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
과제명	SIC/XE 어셈블러 구현 (Project #1b)
담당교수	최 재 영 교 수
제출인	컴퓨터학부 20163340 강원경 (출석번호 107번)
제출일	2021년 05월 17일

차 례

1장. 프로젝트 동기/목적

이번 프로젝트는 우리가 2021년 1학기 '시스템 프로그래밍' 시간에 배운 SIC/XE 머신의 어셈블러 프로그램 을 구현하는 것이다. 지난 프로젝트에서는 절차 지향 언어인 C를 통해 어셈블러를 구현하였다면, 이번에는 객체지향 언어인 JAVA를 통해 동일한 프로그램을 구현해내는 것이 목표이다.

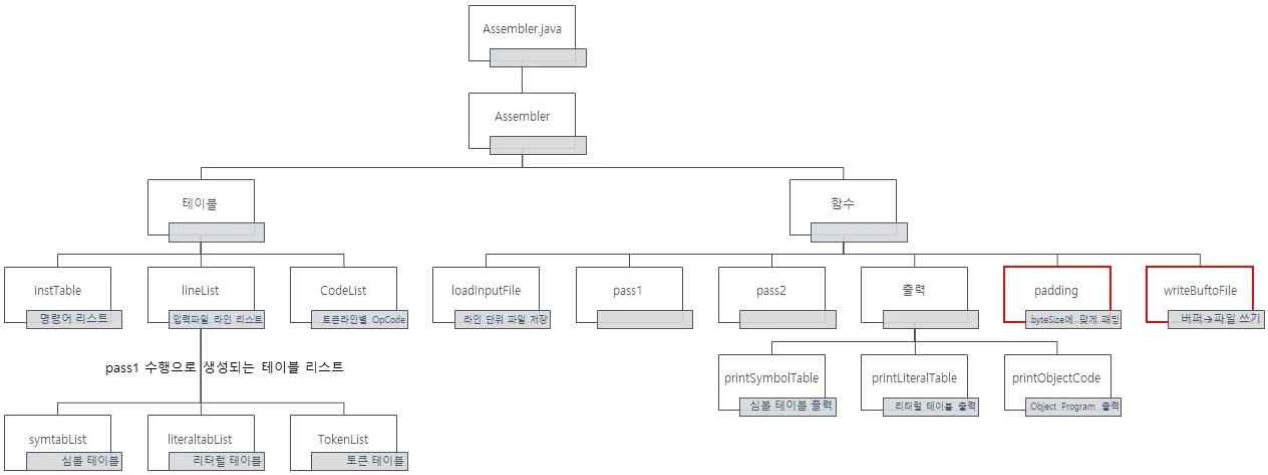
절차지향 언어인 C는 동작 순서에 맞게 코드를 구현하기만 하면 된다. 하지만 JAVA는 객체지향 언어이고, 객체 지향 언어의 특성상 각각의 객체들이 하나의 프로그램에서 유기적으로 결합하여 동작한다. 이 때문에 각각의 객체를 어떻게 이용할 것인지, 내가 구현하려는 프로그램의 기능과 동작을 어느 부분에 결합시킬 것인지에 대한 고민이 설계 과정에서 중요한 부분 중 하나이다.

평소 C++를 주로 사용했고, C++ 역시 객체지향 언어이기 때문에 프로그램을 설계하는 것은 그리 어렵진 않았다. 하지만, JAVA를 사용 해본지 꽤 오래되었고, JAVA의 또 다른 특성인 Call by Reference가 익숙하지 않아 프로그램을 구현과정에서 다양한 고민들을 많이 했다. C로 구현했을 때와는 또 다른 매력이 있는 언어라는 것을 구현하면서 느꼈고, 그동안 소홀히 했던 JAVA를 다시 한번 사용함으로써 다양한 언어를 활용할 수 있는 능력을 기르는 것이 이번 프로젝트에서의 개인적인 목표였다.

2장. 설계/구현 아이디어

설계/구현의 아이디어의 설명은 각 .java 파일의 구조를 설명한 뒤, main 함수의 절차에 따라 어떤 함수들이 어떻게 작동되는지 기술하도록 한다.

우선, Assembler.java 는 다음과 같은 구조로 이루어져있다.



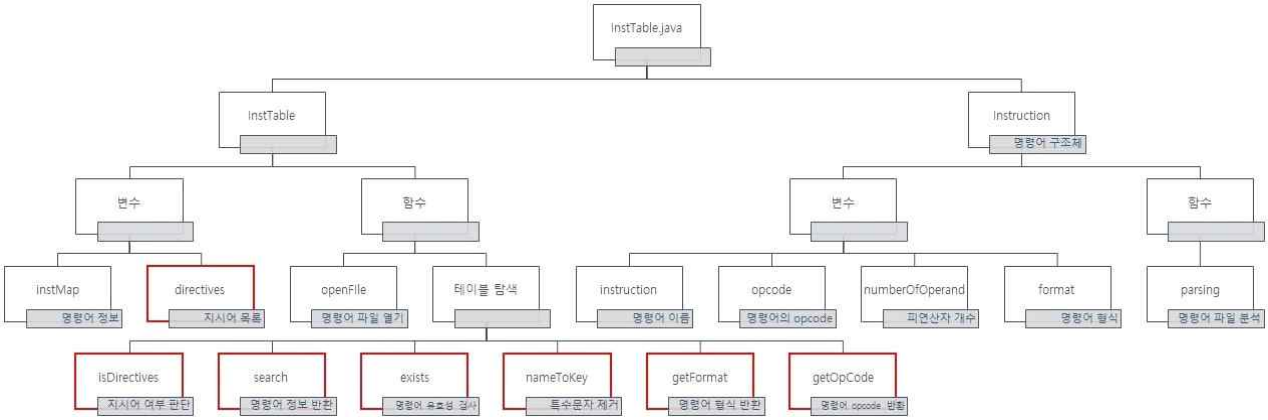
Assembler.java에서 본인이 추가적으로 구현한 함수는 padding과 writeBuftoFile 함수이다.

private String padding(String objectCode, int byteSize)
ㄴ byteSize에 맞게 object code 앞에 0으로 패딩을 넣어준다.
- String objectCode : 패딩이 들어가지 않은 objectCode
- int byteSize : 입력받은 objectCode가 실제로 가져야 하는 바이트의 크기
→ return : padding이 들어간 objectCode (ex. 32023 => 032023)

private String padding(int objectCode_int, int byteSize)
ㄴ 위의 padding 함수를 오버로딩한 함수이다.
int형태의 objectCode를 String형태로 바꾼 뒤 다시 padding 함수를 호출한다.
- int objectCode_int : 패딩이 들어가지 않은 objectCode
- int byteSize : 입력받은 objectCode가 실제로 가져야 하는 바이트의 크기
→ return : padding이 들어간 objectCode (ex. 32023 => 032023)

private void writeBuftoFile(BufferedWriter bufWriter, String Linebuffer)
ㄴ 한 줄의 object Code가 저장된 String을 버퍼를 통해 file에 기록하는 함수이다.
이 과정에서 그 줄의 코드 길이에 대한 정보가 갱신되어 파일에 기록된다.
- BufferedWriter bufWriter : 기록하려는 파일과 연결되어있는 버퍼
- String Linebuffer : 한 줄의 object code가 저장된 문자열 형태의 버퍼

다음은 InstTable.java의 구조이다.



InstTable.java에는 directives라는 문자열 배열과 테이블 탐색에 관련된 함수들을 추가로 구현하였다.

String[] directives : 지시어/예약어가 저장된 String 배열

public boolean isDirectives(String name)
ㄴ 입력받은 명령어가 지시어 또는 예약어에 해당되는지 확인하는 함수
- String name : 판단하려는 명령어의 이름
→ return : 지시어/예약어일 경우 true, 아닐 경우 false

private Instruction search(String name)
ㄴ 입력받은 이름의 명령어에 대한 정보를 반환
- String name : 검색하려는 명령어의 이름
→ return : 명령어에 대한 정보가 담긴 Instruction

