하는 공간. */
ArrayList<Token> tokenList;

public int lastAddr; //각 프로그램이 끝날 때마다 마지막 주소값을

/**
 * 초기화하면서 symTable과 literalTable과 instTable을 링크시킨
다.
 * @param symTab : 해당 section과 연결되어있는 symbol table
 * @param literalTab : 해당 section과 연결되어있는 literal table
 * @param instTab : instruction 명세가 정의된 instTable
 */
public TokenTable(SymbolTable symTab, LiteralTable literalTab,
InstTable instTab) {
    tokenList = new ArrayList<>();
    this.symTab = symTab;
    this.literalTab = literalTab;
    this.instTab = instTab;
}

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

```

```

}

/**
 * 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 literal table 등을 참조하여
objectcode를 생성하고, 이를 저장한다.
 * @param index : object 코드를 생성하고자하는 토큰 인스턴스
의 인덱스
 */
public void makeObjectCode(int index){
    String register[] =
{"A","X","L","B","S","T","F","","PC","SW"};
    Token tmpToken = tokenList.get(index);

    if(tmpToken.type.equals("DIRECTIVE")) {
        if(tmpToken.directive.equals("LTORG")) { //프로
그램에 나왔던 literal값들을 메모리에 저장
for(int i = 0; i<literalTab.getSize(); i++)
{

```

```
if(literalTab.getLiteral(i).charAt(0) == 'X') { //literal이 'X'로 시작하면 literal  
값을 그대로 메모리에 할당
```

```
String str = new  
String(literalTab.getLiteral(i));  
str =  
str.substring(2, str.length()-1);
```

```
tokenList.get(index).setObjectCode(String.format("%s", str));
```

```
tokenList.get(index).setByteSize(1);
```

```
    }  
    else { //literal이 'C'로 시작하  
면 각 char 값의 ASCII code를 메모리에 할당
```

```
String str = new  
String(literalTab.getLiteral(i));  
str =  
str.substring(2, str.length()-1);
```

```
String tmpStr = "";  
for(int j = 0; j <  
str.length(); j++) {  
    tmpStr +=  
String.format("%02X", (int)str.charAt(j));  
}
```

```
tokenList.get(index).setObjectCode(tmpStr);
```

```
tokenList.get(index).setByteSize(str.length());
```

```
    }  
    }  
    }  
    else if(tmpToken.directive.equals("END")) { //프  
로그램에 나왔던 literal값들을 메모리에 저장  
    for(int i = 0; i<literalTab.getSize(); i++)  
{
```

```
if(literalTab.getLiteral(i).charAt(0) == 'X') { //literal이 'X'로 시작하면 literal  
값을 그대로 메모리에 할당
```

```
String str = new  
String(literalTab.getLiteral(i));  
str =  
str.substring(2, str.length()-1);
```

```
tokenList.get(index).setObjectCode(String.format("%s", str));
```

```
tokenList.get(index).setByteSize(1);
```

```
    }  
    else { //literal이 'C'로 시작하  
면 각 char 값의 ASCII code를 메모리에 할당
```

```
String str = new  
String(literalTab.getLiteral(i));  
str =  
str.substring(2, str.length()-1);
```

```
String tmpStr = "";  
for(int j = 0; j <  
str.length(); j++) {
```



```

                                tmpStr      +=
                                }

String.format("%02X", (int)str.charAt(j));
                                }

tokenList.get(index).setObjectCode(tmpStr);

tokenList.get(index).setByteSize(str.length());

tokenList.get(index).setByteSize(str.length());
                                }
                                else if(tmpToken.directive.equals("WORD")) {

                                }

                                }
                                else if(tmpToken.directive.equals("BYTE")) {
                                String      str      =      new
String(tmpToken.operand[0]);
                                str = str.substring(2,str.length()-1);
                                if(tmpToken.operand[0].charAt(0) ==
'X' || tmpToken.operand[0].charAt(0) == 'x') {

                                }
                                tokenList.get(index).setObjectCode(String.format("%s", str));

tokenList.get(index).setByteSize(1);
                                }
                                else if(tmpToken.operand[0].charAt(0)
== 'C' || tmpToken.operand[0].charAt(0) == 'c') {
                                String tmpStr = "";
                                for(int j = 0; j < str.length();
j++) {
                                tmpStr      +=
String.format("%02X", (int)str.charAt(j));

                                tokenList.get(index).setObjectCode(tmpStr);

tokenList.get(index).setByteSize(str.length());
                                }
                                else if(tmpToken.directive.equals("WORD")) {

                                }
                                if(Character.isDigit(tmpToken.operand[0].charAt(0))) { // 숫자라면

                                tokenList.get(index).setObjectCode(String.format("%X",
Integer.parseInt(tmpToken.operand[0])));

tokenList.get(index).setByteSize(1);
                                }
                                else { // 문자일때의 처리
                                tmpToken.operand      =
tmpToken.operand[0].split("-");
                                String      str1      =      new
String(tmpToken.operand[0]);
                                String      str2      =      new
String(tmpToken.operand[1]);
                                if(symTab.searchrefSymbol(str1)
== 1) {

                                symTab.modSymbol(String.format("+%s",str1), tmpToken.location,6);
                                }

```