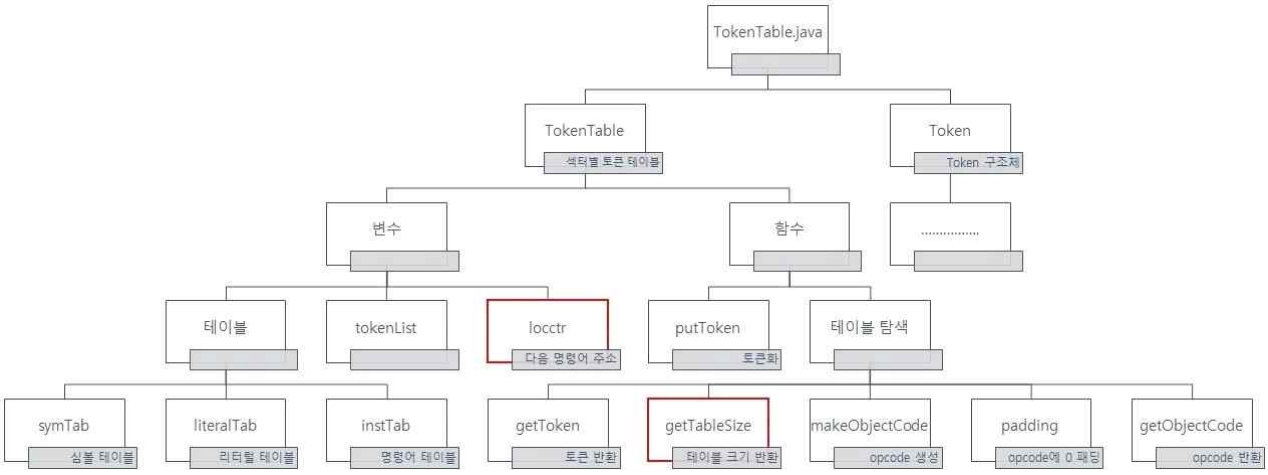
public boolean exists(String name) : 입력받은 이름의 명령어가 유효한 명령어인지 판단해주는 함수
- String name : 찾으려는 명령어의 이름
→ return : 명령어가 instTable에 있을 경우 true, 없을 경우 false

private String nameToKey(String name)
ㄴ 입력받은 명령어에 특수문자를 제거하여 반환
- String name : 변환되기 전 String 형태의 operator
→ return : instTable에서 검색 가능한 형태로 특수문자가 제거된 operator

public int getFormat(String name)
ㄴ 입력받은 이름을 가진 명령어가 몇 형식인지 검색
- String name : 찾으려는 명령어의 이름
→ return : 명령어의 형식
!! Exception : 입력받은 이름을 가진 명령어가 없을 경우

public int getOpCode(String name)
ㄴ 입력받은 이름의 명령어에 해당하는 opcode를 반환한다.
- String name : 찾으려는 명령어의 이름
→ return : 명령어의 opcode
!! Exception : 입력받은 이름을 가진 명령어가 없을 경우

TokenTable.java는 class별로 구조를 설명하려고 한다. 우선 TokenTable class이다.

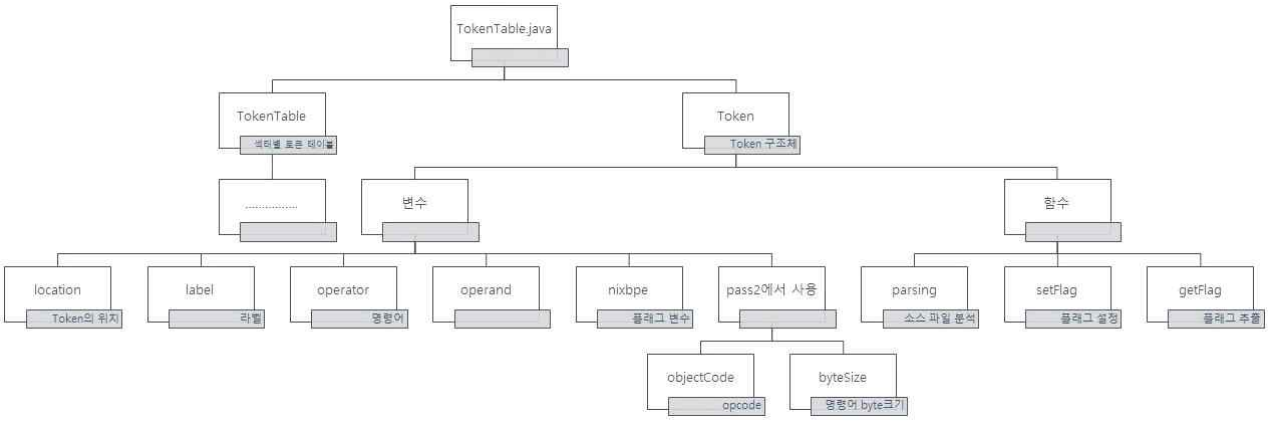


TokenTable class에서는 int형 변수 locctr와 getTableSize 함수를 추가로 구현하였다.

int locctr : 토큰테이블의 location counter로, 토큰 파싱이 모두 끝나면
최종적으로 그 섹션의 프로그램 길이가 저장된다.

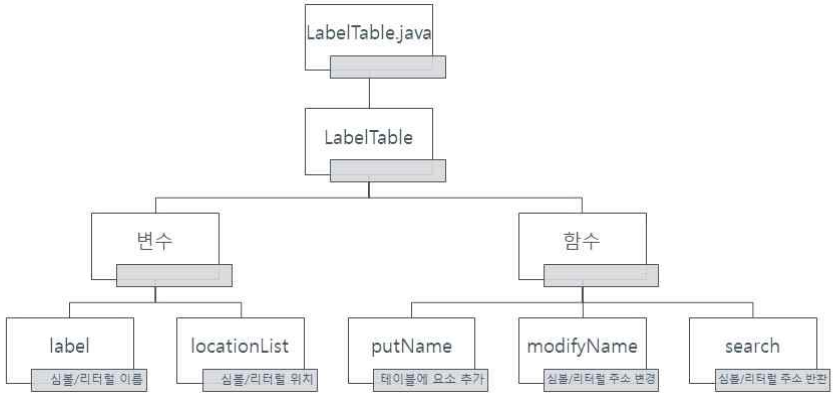
public int getTableSize()
ㄴ tokenList의 크기를 반환한다.
→ return : tokenList의 크기

다음은 TokenTable.java에 있는 Token class의 구조이다.



Token class에는 추가적으로 선언한 함수나 변수가 없고, 이미 명세된 함수와 변수를 최대한 활용하는 방향으로 설계하였다.

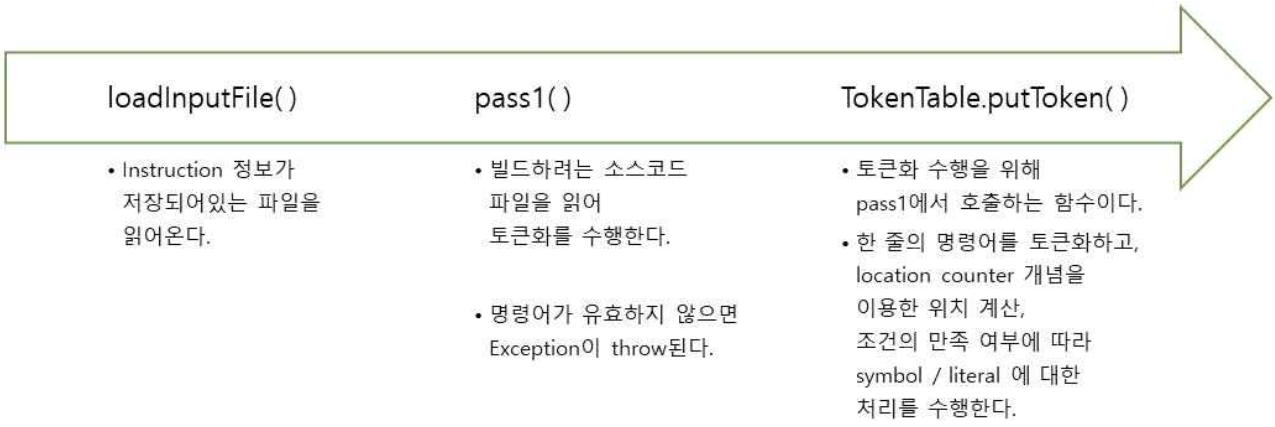
마지막으로 LabelTable.java 의 구조이다. LabelTable은 심볼 테이블과 리터럴 테이블에 사용된다.



Label Table 역시 추가적인 변수나 함수의 선언 없이 기존에 정의된 명세를 바탕으로 코드를 구현하였다.

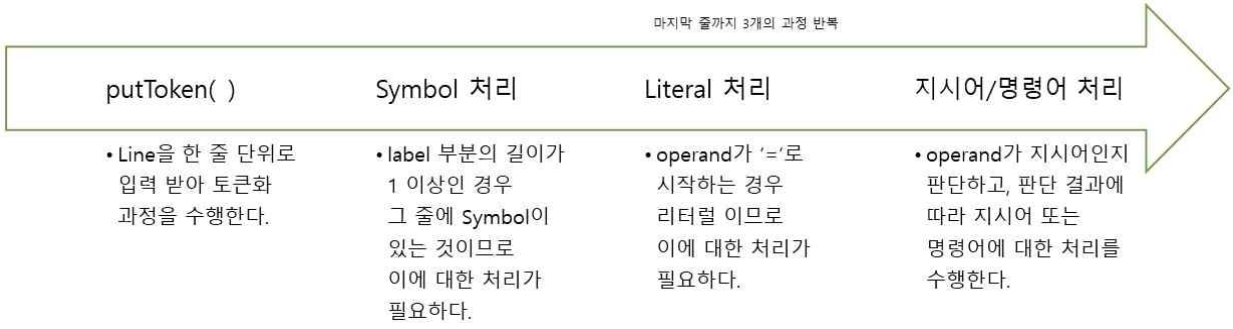
이제부터는 프로그램의 수행 과정의 측면에서 구현된 내용을 설명하려고 한다. main 함수의 진행에 따른 코드의 동작을 간단한 시각화와 설명을 통해 나타내도록 한다.