```

        if(symTab.searchrefSymbol(str2)
== 1) {

symTab.modSymbol(String.format("-%s",str2), tmpToken.location,6);
        }

tokenList.get(index).setObjectCode(String.format("%06X", 0));

tokenList.get(index).setByteSize(3);
        }
    }
    else if(tmpToken.type.equals("INSTRUCTION")) {
        int op = 252;
        int opcode = op &
tmpToken.instTable.getOpcode(tmpToken.operator);

if(tmpToken.instTable.getFormat(tmpToken.operator) == 3) {
        opcode += tmpToken.getFlag(nFlag |
iFlag)/16;
    }

    if(tmpToken.plus_check == 1) { //명령어가 4형
식일 때

tokenList.get(index).setObjectCode(String.format("%02X%01X%05X",
opcode,tmpToken.getFlag(15),0));
        tokenList.get(index).setByteSize(4);

```

```

if(symTab.searchrefSymbol(tmpToken.operand[0]) == 1){

symTab.modSymbol(String.format("+%s",tmpToken.operand[0]),
tmpToken.location + 1,5);
        }
        return;
    }

if(tmpToken.instTable.getFormat(tmpToken.operator) == 1) { //1형식일 때

tokenList.get(index).setObjectCode(String.format("%02X", opcode));
        tokenList.get(index).setByteSize(1);
    }
    else
if(tmpToken.instTable.getFormat(tmpToken.operator) == 2) { //2형식일 때

if(tmpToken.instTable.getNumberOfOperand(tmpToken.operator) == 1) {
//피연산자의 개수가 1개일 때

        int reg_num = 0;
        for(int i = 0; i< 10; i++) {

            if(tmpToken.operand[0].equals(register[i])) {

                reg_num = i;
                break;
            }
        }
    }
}

```

```

tokenList.get(index).setObjectCode(String.format("%02X%01X%01X",
opcode,reg_num,0));

tokenList.get(index).setByteSize(2);
    }
        e            l            s            e
if(tmpToken.instTable.getNumberOfOperand(tmpToken.operator) == 2) {
//피연산자의 개수가 2개일 때
        int reg_num1 = 0, reg_num2 =
0;
        for(int i = 0; i<10; i++) {

if(tmpToken.operand[0].equals(register[i])) {
                reg_num1 = i;
            }

if(tmpToken.operand[1].equals(register[i])) {
                reg_num2 = i;
            }
        }

tokenList.get(index).setObjectCode(String.format("%02X%01X%01X",
opcode,reg_num1,reg_num2));

tokenList.get(index).setByteSize(2);
    }
        e            l            s            e

```

```

if(tmpToken.instTable.getFormat(tmpToken.operator) == 3) { //3형식일 때

if(tmpToken.instTable.getNumberOfOperand(tmpToken.operator) == 0) {
//피연산자가 존재하지 않을 때

tokenList.get(index).setObjectCode(String.format("%02X%04X",
opcode,0));

tokenList.get(index).setByteSize(3);

                return;
            }
            else { //피연산자가 존재할 때

if(tmpToken.operand[0].contains("#")) { //immediate addressing일 때
                tmpToken.operand[0]
= tmpToken.operand[0].substring(1);

tokenList.get(index).setObjectCode(String.format("%02X%04X",
opcode,Integer.parseInt(tmpToken.operand[0])));

tokenList.get(index).setByteSize(3);

                return;
            }

if(tmpToken.operand[0].contains("@")) { //indirection addressing일 때
                tmpToken.operand[0]
= tmpToken.operand[0].substring(1);
            }

```

```

        int target;

if(tmpToken.operand[0].contains("=")) { //피연산자가 literal일 때
        tmpToken.operand[0]
= tmpToken.operand[0].substring(1);

        target =

literalTab.search(tmpToken.operand[0]);
    }
    else {
        target =

symTab.search(tmpToken.operand[0]);
    }
    int pc =

tokenList.get(index+1).location;

    int addr = target - pc;

tokenList.get(index).setObjectCode(String.format("%02X%01X%03X",
opcode,tmpToken.getFlag(15),addr&0xFFF));

tokenList.get(index).setByteSize(3);
    }
}
return;
}

```

/**

* 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 데
에 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;

int plus_check; //operand에 +가 포함되어 있는지에 대한 정보를

저장

String directive; // 명령어 라인 중 지시어

String type; // 해당 명령어 라인이 어떤 타입인지 명시

// object code 생성 단계에서 사용되는 변수들

```
String objectCode;  
int byteSize;
```

```
InstTable instTable;
```

```
/**
```

```
 * 클래스를 초기화 하면서 바로 line의 의미 분석을 수행한다.  
 * @param line 문장단위로 저장된 프로그램 코드  
 */
```

```
public Token(String line, InstTable instTable) {  
    //토큰을 파싱하기 위해 instruction table 링크  
    this.instTable = instTable;  
    //initialize 추가  
    parsing(line);  
}
```

```
/**
```

```
 * line의 실질적인 분석을 수행하는 함수. Token의 각 변수에 분  
석한 결과를 저장한다.
```

```
 * @param line 문장단위로 저장된 프로그램 코드.  
 */
```

```
public void parsing(String line) {  
    String tmpStr;  
    String tok;  
    String tmpInst;  
    int numofOperand;  
    nixbpe = '0';
```

없으면 리턴

계어라면

```
tmpStr = new String(line);  
operand = new String[3];
```

```
if(tmpStr.charAt(0) == '.') {  
    comment = new String(tmpStr);  
    type = new String("COMMENT");  
    return;  
}
```

```
StringTokenizer tokens = new StringTokenizer(tmpStr);  
if(tokens.hasMoreTokens() == false) { //분리할 문자열이  
    return;  
}
```