우선, 명령어가 저장된 데이터 셋과 초기 코드가 저장되어있는 파일을 읽어오는 과정이다.



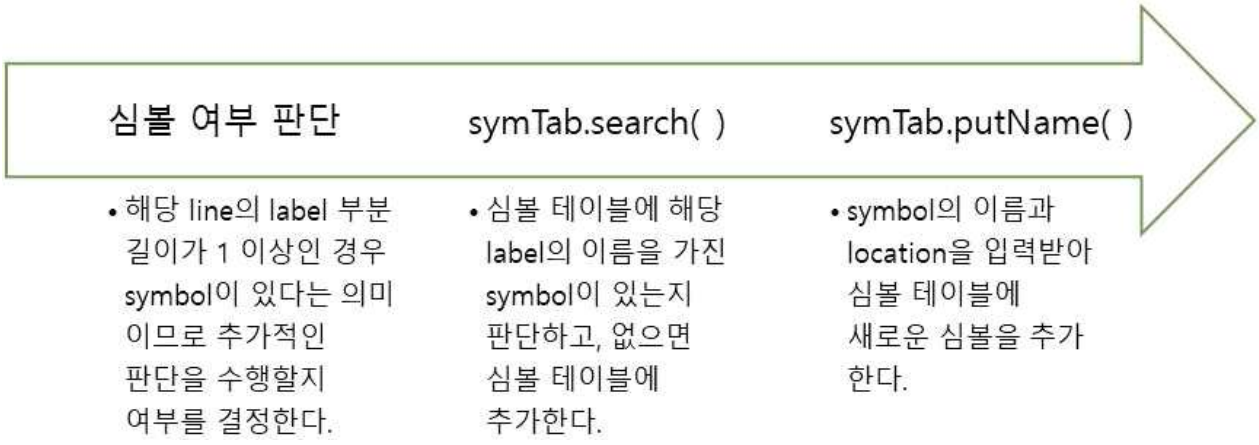
명령어 셋이 저장된 Inst.data 파일은 loadInputFile 함수에서 분석되어 InstTable에 저장되고, 분석하려는 소스코드는 TokenTable class의 putToken 함수에서 토큰화되어 TokenTable에 저장된다.

TokenTable에서는 일반적인 명령어일 때 뿐만 아니라 해당 라인에 심볼이나 리터럴이 있을 경우 그것에 대한 처리도 수행하는데, 이 함수의 수행 과정을 조금 더 자세히 설명하기 위해 시각화를 통해 설명하고자 한다.

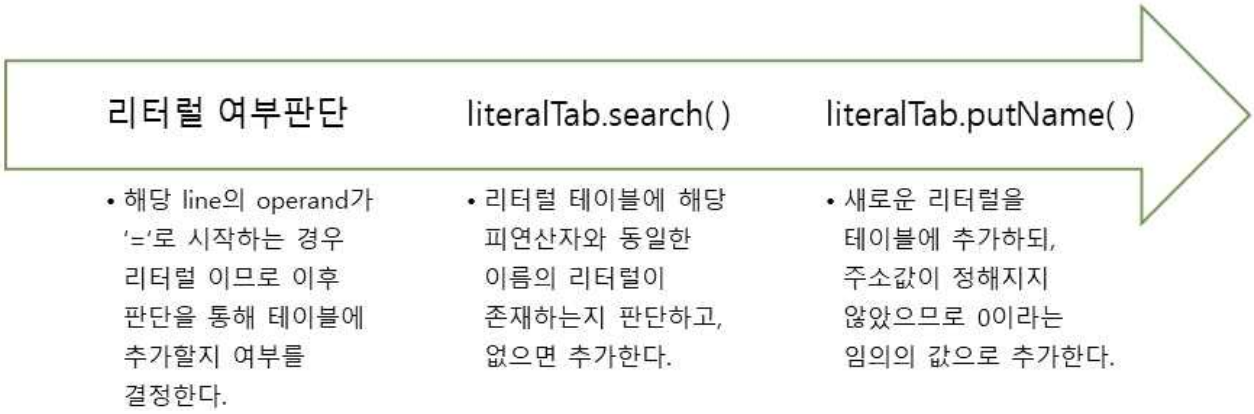


라인을 분석할 때는 위와 같은 기준을 통해 여부를 판단하고, 각각의 조건을 만족하면 그 요소에 대한 처리를 수행한다.

우선, 해당 Line에 Symbol이 있는지를 판단하고 심볼 테이블에 심볼을 추가하는 과정이다.



다음으로, 해당 Line에 리터럴이 있는지를 판단하고 리터럴 테이블에 리터럴을 추가하는 과정이다.



마지막으로 일반적인 operator를 처리하는 과정이다. 이 때, operator가 지시어/예약어인지에 대한 여부를 판단하고, 이 여부에 따라 operator에 대한 처리 방법을 다르게 하여 토큰화를 수행한다.

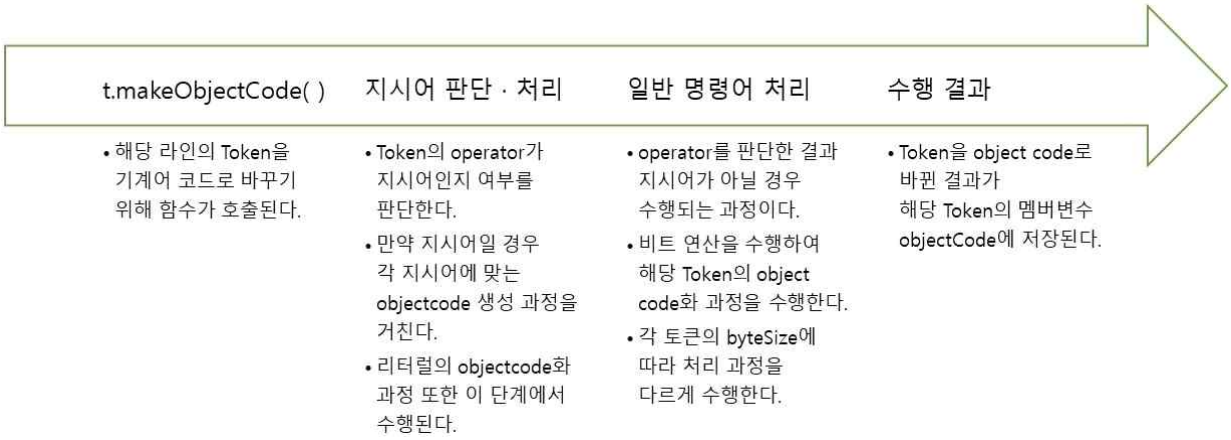
이 때, LTORG나 END를 만나면 그 전까지 선언된 literal들을 모두 TokenTable에 넣어주는 과정을 수행한다. 이 때의 operator는 '*'로 설정하며, 이 operator 역시 directives 배열에 지시어로써 정의되어있다.

pass1을 통해 소스코드의 토큰화가 수행되면 printSymbolTable 함수와 printLiteralTable 함수를 통해 각각의 테이블이 파일에 기록된다.

다음으로 설명하는 내용은 pass2에서 수행되는 과정에 대한 부분이다.

여기서 t는 각 Control Section의 TokenTable을 가리키는 변수로, 각각의 TokenTable을 전체순회하면서 각 토큰에 대해 필요에 따라 objectCode를 만드는 과정을 수행한다.

makeObjectCode 함수는 object code를 만드는데 가장 핵심이 되는 함수로, 그 토큰에 있는 operator가 지시어인지 일반적인 명령어인지에 따라 처리를 다르게 한다.