```
tok = tokens.nextToken(" \t");
```

```
if(tok.charAt(0) == '+') { //기계어에 +가 추가되어 있으면  
    plus_check = 1; // 토큰테이블에 따로 저장  
    tmpInst = new String(tok.substring(1));  
    tok = tmpInst;  
}
```

```
if(instTable.isInstruction(tok) == 1) { //잘린 문자열이 기  
    operator = tok;  
    if(instTable.getFormat(tok) == 3) {  
        if(plus_check == 1) {  
            setFlag(TokenTable.eFlag,1);
```

```

//4형식명령어이므로 e를 표시
    }
    else {
        setFlag(TokenTable.pFlag,1); //
기계어에 +가 추가되어 있지 않다면 pc relative를 사용할 것이므로 p를 표
시
    }
}
if((numofOperand ==
instTable.getNumberOfOperand(tok)) != 0) { //operand가 있다면
    if(numofOperand == 1) { //operand 개
수가 1개일 때
        tok = tokens.nextToken();
        operand[0] = new String(tok);
        if(tok.contains(",")) {
//BUFFER,x 같이 배열을 쓰는 경우에 토큰분리
            operand =
tok.split(",");

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

setFlag(TokenTable.xFlag,1);

        }
    }
}
else { // operand 개수가 2개일 때
    tok = tokens.nextToken();
    operand[0] = new String(tok);

```

```

        operand = tok.split(",");
    }

//addressing 방식이
if(operand[0].charAt(0) == '@')
{///indirection addressing일 때
    setFlag(TokenTable.nFlag,1);
    setFlag(TokenTable.iFlag,0);
}
else if(operand[0].charAt(0) == '#')
{///immediate addressing일 때
    setFlag(TokenTable.nFlag,0);
    setFlag(TokenTable.iFlag,1);
    setFlag(TokenTable.pFlag,0);
}
else {// 둘다 아니라면
    if(instTable.getFormat(tok) ==
3) {//그런데 기계어의 형식이 3형식/4형식이라면

setFlag(TokenTable.nFlag,1);

setFlag(TokenTable.iFlag,1);

    }
}
else {//operand가 없는 3형식 명령어라면
    if(instTable.getFormat(tok) == 3) {
        setFlag(TokenTable.nFlag,1);
    }
}

```

```

        setFlag(TokenTable.iFlag,1);
    }
}
if(tokens.hasMoreTokens() == true) { //comment
    tok = tokens.nextTokent("\0");
    comment = tok;
}
type = new String("INSTRUCTION");
}
else if((numofOperand = instTable.isDirective(tok)) >= 0)
{ //잘린 문자열이 지시어라면
    directive = new String(tok);
    if(numofOperand == 1) { //지시어 뒤에 다른 정보
        tok = tokens.nextToken();
        operand[0] = new String(tok);
        if(tok.contains(",")) {
            operand = tok.split(",");
        }
    }
    if(tokens.hasMoreTokens() == true) { //comment
        tok = tokens.nextToken("\0");
        comment = tok;
    }
    type = new String("DIRECTIVE");
}

```

분리

가 포함되어 있을 때

분리

```

else { //잘린 문자열이 label이라면
    label = new String(tok);
    tok = tokens.nextToken();
    if(tok.charAt(0) == '+') { //기계어에 +가 추가되
        plus_check = 1; // 토큰테이블에 따로
        tmpInst = new String(tok.substring(1));
        tok = tmpInst;
    }
    if(instTable.isInstruction(tok) == 1) { //잘린 문자
        operator = tok;
        if(instTable.getFormat(tok) == 3) {
            if(plus_check == 1) {
                setFlag(TokenTable.eFlag,1); //4형식명령어이므로 e를 표시
            }
            else {
                setFlag(TokenTable.pFlag,1); //기계어에 +가 추가되어 있지 않다면 pc
                relative를 사용할 것이므로 p를 표시
            }
        }
        if((numofOperand = instTable.getNumberOfOperand(tok)) != 0) { //operand가 있다면
            if(numofOperand == 1) {

```

어 있으면

저장

열이 기계어라면

//operand 개수가 1개일 때		{//indirection addressing일 때
	tok =	
tokens.nextToken();		setFlag(TokenTable.nFlag,1);
	operand[0] = new	
String(tok);		setFlag(TokenTable.iFlag,0);
	if(tok.contains(",")) {	}
//BUFFER,x 같이 배열을 쓰는 경우에 토큰분리		else if(operand[0].charAt(0) ==
	operand =	'#') {////immediate addressing일 때
tok.split(",");		
		setFlag(TokenTable.nFlag,0);
if(operand[1].contains("X")) {		setFlag(TokenTable.iFlag,1);
setFlag(TokenTable.xFlag,1);		setFlag(TokenTable.pFlag,0);
	}	}
	}	else {// 둘다 아니라면
else { // operand 개수가 2개일		
때		if(instTable.getFormat(tok) == 3) {//그런데 기계어의 형식이 3형식/4형식
	tok =	이라면
tokens.nextToken();		setFlag(TokenTable.nFlag,1);
	operand[0] = new	
String(tok);		setFlag(TokenTable.iFlag,1);
	operand =	
tok.split(",");		}
	}	}
		}
		else {//operand가 없는 3형식 명령어라
//addressing 방식이		
if(operand[0].charAt(0) == '@') 면		

<pre> 3) { setFlag(TokenTable.nFlag,1); setFlag(TokenTable.iFlag,1); } } if(tokens.hasMoreTokens() == true) { //comment 분리 tok = tokens.nextToken("\0"); comment = tok; } type = new String("INSTRUCTION"); } else if((numofOperand == instTable.isDirective(tok)) >= 0) { //잘린 문자열이 지시어라면 directive = new String(tok); if(numofOperand == 1) { //지시어 뒤에 다른 정보가 포함되어 있을 때 tok = tokens.nextToken(); operand[0] = new String(tok); if(tok.contains(",")) { operand = tok.split(","); } } </pre>	<pre> == //comment 분리 == return; } /** * 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(value == 1) { nixbpe = flag; } else { nixbpe ^= flag; } } </pre>	<pre> if(tokens.hasMoreTokens() == true) { tok = tokens.nextToken("\0"); comment = tok; } type = new String("DIRECTIVE"); } return; } /** * 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(value == 1) { nixbpe = flag; } else { nixbpe ^= flag; } } </pre>
---	--	---

```

/**
 * 원하는 flag들의 값을 얻어올 수 있다. flag의 조합을 통해 동시
에 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;
}

/**
 * loc값을 인자로 받아와 저장한다.
 *
 * @param loc : 저장하고자하는 location
 *
 */
public void setlocation(int loc) {
    location = loc;
}

/**
 * object code을 인자로 받아와 저장한다.

```

```

*
* @param objcode : 저장하고자하는 object code
*
*/
public void setObjectCode(String objcode) {
    objectCode = new String(objcode);
}

/**
 * byteSize을 인자로 받아와 저장한다.
 *
 * @param bsize : 저장하고자하는 byte code
 *
 */
public void setByteSize(int bSize) {
    byteSize = bSize;
}
}

```

3) SymbolTable.java

```

import java.util.ArrayList;

```

```

/**
 * symbol과 관련된 데이터와 연산을 소유한다.
 * section 별로 하나씩 인스턴스를 할당한다.
 */
public class SymbolTable {
    ArrayList<String> symbolList;
    ArrayList<Integer> locationList;
    ArrayList<String> modList;
    ArrayList<Integer> modLocationList;
    ArrayList<Integer> modSize;
    ArrayList<String> defList;
    ArrayList<String> refList;
    // 기타 literal, external 선언 및 처리방법을 구현한다.

    public SymbolTable() { //초기화
        symbolList = new ArrayList<>();
        locationList = new ArrayList<>();
        modList = new ArrayList<>();
        modLocationList = new ArrayList<>();
        modSize = new ArrayList<>();
        defList = new ArrayList<>();
        refList = new ArrayList<>();
    }

    /**
     * 새로운 Symbol을 table에 추가한다.
     * @param symbol : 새로 추가되는 symbol의 label

```

```

        * @param location : 해당 symbol이 가지는 주소값
        * 주의 : 만약 중복된 symbol이 putSymbol을 통해서 입력된다면
        이는 프로그램 코드에 문제가 있음을 나타낸다.
        * 매칭되는 주소값의 변경은 modifySymbol()을 통해서 이루어져야
        한다.

        */
        public void putSymbol(String symbol, int location) {
            if(symbolList.contains(symbol) == false) {
                symbolList.add(symbol);
                locationList.add(location);
            }
        }

    /**
     * 기존에 존재하는 symbol 값에 대해서 가리키는 주소값을 변경
     한다.

     * @param symbol : 변경을 원하는 symbol의 label
     * @param newLocation : 새로 바꾸고자 하는 주소값
     */
        public void modifySymbol(String symbol, int newLocation) {
            int index;
            index = symbolList.indexOf(symbol);
            locationList.set(index, newLocation);
        }

    /**
     * modification record에 작성할 symbol들을 table에 추가한다.
     * @param symbol : modify될 symbol의 label

```

```

    * @param location : 해당 symbol의 location
    * @param size : modify될 바이트의 크기
    */
    public void modSymbol(String symbol, int location, int size) {
        modList.add(symbol);
        modLocationList.add(location);
        modSize.add(size);
    }

    /**
     * 지시어 EXTDEF가 나왔을 때 symbol을 저장
     * @param symbol : 새로 추가되는 symbol의 label
     *
     */
    public void setdefSymbol(String symbol) {
        defList.add(symbol);
    }

    /**
     * 지시어 EXTREF가 나왔을 때 symbol을 저장
     * @param symbol : 새로 추가되는 symbol의 label
     *
     */
    public void setrefSymbol(String symbol) {
        refList.add(symbol);
    }

    /**

```

```

     * 주어진 symbol이 definition된 symbol인지 확인
     * @param symbol : 확인하고자 하는 symbol
     * @return : definition된 symbol이면 1, 아니라면 -1 반환
     *
     */
    public int searchdefSymbol(String symbol) {
        if(defList.contains(symbol) == true) {
            return 1;
        }
        else {
            return -1;
        }
    }

    /**
     * 주어진 symbol이 reference된 symbol인지 확인
     * @param symbol : 확인하고자 하는 symbol
     * @return : reference된 symbol이면 1, 아니라면 -1 반환
     *
     */
    public int searchrefSymbol(String symbol) {
        if(refList.contains(symbol) == true) {
            return 1;
        }
        else {
            return -1;
        }
    }
}

```