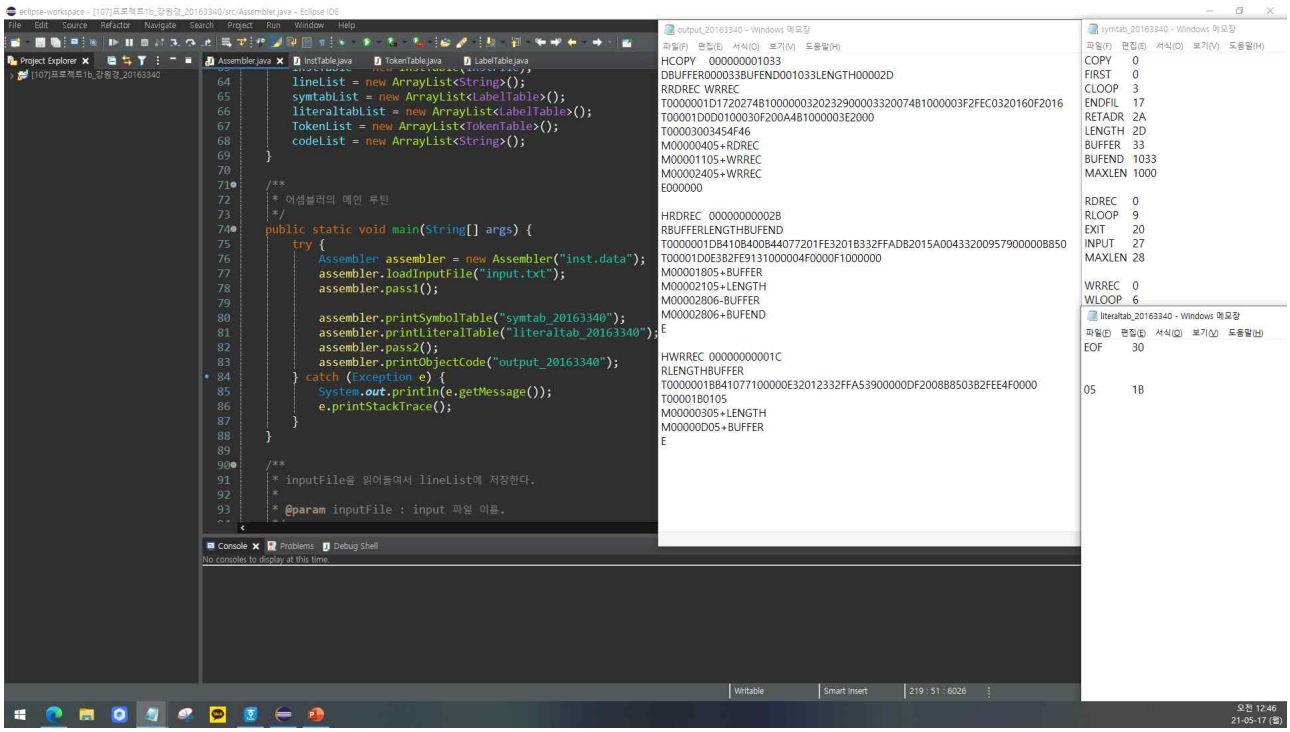
리터럴의 object code화의 경우, pass1에서 정의된 '*'라는 operator가 지시어로써 정의되어있기 때문에 지시어로 분류되어 해당 operator에 맞는 연산 과정을 통해 object code가 생성된다.

pass2까지 모두 수행되면 최종적으로 objectProgram을 출력하는 함수를 거치게 된다.

H, M 레코드는 파일에 기록하기 전 추가적인 수정 작업이 필요 없지만, T 레코드는 길이 정보에 대한 갱신이 필요하기 때문에 다른 레코드들과는 약간 다른 방식으로 파일에 기록하는 과정을 수행한다. writeBuftoFile 함수는 T 레코드를 저장할 때만 사용되는 함수로, 나머지 레코드들은 직접적으로 BufferedWriter에 저장하기 때문에 writeBuftoFile을 호출하지 않고 직접 파일에 기록된다.

위의 일련의 과정을 모두 거치면 Object Program 파일이 생성되고 프로그램은 정상적으로 종료된다.

3장. 수행 결과



코드를 실행할 경우 위의 사진과 같이 심볼 테이블과 리터럴 테이블이 각각의 파일로 저장되어있는 모습을 확인할 수 있다.

또한, 최종 결과 파일에도 프로세스를 거친 오브젝트 프로그램의 코드들이 저장되어있는 것을 확인할 수 있다.

4장. 결론 및 보충할 점.

이번 과제 수행을 통해 본인은 그동안 잠시 멀리했던 JAVA를 다시 한 번 다뤄보고, 절차 지향적인 언어로 작성했던 프로그램을 객체 지향 언어로 변환하면서 설계, 구현적인 측면에서 다양한 고민을 하고 그것을 구현에 옮겨보자는 목표를 달성하였다.

C가 비록 포인터를 통해 강력한 기능을 제공하는 언어이기도 하고, 절차 지향적인 언어이기 때문에 프로그램의 동작 흐름에 따라 코드의 작성만 성공적으로 수행한다면 별다른 고민 없이 코드의 작성이 가능하다는 것은 장점이다. 하지만, C는 언급한대로 프로그램의 흐름대로 코드가 작성되어있기 때문에 코드 작성자가 의도하지 않은 결과가 출력되거나 런타임 오류가 발생할 경우, 그 과정을 하나하나 추적해서 수정해야 하므로 이러한 과정이 까다로운 편이다. 또, 고급 함수나 변수들이 기본적으로 제공되지 않아 사용자가 직접 정의하거나 다른 사람이 작성한 헤더파일등을 추가하여 구현해야 하고, 이러한 함수들은 기본적으로 제공되는 함수보다 의도하지 않은 동작을 할 가능성이 매우 높기 때문에 하나의 프로그램을 구현하는 데에도 상당히 많은 비용이 소모된다.

하지만 JAVA의 경우 C보다는 더 다양한 함수나 변수 타입 등이 정의된 기본 라이브러리를 제공해줌으로써 사용자는 자신이 구현하고자 하는 프로그램에만 집중하여 구현할 수 있게 되고, 이를 통해 구현에 필요한 시간적, 물질적 비용이 매우 많이 감소하게 된다. 또 객체지향적이기 때문에 각 객체별로 기능만 정의된다면 그 기능을 호출하여 사용하면 되기 때문에 반복되는 코드의 작성이 매우 간결해지고, 오류에 대한 수정도 매우 간편해지는 장점이 있다.

하나의 예를 들면, 지난번 C로 작성한 과제에서는 심볼과 리터럴을 처리하는 과정이 pass1이라는 함수 안에 있었고 이 부분을 구현하다 보니 pass1이라는 함수가 150줄이 넘어가게 되었다. 이 때문에 디버깅을 하면서 한 단계씩 코드를 실행할 때 pass1의 모든 과정을 한번씩 방문하면서 코드를 수행해야 하므로 좋지 않은 가독성과 디버깅 중의 시간적 비용은 물론이고, 코드 수정 중에도 내가 올바르게 수정하고 있는 것이 맞는지에 대한 의문이 계속 들었다. 하지만 JAVA의 경우 하나의 기능을 만드는데 여러 객체들에 정의된 메소드들을 호출하여 구현하면 되므로 의도하지 않은 동작을 하는 메소드로 이동하여 그 부분을 호출하면 되기 때문에 가독성과 시간적인 비용 모두 사용자에게 편의성을 제공해줌으로써 효율적인 프로그램 구현이 가능했다.

이렇게 객체지향 언어의 특성을 이용하여 효율적인 코드를 작성하는 목표를 달성하고 여러 프로그램 언어를 다루는 능력을 기른다는 목표는 달성하였으나, 결과적으로 아쉬운 부분이 두 부분 있다.
수업시간에 질문했던 내용의 답변에 따르면 이와 같이 object code가 작성되어도 실제적인 실행에는 큰 문제가 발생하지 않는다고는 하지만, 명세에서 요구하는 완벽한 정답은 아니기 때문에 프로젝트의 결과를 작성하는 지금 아쉬움이 많이 남는 부분이다. 이 부분에 대한 설명을 자세히 설명하도록 한다.

```
HRDREC  000000000002B
RBUFFERLENGTHBUFEND
T0000001DB4108400B44077201FE3201B332FFADB2015A0043320095790000 0B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFFEND
M00002806-BUFFER
E
```

```
HRDREC  000000000002B
RBUFFERLENGTHBUFEND
T0000001DB4108400B44077201FE3201B332FFADB2015A0043320095790000 0B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806-BUFFER
M00002806+BUFEND
E
```

본인의 코드 개선 부분점들 중 첫 번째로 살펴볼 내용은 2번째 Control Section인 RDREC에서 M 레코드에 대한 부분이다. 왼쪽의 텍스트 파일은 이론에 가장 부합하는 Object Program의 작성 결과이고, 오른쪽의 파일은 본인이 작성한 코드의 실행 결과로 생성된 Object Program이다. 모범 답안을 살펴보면 (그림에서의 연녹색 부분) BUFEND의 주소값을 먼저 더한 후 BUFFER의 주소값을 빼지만, 본인이 작성한 프로그램(그림에서의 빨간 부분)은 BUFFER의 주소값을 먼저 빼고 BUFEND의 주소값을 더한다.

```
/* EXTREF 처리 */
else if (t.operator.equals("EXTREF")) {
    ref_ops = t.operand;
```

```
// M 처리
for (Token t : table.tokenList) {
    if (t.operand != null) {
    for (String line_op : t.operand) {
            for (String ext_op : ref_ops) {
                int i = line_op.indexOf(ext_op);
                if (i != -1 && t.location != -1) {
                    if (instTable.exists(t.operator)) {
                        bufWriter.write("M" + padding(t.location + 1, 3));
                        bufWriter.write("05+");
                    } else {
                        bufWriter.write("M" + padding(t.location, 3));
                        if (i > 0 && line_op.charAt(i - 1) == '-')
                            bufWriter.write("06-");
                        else
                            bufWriter.write("06+");
                    }
                    bufWriter.write(ext_op + "\n");
                    bufWriter.flush();
                }
            }
        }
    }
}
```

Assembler.java의 makeObjectOutput 메소드에서 M 레코드를 처리하는 부분을 가져온 부분이다. 본인이 생각하기에, EXTREF에서 operand가 언급된 순서 그대로 순회하기 때문에 발생한 문제로 생각된다. 이 부분은 String 배열이 아니라 Map 방식을 사용하면 해결할 수 있을 것으로 생각된다. Map은 저장된 순서 상관없이 Key값의 정렬 순서로 레코드를 탐색하기 때문에 이러한 의도하지 않은 수행 결과를 줄일 수 있을 것으로 생각된다.