```

/**
 * 인자로 전달된 symbol이 어떤 주소를 지칭하는지 알려준다.
 * @param symbol : 검색을 원하는 symbol의 label
 * @return symbol이 가지고 있는 주소값. 해당 symbol이 없을 경
우 -1 리턴
 */
public int search(String symbol) {
    int address = 0;

    if(symbolList.contains(symbol)) {
        for (int i = 0; i < symbolList.size(); i++) {
            if (symbol.equals(symbolList.get(i))){
                address = locationList.get(i);
                break;
            }
        }
    }
    else {
        address = -1;
    }

    return address;
}

/**
 * index에 해당하는 symbol을 리턴한다.

```

```

 * @param index : 리턴할 symbol의 인덱스
 * @return : symbol
 */
public String getSymbol(int index) {
    return symbolList.get(index);
}

/**
 * index를 이용하여 symbol을 찾은 후 그 symbol에 해당하는
address를 리턴한다.
 * @param index : 리턴할 address의 인덱스
 * @return : address
 */
public int getaddress(int index) {
    return locationList.get(index);
}

/**
 * symboltable의 크기를 리턴한다.
 * @return : symboltable의 크기
 */
public int getSize() {
    return symbolList.size();
}

```

```
}
```

4) LiteralTable.java

```
import java.util.ArrayList;
```

```
/**
```

```
 * literal과 관련된 데이터와 연산을 소유한다.
```

```
 * section 별로 하나씩 인스턴스를 할당한다.
```

```
 */
```

```
public class LiteralTable {
```

```
    ArrayList<String> literalList;
```

```
    ArrayList<Integer> locationList;
```

```
    // 기타 literal, external 선언 및 처리방법을 구현한다.
```

```
    public LiteralTable() {
```

```
        literalList = new ArrayList<>();
```

```
        locationList = new ArrayList<>();
```

```
    }
```

```
    /**
```

```
     * 새로운 Literal을 table에 추가한다.
```

```
     * @param literal : 새로 추가되는 literal의 label
```

```
     * @param location : 해당 literal이 가지는 주소값
```

```
     * 주의 : 만약 중복된 literal이 putLiteral을 통해서 입력된다면 이  
는 프로그램 코드에 문제가 있음을 나타낸다.
```

```
     * 매칭되는 주소값의 변경은 modifyLiteral()을 통해서 이루어져야  
한다.
```

```
 */
```

```
public void putLiteral(String literal, int location) {
```

```
    String tmpInst = new String(literal.substring(1));
```

```
    literal = tmpInst;
```

```
    if(literalList.contains(literal) == false) {
```

```
        literalList.add(literal);
```

```
        locationList.add(location);
```

```
    }
```

```
}
```

```
/**
```

```
 * 기존에 존재하는 literal 값에 대해서 가리키는 주소값을 변경한  
다.
```

```
 * @param literal : 변경을 원하는 literal의 label
```

```
 * @param newLocation : 새로 바꾸고자 하는 주소값
```

```
 */
```

```
public void modifyLiteral(String literal, int newLocation) {
```

```
    int index;
```

```
    index = literalList.indexOf(literal);
```

```
    locationList.set(index, newLocation);
```

```
}
```

```
/**
```

```
 * 인자로 전달된 literal이 어떤 주소를 지칭하는지 알려준다.
```

```
 * @param literal : 검색을 원하는 literal의 label
```

```
 * @return literal이 가지고 있는 주소값. 해당 literal이 없을 경우
```

-1 리턴

```
*/
public int search(String literal) {
    int address = 0;

    if(literalList.contains(literal)) {
        for (int i = 0; i < literalList.size(); i++) {
            if (literal.equals(literalList.get(i))){
                address = locationList.get(i);
                break;
            }
        }
    }
    else {
        address = -1;
    }
    return address;
}

/**
 * literal들의 주소를 넣어주는 함수이다
 * @param locctr : 현재 프로그램의 주소
 * @return locctr : literal들의 주소를 넣어주고 증가한 주소 반환
 */
public int addrLiteralTab(int locctr) {
    String tmpstr;
    int strsize;
    for (int i = 0; i < literalList.size(); i++) {
```

```
        if(search(literalList.get(i)) == -2) {
            modifyLiteral(literalList.get(i),locctr);
            if(literalList.get(i).charAt(0) == 'C' ||
literalList.get(i).charAt(0) == 'c') { //literal이 'C'인지 확인
                tmpstr = new
String(literalList.get(i));
                strsize = tmpstr.length();
                tmpstr =
tmpstr.substring(2,strsize-1);
                locctr += tmpstr.length(); //char
개수만큼 locctr 증가
            }
            else if(literalList.get(i).charAt(0) == 'X'
|| literalList.get(i).charAt(0) == 'x') { //literal이 'X'인지 확인
                tmpstr = new
String(literalList.get(i));
                strsize = tmpstr.length();
                tmpstr =
tmpstr.substring(2,strsize-1);
                locctr += tmpstr.length()/2;
                //char개수만큼 locctr 증가
            }
        }
    }
    return locctr;
}

/**
```

```
* index에 해당하는 literal을 리턴한다.  
* @param index : 리턴할 literal의 인덱스  
* @return : literal  
*  
*/
```

```
public String getLiteral(int index) {  
    return literalList.get(index);  
}
```

```
/**
```

* index를 이용하여 literal을 찾은 후 해당하는 literal의 address
을 리턴한다.

```
* @param index : 리턴할 address의 인덱스  
* @return : address  
*  
*/
```

```
public int getAddress(int index) {  
    return locationList.get(index);  
}
```

```
/**
```

```
* literaltable의 크기를 리턴한다.  
* @return : literalable의 크기  
*  
*/
```

```
public int getSize() {  
    return literalList.size();  
}
```


5) Assembler.java

```
import java.util.ArrayList;
import java.io.*;
```

```
/**
 * Assembler :
 * 이 프로그램은 SIC/XE 머신을 위한 Assembler 프로그램의 메인 루틴이
다.
 * 프로그램의 수행 작업은 다음과 같다.
 * 1) 처음 시작하면 Instruction 명세를 읽어들이어서 assembler를 세팅한다.
 * 2) 사용자가 작성한 input 파일을 읽어들이고 저장한다.
 * 3) input 파일의 문장들을 단어별로 분할하고 의미를 파악해서 정리한다.
(pass1)
 * 4) 분석된 내용을 바탕으로 컴퓨터가 사용할 수 있는 object code를 생
성한다. (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 출력시키지 말 것. (채점상의 이유. 주
석에 포함된 한글은 상관 없음)
 *
```

```
 *
 * + 제공하는 프로그램 구조의 개선방법을 제안하고 싶은 분들은 보고서
의 결론 뒷부분에 첨부 바랍니다. 내용에 따라 가산점이 있을 수 있습니다.
 */
public class Assembler {
    /** instruction 명세를 저장한 공간 */
    InstTable instTable;
    /** 읽어들이고 input 파일의 내용을 한 줄 씩 저장하는 공간. */
    ArrayList<String> lineList;
    /** 프로그램의 section별로 symbol table을 저장하는 공간*/
    ArrayList<SymbolTable> symtabList;
    /** 프로그램의 section별로 literal table을 저장하는 공간*/
    ArrayList<LiteralTable> 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;

    static int locctr;
    static int sectionNum;

    /**
     * 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<SymbolTable>();
    literalTabList = new ArrayList<LiteralTable>();
    TokenList = new ArrayList<TokenTable>();
    codeList = new ArrayList<String>();
}

/**
 * 어셈블러의 메인 루틴
 */
public static void main(String[] args) {
    Assembler assembler = new Assembler("inst.data");
    assembler.loadInputFile("input.txt");
    assembler.pass1();

    assembler.printSymbolTable("symtab_20160433.txt");
    assembler.printLiteralTable("literalTab_20160433.txt");
    assembler.pass2();
    assembler.printObjectCode("output_20160433.txt");
}

/**

```

```

* inputFile을 읽어들여서 lineList에 저장한다.
* @param inputFile : input 파일 이름.
*/
private void loadInputFile(String inputFile) {
    try {
        // 파일 객체 생성
        File file = new File(inputFile);
        // 입력 스트림 생성
        FileReader fileReader = new FileReader(file);
        // 입력 버퍼 생성
        BufferedReader bufReader = new
BufferedReader(fileReader);
        String line = "";

        while ((line = bufReader.readLine()) != null) { //
파일을 끝까지 읽기 전까지 한줄씩 읽음
            lineList.add(line); // 읽어들인 라인들은
lineList에 저장
        }
        bufReader.close(); // 입력버퍼를 닫음
    } catch (FileNotFoundException e) {
        System.out.println("not found");
    } catch (IOException e){
        System.out.println(e);
    }
}

/**

```

```

* pass1 과정을 수행한다.
*   1) 프로그램 소스를 스캔하여 토큰단위로 분리한 뒤 토큰테이블 생성
*   2) label을 symbolTable에 정리
*
*   주의사항 : SymbolTable과 TokenTable은 프로그램의
section별로 하나씩 선언되어야 한다.
*/
private void pass1() {
    // TODO Auto-generated method stub
    String line;
    Token tmpToken;
    for(int i = 0; i < lineList.size(); i++ ) {
        line = lineList.get(i);
        if(line.contains("START")) { //프로그램이 시작될
            때 tokentable, symtab, literalab 생성
            locctr = 0;
            symtabList.add(new SymbolTable());
            literalabList.add(new LiteralTable());
            T o k e n L i s t . a d d ( n e w
TokenTable(symtabList.get(sectionNum),    literalabList.get(sectionNum),
instTable));
        }
        else if(line.contains("CSECT")) { //sub 프로그램
이 시작될 때 tokentable, symtab, literalab 생성
            sectionNum++;
            locctr = 0;
            symtabList.add(new SymbolTable());

```

```

        literalabList.add(new LiteralTable());
        T o k e n L i s t . a d d ( n e w
TokenTable(symtabList.get(sectionNum),    literalabList.get(sectionNum),
instTable));
    }
    TokenList.get(sectionNum).putToken(line);
}

//토큰 분리한 것들을 가지고 loc를 포함한 기타 필요한 정
보들 저장
for(int i = 0; i<sectionNum + 1; i++) {
    for(int j = 0; j <
TokenList.get(i).tokenList.size(); j++) {
        tmpToken =
TokenList.get(i).getToken(j);
        if(tmpToken.type.equals("DIRECTIVE")) {
            if(tmpToken.directive.equals("START")) { //프로그램이 시작될 때
                locctr =
Integer.parseInt(tmpToken.operand[0]);
                TokenList.get(i).getToken(j).setlocation(locctr);
                symtabList.get(i).putSymbol(tmpToken.label, locctr);
                continue;
            }
            e l s e

```