두 번째 본인이 작성한 코드의 개선점은 3번째 Control Section의 Literal에 대한 부분이다.

이 부분은 Control Section의 마지막에서 END를 만났을 때 Literal을 처리하는 부분이다. 모범 답안(왼쪽)은 END를 만났을 때 리터럴을 프로그램 코드에 이어붙히지만 본인이 작성한 코드(오른쪽)는 리터럴을 처리하기 위해 새로운 한 줄을 만들고 리터럴들이 기록되기 시작하는 위치와 리터럴들의 길이, 그리고 object code화가 수행된 코드들을 뒤에 기록해 나간다. 이 과정에 따라 오른쪽의 그림에서

명령어가 담긴 토큰들이 모두 기록된 후, 줄 바꿈을 수행한 뒤 리터럴 정의 시작주소 (00001B), 리터럴들의 길이 (1byte = 01), object code화가 수행된 리터럴 (X'05' → 05)이 추가되었다.

```
  // 리터럴 정의 부분에 대한 처리
else {
  if (t.operator.equals("*")) {
     writeBuftoFile(bufWriter,Linebuffer);
     Linebuffer = "T";
     Linebuffer += padding(t.location, 3);
     Linebuffer += "03";
     Linebuffer += padding(t.objectCode, t.byteSize);
```

리터럴의 정의에 대한 부분이다. 위에서 언급한 문제는 토큰의 operator가 "*"일 때 무조건 줄바꿈을 수행하고 새로운 라인을 만들어내기 때문에 발생한 문제로 판단된다. 이 부분은 본인이 작성한 코드의 토큰 테이블 저장 방법의 절차 상, 수정하기 위해서는 토큰 테이블 저장 방식을 비롯한 프로그램 전반적인 로직의 변경이 필요하기 때문에 우선 구현할 수 있는 최대한의 방법으로 구현하였다.

그 후, 본인이 생각하는 이 프로그램의 개선 방안에 대한 내용이다. Object Program을 만들 때, codeList라는 String type의 ArrayList를 활용하라 되어있는데, 본인은 이 codeList를 활용하는 것이 Token의 objectCode를 각각 불러오는 것보다 더 번거로울 것으로 판단하였다. 이유는, 이 codeList는 전체 프로그램의 object code가 control section 구분 없이 하나의 리스트로 저장되기 때문에, 나중에 control section에 대한 구분이 까다로워지기 때문이다. control section은 결국 tokenTable이 끝났는지의 여부로 판단하기 때문에, codeList를 이용해 object program을 작성하면 control section의 구분을 위해 TokenTable을 참조하고, object code를 가져오기 위해 codeList를 참조하게 되어 두 개의 테이블을 참조하게 되고, 이 과정에서 프로그래밍의 오류나 실수가 발생할 가능성이 높다.

따라서 본인은 codeList를 활용하는 것보다, TokenTable을 범위 기반 for문을 통해 Token을 불러와서 object code를 가져오고 그 토큰 테이블이 끝나면 새로운 control section에 대한 object program을 작성하는 것이 훨씬 효율적이라 판단된다.

이 codeList를 String 형태가 아닌 다른 형태로 바꿔 선언하여 control section별로 구분할 수 있도록 변형을 시키면 되는 것 아니냐는 의문이 생길 수 있는데, 이미 Token에 objectCode라는 변수가 있기 때문에 이 변수를 단순히 codeList라는 배열 리스트에 저장하기 전 임시 저장용 변수로 쓰기에는 너무나도 많은 공간이 낭비된다 생각한다. 따라서 중복된 역할을 수행하는 변수의 선언을 최소화하고 여러 변수를 참조하는 것보다 하나의 테이블에서 여러 가지 기능을 처리한다면 더 효율적인 프로그램의 작성이 가능하지 않을까 하는 개선 방안을 제안해본다.

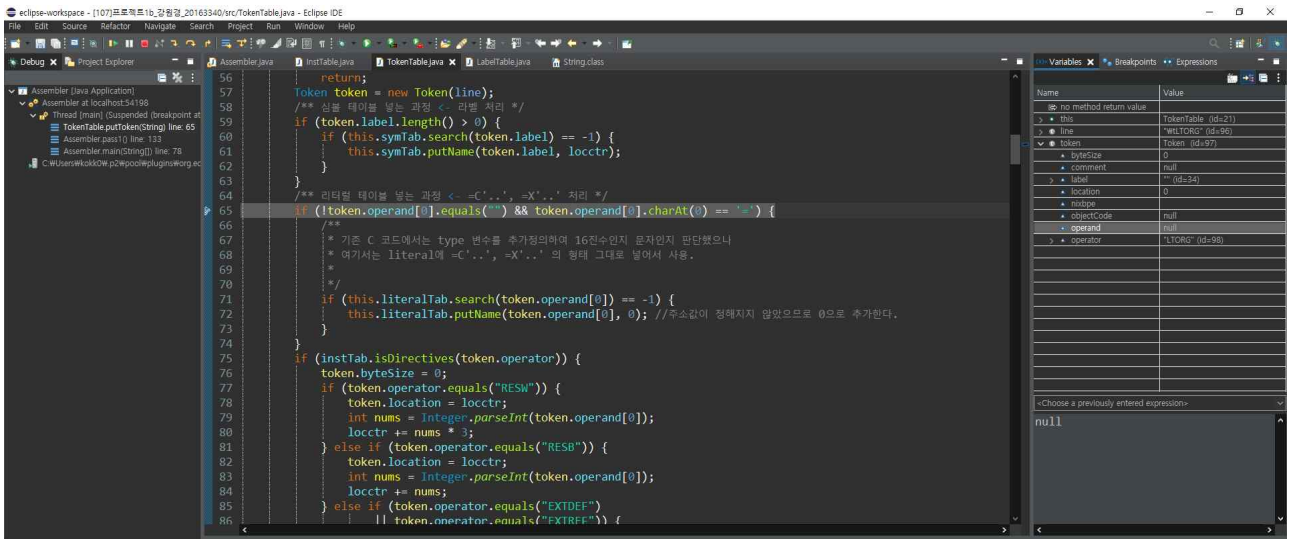
5장. 디버깅

eclipse에서도 디버거를 지원하기 때문에 코드를 작성하고 실행하면서 생각하지 못했던 부분에서 Exception이 발생하거나 의도한 코드 결과가 나오지 않았을 경우 디버거를 통한 디버깅을 수행하였다.

그 중 NullPointerException이 발생했던 상황에서 디버거를 사용했던 예를 들어보려고 한다.

```
Cannot load from object array because "token.operand" is null
java.lang.NullPointerException: Cannot load from object array because "token.operand" is null
        at TokenTable.putToken(TokenTable.java:65)
        at Assembler.pass1(Assembler.java:133)
        at Assembler.main(Assembler.java:78)
```

pass1에서 Token화 과정 중 putToken함수에서 nullpoint Exception이 발생하였고, 디버거를 통해 해당 스택에서의 변수를 조사해보기로 했다.



조사 결과 operator가 LTORG인 line에 해당하는 Token에 operand가 없어 null을 가리키고 있기 때문에 발생한 Exception임을 확인하였고, 해당 라인에는 null이 아닐 경우 실행할 수 있도록 조건을 추가하였다.