```
if(tmpToken.directive.equals("CSECT")) { //sub 프로그램이 시작될 때
```

```
TokenList.get(i-1).lastAddr = locctr;
```

```
locctr = 0;
```

```
TokenList.get(i).getToken(j).setlocation(locctr);
```

```
symtabList.get(i).putSymbol(tmpToken.label, locctr);
```

```
continue;
```

```
}
```

```
if(tmpToken.directive.contentEquals("EXTDEF")) { //EXTDEF 뒤에 나오는  
값들 저장
```

```
for(int k = 0;
```

```
k<tmpToken.operand.length; k++) {
```

```
symtabList.get(i).setdefSymbol(tmpToken.operand[k]);
```

```
}
```

```
}
```

```
if(tmpToken.directive.contentEquals("EXTREF")) { //EXTREF 뒤에 나오는  
값들 저장
```

```
for(int k = 0;
```

```
k<tmpToken.operand.length; k++) {
```

```
symtabList.get(i).setrefSymbol(tmpToken.operand[k]);
```

```
}
```

```
}
```

```
}
```

```
if(tmpToken.label != null) {
```

```
symtabList.get(i).putSymbol(tmpToken.label, locctr);
```

```
}
```

```
TokenList.get(i).getToken(j).setlocation(locctr);
```

```
if(tmpToken.type.equals("INSTRUCTION")) {
```

```
if(tmpToken.plus_check == 1) {
```

```
//명령어의 형식에 맞춰 locctr값 증가
```

```
locctr += 4;
```

```
}
```

```
else {
```

```
locctr +=
```

```
tmpToken.instTable.getFormat(tmpToken.operator);
```

```
}
```

```
if(tmpToken.operand[0] != null)
```

```
{
```

```
if(tmpToken.operand[0].contains("=")) { //indirection addressing일 때
```

```
literalList.get(i).putLiteral(tmpToken.operand[0], -2);
```

```
}
```

```
}
```

```
}
```

어셈블리 코드	C 코드
if(tmpToken.type.equals("DIRECTIVE")) {	}
if(tmpToken.directive.equals("WORD")) { //locctr을 3 증가시킴	else { // 그렇지 않은 경우 피연산자를 통해 주소값을 계산하여 저장
locctr += 3;	
}	if(tmpToken.operand[0].contains("-")) {
if(tmpToken.directive.equals("RESW")) { //locctr을 (3 * 피연산자값)만큼 증가시킴	tmpToken.operand = tmpToken.operand[0].split("-");
locctr += 3 *	String
Integer.parseInt(tmpToken.operand[0]);	String
}	str1 = new String(tmpToken.operand[0]);
if(tmpToken.directive.equals("RESB")) { //locctr을 피연산자값만큼 증가시킴	str2 = new String(tmpToken.operand[1]);
locctr +=	syntabList.get(i).modifySymbol(tmpToken.label, syntabList.get(i).search(str1) - syntabList.get(i).search(str2));
Integer.parseInt(tmpToken.operand[0]);	TokenList.get(i).getToken(j).setlocation(syntabList.get(i).search(str1) - syntabList.get(i).search(str2));
}	}
if(tmpToken.directive.equals("BYTE")) { //locctr을 1 증가시킴	}
locctr += 1;	}
}	}
if(tmpToken.directive.equals("EQU")) {	if(tmpToken.directive.equals("LTORG")) { //프로그램에 나왔던 literal을 저장
if(tmpToken.operand[0].equals("*")) { // 피연산자가 *인 경우	locctr =
syntabList.get(i).modifySymbol(tmpToken.label, locctr); //현재 locctr의	literalTabList.get(i).addrLiteralTab(locctr);
	}

```

        e        l        s        e
if(tmpToken.directive.equals("END")) { //프로그램에 나왔던 literal을 저장
        locctr        =
literalTabList.get(i).addrLiteralTab(locctr);

TokenList.get(i).lastAddr = locctr;
        }
    }
}

return;
}

/**
 * 작성된 SymbolTable들을 출력형태에 맞게 출력한다.
 * @param fileName : 저장되는 파일 이름
 */
private void printSymbolTable(String fileName) {
    try {
        // 파일 객체 생성
        File file = new File(fileName);
        // 출력 스트림 생성
        FileWriter fileWriter = new FileWriter(file);
        // 출력 버퍼 생성
        BufferedWriter bufferedWriter = new
BufferedWriter(fileWriter);
        for(int i = 0; i< symtabList.size(); i++) {

```

```

        for(int        j        =        0;
j<symtabList.get(i).getSize(); j++) {
        bufferedWriter.write(String.format("%s\t\t\t%X",
symtabList.get(i).getSymbol(j),symtabList.get(i).getaddress(j)));
        bufferedWriter.newLine();
        }
        bufferedWriter.newLine();
    }
    bufferedWriter.close(); // 출력버퍼를 닫음
}catch(IOException e) {
    System.out.println(e);
}
}

/**
 * 작성된 LiteralTable들을 출력형태에 맞게 출력한다.
 * @param fileName : 저장되는 파일 이름
 */
private void printLiteralTable(String fileName) {
    try {
        // 파일 객체 생성
        File file = new File(fileName);
        // 출력 스트림 생성
        FileWriter fileWriter = new FileWriter(file);
        // 출력 버퍼 생성
        BufferedWriter bufferedWriter = new
BufferedWriter(fileWriter);

```

```

        String str;
        for(int i = 0; i< literalTabList.size(); i++) {
            for(int j = 0; j<literalTabList.get(i).getSize(); j++) {
                str = literalTabList.get(i).getLiteral(j);
                str = str.substring(2,str.length()-1);

                bufferedWriter.write(String.format("%s\t\t\t%X",
                str,literalTabList.get(i).getaddress(j)));
            }
            if(literalTabList.get(i).getSize() == 0) {
                continue;
            }
            bufferedWriter.newLine();
        }
        bufferedWriter.close(); // 출력버퍼를 닫음
    }catch(IOException e) {
        System.out.println(e);
    }
}

/**
 * pass2 과정을 수행한다.
 * 1) 분석된 내용을 바탕으로 object code를 생성하여 codeList
에 저장.
 */

```

```

private void pass2() {
    int num = 0;
    String textTmp = null;
    String code = null;
    //토큰분리한 것들을 가지고 각 라인별 object code 작성
    for(int i = 0; i<TokenList.size(); i++) {
        for(int j = 0; j<TokenList.get(i).tokenList.size(); j++) {
            TokenList.get(i).makeObjectCode(j);
        }
    }

    Token tmpToken;
    String str;
    int check = 1;
    int line_max = 0;
    int line_check = 0;
    int line = 0;
    //프로그램의 object code를 작성하여 codeList에 저장
    for(int i = 0; i<TokenList.size(); i++) {
        int notFinish = -1;
        for(int j = 0; j<TokenList.get(i).tokenList.size(); j++) {
            tmpToken = TokenList.get(i).getToken(j);

            //Header record 작성
            if(tmpToken.type.equals("DIRECTIVE")) {

```

```

if(tmpToken.directive.equals("START") || str = new String("R");
tmpToken.directive.equals("CSECT")) { for(int k = 0; k <

int startAddr = symtabList.get(i).refList.size(); k++) {
TokenList.get(i).getToken(0).location; str +=
String name = new String.format("%-6s", symtabList.get(i).refList.get(k));
String(TokenList.get(i).getToken(0).label); }
int lastAddr = codeList.add(str);
TokenList.get(i).lastAddr; continue;
} }

codeList.add(String.format("H%-6s%06X%06X",name,startAddr,lastAddr)); }

} //Text record 작성
if(tmpToken.objectCode != null) {
line_check = 0;
if(check == 1) {
textTmp =
String.format("T%06X",TokenList.get(i).getToken(0).location+line);
check = 0;
notFinish = 2;
}
num += tmpToken.byteSize;
if(num > 30) { //현재까지의
byte의 개수가 한줄에 쓸수 있는 분량보다 많다면
num -=
tmpToken.byteSize;

check = 1;
textTmp =
String.format("%s%02X%s", textTmp, num, code);
}
}
if(tmpToken.directive.equals("EXTDEF")){ //Define record 작성
str = new String("D");
for(int k = 0; k <
symtabList.get(i).defList.size(); k++) {
str +=
String.format("%-6s%06X", symtabList.get(i).defList.get(k),
symtabList.get(i).search(symtabList.get(i).defList.get(k)));
}
codeList.add(str);
continue;
}

}
if(tmpToken.directive.equals("EXTREF")){ //Refer record 작성

```



```

        codeList.add(textTmp);
        line += num;
        textTmp = null;
        code = null;
        line_max = 1;
        num = 0;
        j--;
        continue;
    }
    else {
        if(notFinish == 1) {
            textTmp =
String.format("T%06X",TokenList.get(i).getToken(j).location);
        }
        if(line_max == 1) {
            textTmp =
String.format("T%06X",TokenList.get(i).getToken(0).location+line);
            line_max = 0;
        }
        if(code != null) {
            code +=
String.format("%s",TokenList.get(i).getToken(j).objectCode);
        }
        else {
            code =
String.format("%s",TokenList.get(i).getToken(j).objectCode);
        }
        line_check = 1;
    }
}
continue;
}
else {
    if(line_check == 1) {
        if(notFinish == 2) {
            if(textTmp ==
null) {
                continue;
            }
            notFinish = 1;
            textTmp =
String.format("%s%02X%s", textTmp,num,code);
            codeList.add(textTmp);
            line += num;
            textTmp = null;
            code = null;
            num = 0;
        }
    }
}
if(line_check == 1) {
    textTmp = String.format("%s%02X%s",
textTmp,num,code);
    codeList.add(textTmp);
}
}
}

```

```

        textTmp = null;
        code = null;
        num = 0;
        line = 0;
        check = 1;
    }

    if (symtabList.get(i).refList.size() != 0) {
//Modification record 작성
        for (int k = 0; k <
symtabList.get(i).modList.size(); k++) {

codeList.add(String.format("M%06X%02X%s",symtabList.get(i).modLocatio
nList.get(k),symtabList.get(i).modSize.get(k),symtabList.get(i).modList.get(
k)));

        }
    }

    if (i == 0) { //End record 작성
        codeList.add(String.format("E%06X",
TokenList.get(i).getToken(0).location));
    }
    else {
        codeList.add("E");
    }
}
}

```

```

    }

/**
 * 작성된 codeList를 출력형태에 맞게 출력한다.
 * @param fileName : 저장되는 파일 이름
 */
private void printObjectCode(String fileName) {
    try {
        // 파일 객체 생성
        File file = new File(fileName);
        // 출력 스트림 생성
        FileWriter fileWriter = new FileWriter(file);
        // 출력 버퍼 생성
        BufferedWriter bufferedWriter = new
BufferedWriter(fileWriter);

        for(int i = 0; i< codeList.size(); i++) {
            bufferedWriter.write(String.format("%s",
codeList.get(i)));

            bufferedWriter.newLine();

            if(codeList.get(i).charAt(0) == 'E') {
                bufferedWriter.newLine();
            }
        }
        bufferedWriter.close(); // 출력버퍼를 닫음
    }catch(IOException e) {
        System.out.println(e);
    }
}

```

}
}
}