6장. 소스코드(+주석)

Assembler.java

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

/**
 * 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
상관 없음)
 *
 * + 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;

    int sector = -1;

    /**
     * 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>();
    }

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

            assembler.printSymbolTable("symtab_20163340");
            assembler.printLiteralTable("literaltab_20163340");
            assembler.pass2();
            assembler.printObjectCode("output_20163340");
        } catch (Exception e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }

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

```
    private void loadInputFile(String inputFile) { // init_input_file
        try {
            File file = new File(inputFile);
            FileReader filereader = new FileReader(file);
            BufferedReader bufReader = new BufferedReader(filereader);
            String line = "";
            while ((line = bufReader.readLine()) != null) {
                lineList.add(line);
            }
            bufReader.close();
        } catch (FileNotFoundException e) {
            System.out.println(e);
        } catch (IOException e) {
            System.out.println(e);
        }
    }

    /**
     * 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.
     * @throws Exception
     */

    private void pass1() throws Exception {
        for (String line : lineList) {
            if (line.matches(".*(START|CSECT).*")) { // 새로 섹터 시작되면 새로운 테이블들 생성
                sector++;
                symtabList.add(new LabelTable());
                literaltabList.add(new LabelTable());
                TokenList.add(new TokenTable(symtabList.get(sector), literaltabList.get(sector),
instTable));
            }
            TokenList.get(sector).putToken(line);
        }
    }

    /**
     * 작성된 SymbolTable들을 출력형태에 맞게 출력한다.
     *
     * @param fileName : 저장되는 파일 이름
     */
    private void printSymbolTable(String fileName) {
        try {
            File file = new File(fileName);
            FileWriter filewriter = new FileWriter(file);
            BufferedWriter bufWriter = new BufferedWriter(filewriter);

            for (int i = 0; i <= sector; i++) {
                int size = TokenList.get(i).symTab.label.size();
                for (int j = 0; j < size; j++) {
                    bufWriter.write(TokenList.get(i).symTab.label.get(j));
                    bufWriter.write('₩t');

bufWriter.write(Integer.toHexString(TokenList.get(i).symTab.locationList.get(j)).toUpperCase());
                    bufWriter.write('₩n');
                    bufWriter.flush();
                }
                bufWriter.write('₩n');
                bufWriter.flush();
            }
            bufWriter.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 작성된 LiteralTable들을 출력형태에 맞게 출력한다.
     *
     * @param fileName : 저장되는 파일 이름
     */
    private void printLiteralTable(String fileName) {
        try {
            File file = new File(fileName);
            FileWriter filewriter = new FileWriter(file);
            BufferedWriter bufWriter = new BufferedWriter(filewriter);

            for (int i = 0; i <= sector; i++) {
                int size = TokenList.get(i).literalTab.label.size();
                for (int j = 0; j < size; j++) {
                    String literal_temp = TokenList.get(i).literalTab.label.get(j);
                    bufWriter.write(literal_temp.substring(3, literal_temp.length() - 1));
                    bufWriter.write('₩t');

bufWriter.write(Integer.toHexString(TokenList.get(i).literalTab.locationList.get(j)).toUpperCase());
                    bufWriter.write('₩n');
                    bufWriter.flush();
                }
                bufWriter.write('₩n');
                bufWriter.flush();
            }
            bufWriter.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }


    /**
     * pass2 과정을 수행한다.
     *
     * 1) 분석된 내용을 바탕으로 object code를 생성하여 codeList에 저장.
```

```
     *
     * @throws Exception
     */
    private void pass2() throws Exception {
        for (TokenTable t : TokenList) {
            for (int i = 0; i < t.getTableSize(); i++) {
                t.makeObjectCode(i);
                codeList.add(t.getObjectCode(i));
            }
        }
    }


    /**
     * 작성된 codeList를 출력형태에 맞게 출력한다.
     * --> 실제 구현 : codeList를 사용하지 않고, Token의 멤버변수인 objectCode 사용.
     *          └> 이유 : codeList로 ObjectProgram 작성 시 Control Section에 대한 구분 과정이
복잡해짐.
     * @param fileName : 저장되는 파일 이름
     */
    private void printObjectCode(String fileName) {
        try {
            File file = new File(fileName);
            FileWriter filewriter = new FileWriter(file);
            BufferedWriter bufWriter = new BufferedWriter(filewriter);

            int cur_buf_length = 0; // 버퍼 길이 확인용 변수. 한줄에는 최대 68개 문자까지 가능.
            int cur_sector_num = 0;
            /** Section별 table 가져오기 */
            for (TokenTable table : TokenList) {
                /* Control Section 시작할 때 Header 정보 기록 */
                bufWriter.write('H');
                // Control Section 이름
                bufWriter.write(table.symTab.label.get(0) + "₩t");
                // Control Section 시작 주소
                bufWriter.write(padding(table.symTab.locationList.get(0), 3));
                // 프로그램 길이
                bufWriter.write(padding(table.locctr, 3) + "₩n");
                bufWriter.flush();

                String[] ref_ops = new String[3];  //EXTREF의 operand를 저장하는 String 배열

                /* EXTDEF, EXTREF 처리 */
                for (Token t : table.tokenList) {
                    // 특수한 operator에 대한 처리
                    if (t.byteSize == 0) {
                        /* EXTDEF 처리 */
                        if (t.operator.equals("EXTDEF")) {
                            bufWriter.write('D');
                            cur_buf_length = 1;
                            for (String s : t.operand) {
                                if (cur_buf_length >= 57) {
                                    bufWriter.write('₩n');
                                    bufWriter.flush();
                                    bufWriter.write('D');
                                    cur_buf_length = 1;
                                }
                                bufWriter.write(s);
                                for (int i = s.length(); i < 6; i++)
                                    bufWriter.write(' ');
                                int addr = table.symTab.search(s);
                                bufWriter.write(padding(addr, 3));
                                cur_buf_length += s.length() + 6;
                            }
                            bufWriter.write('₩n');
                            bufWriter.flush();
                            cur_buf_length = 0;
                        }
                        /* EXTREF 처리 */
                        else if (t.operator.equals("EXTREF")) {
                            ref_ops = t.operand;
                            bufWriter.write('R');
                            cur_buf_length = 1;
                            for (String s : t.operand) {
                                if (cur_buf_length >= 57) {
                                    bufWriter.write('₩n');
                                    bufWriter.flush();
                                    bufWriter.write('R');
                                    cur_buf_length = 1;
                                }
                                bufWriter.write(s);
                                for (int i = s.length(); i < 6; i++)
                                    bufWriter.write(' ');
                                cur_buf_length += 6;
                            }
                            bufWriter.write('₩n');
                            bufWriter.flush();
                            cur_buf_length = 0;
                            break;
                        }

                    }

                }

                boolean is_newline = true;
                /** 파일 쓰기 시 라인의 길이 정보 수정때문에 String 형태의 버퍼에 1차적으로 쓰고
                 * 기록 전 subString을 활용한 줄 길이에 대한 내용 갱신 후 버퍼를 통해 파일 기록*/

                String Linebuffer = new String();

                /** T 부분 처리 */
                for (Token t : table.tokenList) {
                    if (is_newline) {
                        Linebuffer += 'T';
                        Linebuffer += padding(t.location, 3);
                        Linebuffer += "00";
                        is_newline = false;
                    }
                    if (!instTable.isDirectives(t.operator)) {
                        if (Linebuffer.length() + t.byteSize * 2 > 69) {
                            writeBuftoFile(bufWriter,Linebuffer);
```

```
            Linebuffer = "T";
            Linebuffer += padding(t.location, 3);
            Linebuffer += "00";
        }
        Linebuffer += padding(t.objectCode, t.byteSize);
        // END일 때의 처리
    } else {
        if (t.operator.equals("*")) {
            writeBuftoFile(bufWriter,Linebuffer);
            Linebuffer = "T";
            Linebuffer += padding(t.location, 3);
            Linebuffer += "03";
            Linebuffer += padding(t.objectCode, t.byteSize);
        }
        else if(t.operator.equals("WORD")||t.operator.equals("BYTE")) {
            Linebuffer+=t.objectCode;
        }
    }
}
writeBuftoFile(bufWriter,Linebuffer);

// M 처리
for (Token t : table.tokenList) {
    if (t.operand != null) {
    for (String line_op : t.operand) {
            for (String ext_op : ref_ops) {
                int i = line_op.indexOf(ext_op);
                if (i != -1 && t.location != -1) {
                    if (instTable.exists(t.operator)) {
                        bufWriter.write("M" + padding(t.location + 1, 3));
                        bufWriter.write("05+");
                    } else {
                        bufWriter.write("M" + padding(t.location, 3));
                        if (i > 0 && line_op.charAt(i - 1) == '-')
                            bufWriter.write("06-");
                        else
                            bufWriter.write("06+");
                    }
                    bufWriter.write(ext_op + "₩n");
                    bufWriter.flush();
                }
            }
        }
    }

}

// E 처리
if (cur_sector_num == 0) {
    bufWriter.write("E");
    bufWriter.write(padding(table.symTab.locationList.get(0), 3));
    bufWriter.write("₩n₩n");
} else
    bufWriter.write("E₩n₩n");
bufWriter.flush();
cur_sector_num++;
        }
        bufWriter.close();
    } catch (Exception e) {
        e.printStackTrace();
    }

}

/**
 * byteSize에 맞도록 objectCode의 앞에 0으로 패딩을 넣어준다.
 *
 * @param objectCode : 저장된 주소값
 * @param byteSize   : objectCode가 가져야하는 바이트 크기
 * @return byteSize에 맞게 0으로 패딩을 넣은 objectCode
 */
private String padding(String objectCode, int byteSize) {
    int len = objectCode.length();
    String ans = new String();
    while (len < byteSize * 2) {
        ans += "0";
        len++;
    }
    ans += objectCode;
    return ans.toUpperCase();
}

/**
 * int 형태의 objectCode를 String으로 바꾼 뒤 다시 padding 함수를 호출한다.
 *
 * @param objectCode_int : integer 형태의 objectCode
 * @param byteSize   : objectCode가 가져야하는 바이트 크기
 * @return byteSize에 맞게 0으로 패딩을 넣은 objectCode
 */
private String padding(int objectCode_int, int byteSize) {
    String temp = Integer.toHexString(objectCode_int);
    return padding(temp, byteSize);
}

/**
 * 한 줄의 object code가 저장된 문자열을 버퍼를 통해 파일에 기록하는 함수이다.
 *
 * @param bufWriter : 기록하려는 파일과 연결된 버퍼
 * @param Linebuffer: 한 줄의 object code가 저장된 문자열 형태의 버퍼
 */
private void writeBuftoFile(BufferedWriter bufWriter,String Linebuffer) throws IOException {
    int line_len = (Linebuffer.length() - 9) / 2;
    bufWriter.write(Linebuffer.substring(0, 7)); // 0~6까지 버퍼 넣음
    bufWriter.write(padding(line_len, 1)); // 7~8까지는 String 길이 넣음 (길이 정보 갱신)
    bufWriter.write(Linebuffer.substring(9)); // 9~끝까지 버퍼 넣음
    bufWriter.write('₩n');
    bufWriter.flush();
}
}
```

InstTable.java

```
import java.util.HashMap;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

/**
 * 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;

    public static final int MAX_INST = 256;
    final String[] directives = { "START", "END", "BYTE", "WORD", "RESB", "RESW", "EXTDEF", "EXTREF", "LTORG", "*",
            "EQU", "CSECT" }; // 지시어/예약어가 저장된 String 배열

    /**
     * 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 {
            File file = new File(fileName);
            FileReader filereader = new FileReader(file);
            BufferedReader bufReader = new BufferedReader(filereader);
            String line = "";
            while ((line = bufReader.readLine()) != null) {
                Instruction inst = new Instruction(line);
                instMap.put(inst.instruction, inst);
            }
            bufReader.close();
        } catch (FileNotFoundException e) {
            System.out.println(e);
        } catch (IOException e) {
            System.out.println(e);
        }
    }

    /**
     * 입력받은 명령어가 지시어 또는 예약어(ex. 선언자 '*')인지 확인하는 함수
     *
     * @param name : 판단하려는 명령어 이름
     * @return : 지시어/선언자일 경우 true, 아닐 경우 false
     */
    public boolean isDirectives(String name) {
        for (String s : directives) {
            if (s.equals(name))
                return true;
        }
        return false;
    }

    /**
     * 입력받은 이름의 명령어에 대한 정보를 반환
     *
     * @param name : 검색하려고 하는 명령어의 이름
     * @return : 명령어에 대한 정보가 담긴 Instruction
     */
    private Instruction search(String name) {
        return instMap.get(nameToKey(name));
    }

    /**
     * 입력받은 이름의 명령어가 instTable에 있는지 판단해주는 함수.
     *
     * @param name : instTable에 있는지 판단하려는 명령어
     * @return : 명령어 테이블에 있는경우 true, 없는경우 false.
     */
    public boolean exists(String name) {
        return instMap.containsKey(nameToKey(name));
    }

    /**
     * 입력받은 operator가 어떤 형태이던지 간에 instTable의 검색에 사용 가능한 형태로 바꿔주는 함수.
     *
     * @param name : 변환되기 전 String 형태의 operator
     * @return : instTable에서 검색 가능한 형태로 변환된 String 형태의 operator.
     */
    private String nameToKey(String name) {
        return name.replace("+", "");
    }

    /**
     * 입력받은 이름을 가진 명령어가 몇 형식인지 찾아주는 함수.
     *
     * @param name : 찾으려는 명령어의 이름
     * @return : 명령어의 format
```

```
    * @exception : 입력받은 이름을 가진 명령어가 없을 경우 Exception throw.
    */
   public int getFormat(String name) throws Exception {
      if (name.charAt(0) == '+')
         return 4;
      if (!exists(name))
         throw new Exception("없는 명령어입니다.");
      return search(name).format;
   }

   /**
    * 입력받은 이름을 가진 명령어가 있다면 그 명령어의 opcode를, 아닐 경우 Exception을
throw한다.
    *
    * @param name : 찾으려는 명령어의 이름
    * @return : 명령어의 opcode
    * @exception : 입력받은 이름을 가진 명령어가 없다면 오류문과 함께 Exception이
Throw됨.
    */
   public int getOpCode(String name) throws Exception {
      if (!exists(name))
         throw new Exception("없는 명령어입니다.");
      return search(name).opcode;
   }
}

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

   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[] line_seperate = line.split("₩t");
      instruction = line_seperate[0];
      format = Integer.parseInt(line_seperate[1]);
      opcode = Integer.parseInt(line_seperate[2], 16);
      numberOfOperand = Integer.parseInt(line_seperate[3]);
   }

}
```

TokenTable.java

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

    /** 현재 Control Section의 location counter.
     * 해당 Control Section의 parsing이 끝나면 최종적으로
     * 그 섹션의 프로그램 길이가 저장된다. */
    int locctr = 0;

    /**
     * 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;
    }

    /**
     * 일반 문자열을 받아서 Token단위로 분리시켜 tokenList에 추가한다.
     *
     * @param line : 분리되지 않은 일반 문자열
     * @throws Exception
     */
    public void putToken(String line) throws Exception {
        if (line.charAt(0) == '.')
            return;
        Token token = new Token(line);
        /** 심볼 테이블 넣는 과정 <- 라벨 처리 */
        if (token.label.length() > 0) {
            if (this.symTab.search(token.label) == -1) {
                this.symTab.putName(token.label, locctr);
            }
        }
        /** 리터럴 테이블 넣는 과정 <- =C'..', =X'..' 처리 */
        if (token.operand != null && !token.operand[0].equals("")
                && token.operand[0].charAt(0) == '=') {
            /**
             * 기존 C 코드에서는 type 변수를 추가정의하여 16진수인지 문자인지 판단했으나
             * 여기서는 literal에 =C'..', =X'..' 의 형태 그대로 넣어서 사용.
             *
             */
            if (this.literalTab.search(token.operand[0]) == -1) {
                this.literalTab.putName(token.operand[0], 0); //주소값이 정해지지 않았으므로 0으로 추가한다.
            }
        }
        if (instTab.isDirectives(token.operator)) {
            token.byteSize = 0;
            if (token.operator.equals("RESW")) {
                token.location = locctr;
                int nums = Integer.parseInt(token.operand[0]);
                locctr += nums * 3;
            } else if (token.operator.equals("RESB")) {
                token.location = locctr;
                int nums = Integer.parseInt(token.operand[0]);
                locctr += nums;
            } else if (token.operator.equals("EXTDEF")
                    || token.operator.equals("EXTREF")) {
                token.location = -1; // 문제 발생 여지 있음.
            } else if (token.operator.equals("LTORG")
                    || token.operator.equals("END")) {
                token.location = -1;
                for (String s : this.literalTab.label) {
                    String literal_temp = "₩t*₩t";
                    literal_temp += s;
                    putToken(literal_temp);
                }
            } else if (token.operator.equals("*")) {
                int len = token.operand[0].length() - 4;
                if (token.operand[0].charAt(1) == 'X')
                    len /= 2;
                token.byteSize = len;
                token.location = locctr;
                this.literalTab.modifyName(token.operand[0], locctr);
```

```
            locctr += len;
        } else if (token.operator.equals("CSECT")
                || token.operator.equals("START")) {
            token.location = locctr;

        } else if (token.operator.equals("EQU")) {
            int templocctr=locctr;
            if (token.operand[0].charAt(0) != '*') {
                char op = '₩0';
                for (int i = 0; i < token.operand[0].length(); i++) {
                    if (token.operand[0].charAt(i) == '+' || token.operand[0].charAt(i) == '-') {
                        op = token.operand[0].charAt(i);
                    }
                }
                if (op != '₩0') {
                    String[] operation = token.operand[0].split("₩₩+|₩₩-");
                    int v1 = this.symTab.search(operation[0]);
                    int v2 = this.symTab.search(operation[1]);
                    if (op == '+')
                        templocctr = v1 + v2;
                    else
                        templocctr = v1 - v2;
                }
            }
            token.location = templocctr;
            this.symTab.modifyName(token.label, templocctr);
        } else if (token.operator.equals("BYTE")) {
            token.location = locctr;
            if (token.operand[0].charAt(0) == 'C') {
                locctr += token.operand[0].length() - 3;
            } else
                locctr += (token.operand[0].length() - 3) / 2;
        } else if (token.operator.equals("WORD")) {
            token.location = locctr;
            locctr += 3;
        }
    } else {
        token.byteSize = instTab.getFormat(token.operator);
        token.location = locctr;
        locctr += token.byteSize;
    }

    tokenList.add(token);
}

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

/**
 * tokenList의 크기를 리턴한다.
 *
 * @return : tokenList의 크기
 */
public int getTableSize() {
    return tokenList.size();
}

/**
 * Pass2 과정에서 사용한다. instruction table, symbol table 등을 참조하여 objectcode를 생성하고, 이를
 * 저장한다.
 *
 * @param index
 * @throws Exception
 */
public void makeObjectCode(int index) throws Exception {
    Token t = getToken(index);
    int object = 0; // Integer.toHexString으로 바꾸기 전 비트연산용
    if (t.operator == null && t.operator == "")
        return;
    /* 지시어에 대한 처리 시작 */
    if (this.instTab.isDirectives(t.operator)) {
        if (t.operator.equals("BYTE")) {
            String temp = t.operand[0].substring(2, t.operand[0].length() - 1);
            t.objectCode = temp;
            t.byteSize = temp.length() / 2;
        } else if (t.operator.equals("WORD")) {
            t.byteSize = 3;
            // operand가 숫자일 때 그대로 넣어주면 됨.
            if (t.operand[0].charAt(0) >= '0' && t.operand[0].charAt(0) <= '9')
                t.objectCode = t.operand[0];
            // 문자일 경우
            else {
                char op = '₩0';
                for (int i = 0; i < t.operand[0].length(); i++) {
                    if (t.operand[0].charAt(i) == '+' || t.operand[0].charAt(i) == '-') {
                        op = t.operand[0].charAt(i);
                    }
                }
                String[] operation = t.operand[0].split("₩₩+|₩₩-");
                int v1 = this.symTab.search(operation[0]);
                if (v1 == -1)
                    v1 = 0;
                int v2 = this.symTab.search(operation[1]);
                if (v2 == -1)
                    v2 = 0;
                if (op == '+')
                    object = v1 + v2;
                else
                    object = v1 - v2;
                String temp = Integer.toHexString(object).toUpperCase();
                t.objectCode = padding(temp, t.byteSize);
            }
        } else if (t.operator.equals("*")) {
            if (t.operand[0].charAt(1) == 'C') {
```

```
            for (int j = 3; j < t.operand[0].length() - 1; j++) {
                object <<= 8;
                object |= t.operand[0].charAt(j);
            }
            t.objectCode = Integer.toHexString(object).toUpperCase();
        } else if (t.operand[0].charAt(1) == 'X') {
            String temp = t.operand[0].substring(3, t.operand[0].length() - 1);
            t.objectCode = temp;
        }
    }
}
/* 일반 명령어의 처리 */
else {
    object = this.instTab.getOpCode(t.operator);

    switch (t.byteSize) {
    case 1:
        break;
    case 2:
        for (int j = 0; j < t.operand.length; j++) {
            object <<= 4;
            if (j == 1 && (t.operand[j] == "" || t.operand[j] == null))
                object |= 0;
            else if (t.operand[j].equals("SW"))
                object |= 9;
            else if (t.operand[j].equals("PC"))
                object |= 8;
            else if (t.operand[j].equals("F"))
                object |= 6;
            else if (t.operand[j].equals("T"))
                object |= 5;
            else if (t.operand[j].equals("S"))
                object |= 4;
            else if (t.operand[j].equals("B"))
                object |= 3;
            else if (t.operand[j].equals("L"))
                object |= 2;
            else if (t.operand[j].equals("X"))
                object |= 1;
            else if (t.operand[j].equals("A"))
                object |= 0;
        }
        if (t.operand.length == 1)
            object <<= 4;
        break;
    case 3:
        object <<= 4;
        if (!t.operand[0].equals("")) { // null 포인터 exception 방지용.
            /* n, i flag setting */
            if (t.operand[0].charAt(0) == '@')
                t.setFlag(nFlag, 1);
            else if (t.operand[0].charAt(0) == '#')
                t.setFlag(iFlag, 1);
            else {
                t.setFlag(nFlag, 1);
                t.setFlag(iFlag, 1);
            }
            /* X flag setting */
            for (String op_iter : t.operand) {
                if (op_iter.equals("X")) {
                    t.setFlag(xFlag, 1);
                    break;
                }
            }

            if (!t.operand[0].equals("") && t.operand[0] != null) { // operand가 있는 경우
                if (t.operand[0].charAt(0) == '#') {// immediate addressing
                    object |= t.nixbpe;
                    object <<= 12;
                    String tmp = t.operand[0].substring(1);
                    object |= Integer.parseInt(tmp);
                } else {
                    t.setFlag(pFlag, 1);
                    object |= t.nixbpe;
                    object <<= 12;
                    int pc = t.location + t.byteSize;
                    int target_addr;

                    if (t.operand[0].charAt(0) == '=') // literal일 경우 literal tab 찾아야 함
                        target_addr = this.literalTab.search(t.operand[0]);
                    else if (t.operand[0].charAt(0) == '@')// Indirect addressing인 경우 operand 맨
앞의 @ 빼고 찾아야함.
                        target_addr = this.symTab.search(t.operand[0].substring(1));
                    else // Simple addressing인 경우 그냥 symbol table 찾으면 됨.
                        target_addr = this.symTab.search(t.operand[0]);

                    if (target_addr == -1)
                        target_addr = 0;
                    int disp = target_addr - pc;
                    disp &= 0xFFF;
                    object |= disp;
                }
            } else { // operand가 없는 경우. b,p,e flag 모두 0이고 disp도 없으므로 그냥 더함.
                object |= t.nixbpe;
                object <<= 12;
            }
            break;
        case 4:
            object <<= 4;
            /*4형식 명령어(e=1) 특성상 simple addressing이고(n,i=1), base나 pc를 참조할 수
없기때문에 (→주소 모름 || b,p=0)
             * n,i,e 플래그를 우선적으로 1로 세팅하고, x는 사용 여부를 판단하여 세팅한다.
             */
            t.setFlag(nFlag, 1);
            t.setFlag(iFlag, 1);
            t.setFlag(eFlag, 1);
            for (String op_iter : t.operand) {
                if (op_iter.equals("X")) {
```

```
                t.setFlag(xFlag, 1);
                break;
              }
            }
            object |= t.nixbpe;
            object <<= 20;
            int target_addr = this.symTab.search(t.operand[0]);
            if (target_addr == -1)
              target_addr = 0;
            target_addr &= 0xFFFFF;
            object |= target_addr;
          } // end of switch
          t.objectCode = Integer.toHexString(object).toUpperCase();
          t.objectCode = padding(t.objectCode, t.byteSize);
        }
    }

    /**
     * objectCode의 앞에 0으로 패딩을 넣어준다.
     *
     * @param objectCode : 저장된 주소값
     * @param byteSize   : objectCode가 가져야하는 바이트 크기
     * @return byteSize에 맞게 0으로 패딩을 넣은 objectCode
     */
    private String padding(String objectCode, int byteSize) {
        int len = objectCode.length();
        String ans = new String();
        while (len < byteSize * 2) {
            ans += "0";
            len++;
        }
        ans += objectCode;
        return ans.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; // <<- C 프로그램에서 locctr_table 역할
    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[] line_temp = line.split("₩t");
        switch (line_temp.length) {
        case 4:
            comment = line_temp[3];
        case 3:
            operand = line_temp[2].split(",");
        case 2:
            operator = line_temp[1];
        case 1:
            label = line_temp[0];
        }

    }

    /**
     * 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;
    }
```

```
    /**
     * 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;
    }
}
```

LabelTable.java

```java
import java.util.ArrayList;

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


    /**
     * 새로운 symbol과 literal을 table에 추가한다.
     *
     * @param label    : 새로 추가되는 symbol 혹은 literal의 label
     * @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
     *                   modifyName()을 통해서 이루어져야 한다.
     */
    public void putName(String label, int location) {
        if (search(label) == -1) {
            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 label, int newLocation) {
        // TODO
        /* 1차적으로 넣기만 하고 LTORG / END 만났을 때 실제 값으로 바꿔서 위치 설정. */
        int index = this.label.indexOf(label);
        if (index != -1)
            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 = -1;
        int index = this.label.indexOf(label);
        if (index != -1)
            address = this.locationList.get(index);
        return address;
    }

}
```