숭실대학교 Soongsil University

프로그램 보고서

나는 숭실대학교 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 2025
프로젝트 명	Project #1b - ControlSection 방식의 어셈블러 구현하기
교과목 교수	최 재 영
제출인	컴퓨터학부(학과) 학번: 20232872 성명: 김도원(출석번호:102)
제출일	2025년 5월 9일

차 례

1장 동기/목적

SIC/XE 어셈블러를 본격적으로 개발한다. ControlSection 방식의 SIC/XE 소스를 분석해서 심볼 테이블과 리터럴 테이블을 파일로 출력하고 소스를 Object Code로 변환하는 프로그램을 구현한다. 이 프로젝트를 통해서 SIC/XE 어셈블러의 동작을 이해한다. 프로젝트 구현을 Java로 함으로써 객체 지향 프로그래밍과 자바 프로그래밍을 학습하기까지 한다.

2장 설계/구현 아이디어

우선 SIC/XE 어셈블리 소스를 읽고 파싱하는 작업을 시행한다. 해당 파싱과정에서 opcode, 심볼, 피연산자, 주석, nixbpe를 추출해서 token 구조체에 저장한다. 이 프로세스가 실행되는 pass1에서는 추가로 Section을 구분하고 리터럴 테이블과 심볼 테이블의 작성도 수행한다. 그다음 pass2를 실행하면서 pass1에서 얻은 token, literal, symbol 정보들을 바탕으로 token을 순회하면서 Section 별 object code를 생성한다. 피연산자 값을 object code로 변환할 때 심볼 테이블에서 심볼을 찾아 EXTREF로 선언됐으면은 관련 정보를 modification record로 추가한다. object code가 만들어졌으면 output file에 Section 별로 object code를 출력한다.

설계/구현한 클래스는 총 10개이며 각각의 역할은 다음과 같다.

Assembler : SIC/XE Assembler 프로그램의 메인루틴을 실행한다. 어셈블리 코드를 읽어서 토큰으로 파싱하고 명령어 테이블, 심볼 테이블, 리터럴 테이블을 작성하고 object code를 생성하는 등의 pass1, pass2 과정을 실행한다.

LiteralTable : 리터럴 테이블을 관리하는 클래스다. 리터럴 상수의 값, 위치, 길이를 저장하고 임시 저장돼있던 리터럴을 메모리에 배치하며 위치를 갱신한다. 각 리터럴의 object code 생성을 지원하고, 테이블 검색 기능을 제공한다.

SymbolTable : 심볼 테이블을 다루는 클래스다. 심볼(label)과 해당 주소(location)를 저장하고 검색한다. 외부 참조 심볼(EXTREF)을 따로 저장하고, 검색할 수 있다.

InstTable : 모든 instruction의 정보를 관리하는 테이블 클래스다. instruction의 정보가 적힌 파일을 열고 완성된 instruction table에 검색 기능을 제공한다.

Instruction : 명령어의 구체적인 정보가 담기는 클래스다. 명령어, opcode, 형식, 피연산자 개수등이 저장된다. instruction file의 한 줄을 입력받아 해당 명령어의 정보들을 파싱해서 저장한다.

TokenTable : 어셈블리 코드를 단어별로 분할 한 후, 의미를 분석하고, 최종 코드로 변환하는 과정을 총괄하는 클래스다. object code 생성, 테이블 참조, 상태 추적 등을 수행한다. Section마다 하나씩 할당된다.

Token : 라인별로 어셈블리 코드를 단어 단위로 나눈 후 해석된 의미를 저장하는 클래스다. pass2 object code 변환 과정에서 명령어 정보와 위치 정보를 바탕으로 object code를 생성하는 역할을 한다.토큰을 바이트 코드로 바꾸는 기능을 제공한다.

ObjectProgram : Section별 헤더 정보, object code 목록, modification 목록, EXTREF EXTDEF 정보 등을 저장하는 클래스다. 정보별로 레코드를 추가하는 기능과 전체 정보가 포함된 object code를 출력하는 기능을 제공한다. Section별로 하나씩 할당된다.

Text : 토큰이랑 리터럴 테이블에서 변환된 object code 클래스다. object code 자체는 Ineteger형 변수에 저장하고 길이와 위치도 같이 저장한다. 전체 4바이트 데이터에서 길이 만큼을 문자열로 바꾸는 기능과 비트 연산을 통한 피연산자 값 더하기, nixbpe 더하기 등의 기능을 제공한다. 위치를 기준으로 Section의 Text들이 오름차순 정렬되어야 하므로 Comparable<Text> 인터페이스를 구현한다.

Header : Section의 헤더 작성을 위한 프로그램의 시작 위치, 길이, 이름 등을 저장하는 클래스다.

Modification : Section의 Modification record 작성을 위한 수정 위치, 길이, 연산자(+,-), 대상 심 심볼 등을 저장하는 클래스다.

3장 수행결과

작업 경로에 있는 input.txt를 읽어서 리터럴 테이블과 심볼 테이블을 파일로 출력하고 object code도 파일 출력한다. 테이블은 심볼(리터럴)과 주소로 이루어져 있다. input2_error.txt와 같은 에러 코드가 있는 파일도 읽어서 에러 내용을 표준 출력한다.

input - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
COPY    START   0       COPY FILE FROM IN TO OUTPUT
        EXTDEF  BUFFER,BUFEND,LENGTH
        EXTREF  RDREC,WRREC
FIRST   STL     RETADR  SAVE RETURN ADDRESS
CLOOP   +JSUB   RDREC   READ INPUT RECORD
        LDA     LENGTH  TEST FOR EOF (LENGTH = 0)
        COMP    #0
        JEQ     ENDFIL  EXIT IF EOF FOUND
        +JSUB   WRREC   WRITE OUTPUT RECORD
        J       CLOOP   LOOP
ENDFIL  LDA     =C'EOF' INSERT END OF FILE MARKER
        STA     BUFFER
        LDA     #3      SET LENGTH = 3
        STA     LENGTH
        +JSUB   WRREC   WRITE EOF
        J       @RETADR RETURN TO CALLER
RETADR  RESW    1
LENGTH  RESW    1       LENGTH OF RECORD
        LTORG
BUFFER  RESB    4096    4096-BYTE BUFFER AREA
BUFEND  EQU     *
MAXLEN  EQU     BUFEND-BUFFER   MAXIMUM RECORD LENGTH
RDREC   CSECT
.
.       SUBROUTINE TO READ RECORD INTO BUFFER
.
        EXTREF  BUFFER,LENGTH,BUFEND
        CLEAR   X       CLEAR LOOP COUNTER
        CLEAR   A       CLEAR A TO ZERO
        CLEAR   S       CLEAR S TO ZERO
        LDT     MAXLEN
RLOOP   TD      INPUT   TEST INPUT DEVICE
        JEQ     RLOOP   LOOP UNTIL READY
        RD      INPUT   READ CHARACTER INTO REGISTER A
        COMPR   A,S     TEST FOR END OF RECORD (X'00')
        JEQ     EXIT    EXIT LOOP IF EOR
        +STCH   BUFFER,X        STORE CHARACTER IN BUFFER
        TIXR    T       LOOP UNLESS MAX LENGTH
        JLT     RLOOP   HAS BEEN REACHED
EXIT    +STX    LENGTH  SAVE RECORD LENGTH
        RSUB            RETURN TO CALLER
INPUT   BYTE    X'F1'   CODE FOR INPUT DEVICE
MAXLEN  WORD    BUFEND-BUFFER
WRREC   CSECT
.
.       SUBROUTINE TO WRITE RECORD FROM BUFFER
.
        EXTREF  LENGTH,BUFFER
        CLEAR   X       CLEAR LOOP COUNTER
        +LDT    LENGTH
WLOOP   TD      =X'05'  TEST OUTPUT DEVICE
        JEQ     WLOOP   LOOP UNTIL READY
        +LDCH   BUFFER,X        GET CHARACTER FROM BUFFER
        WD      =X'05'  WRITE CHARACTER
        TIXR    T       LOOP UNTIL ALL CHARACTERS
        JLT     WLOOP   HAVE BEEN WRITTEN
        RSUB            RETURN TO CALLER
        END     FIRST
```

<input.txt>

output_symtab - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V)

```
COPY    0x0000
FIRST   0x0000  COPY
CLOOP   0x0003  COPY
ENDFIL  0x0017  COPY
RETADR  0x002A  COPY
LENGTH  0x002D  COPY
BUFFER  0x0033  COPY
BUFEND  0x1033  COPY
MAXLEN  0x1000  COPY
RDREC   REF
WRREC   REF

RDREC   0x0000
RLOOP   0x0009  RDREC
EXIT    0x0020  RDREC
INPUT   0x0027  RDREC
MAXLEN  0x0028  RDREC
BUFFER  REF
LENGTH  REF
BUFEND  REF

WRREC   0x0000
WLOOP   0x0006  WRREC
LENGTH  REF
BUFFER  REF
```

<output_symtab.txt>

output_littab - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도

```
=C'EOF'         0x0030

=X'05'          0x001B
```

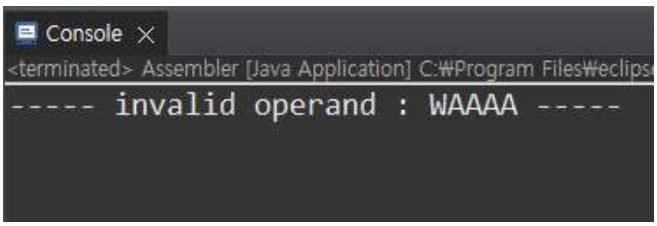
<output_littab.txt>

input2_error - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
COPY    START   0       COPY FILE FROM IN TO OUTPUT
        EXTDEF  BUFFER,BUFEND,LENGTH
        EXTREF  RDREC,WRREC
FIRST   STL     RETADR  SAVE RETURN ADDRESS
CLOOP   +JSUB   RDREC   READ INPUT RECORD
        LDA     LENGTH  TEST FOR EOF (LENGTH = 0)
        COMP    #0
        JEQ     ENDFIL  EXIT IF EOF FOUND
        +JSUB   WAAAA   WRITE OUTPUT RECORD
        J       CLOOP   LOOP
ENDFIL  LDA     =C'EOF' INSERT END OF FILE MARKER
        STA     BUFFER
        LDA     #3      SET LENGTH = 3
        STA     LENGTH
        +JSUB   WRREC   WRITE EOF
        J       @RETADR RETURN TO CALLER
RETADR  RESW    1
LENGTH  RESW    1       LENGTH OF RECORD
        LTORG
BUFFER  RESB    4096    4096-BYTE BUFFER AREA
BUFEND  EQU     *
MAXLEN  EQU     BUFEND-BUFFER   MAXIMUM RECORD LENGTH
RDREC   CSECT
.
.       SUBROUTINE TO READ RECORD INTO BUFFER
.
        EXTREF  BUFFER,LENGTH,BUFEND
        CLEAR   X       CLEAR LOOP COUNTER
        CLEAR   A       CLEAR A TO ZERO
        CLEAR   S       CLEAR S TO ZERO
        LDT     MAXLEN
RLOOP   TD      INPUT   TEST INPUT DEVICE
        JEQ     RLOOP   LOOP UNTIL READY
        RD      INPUT   READ CHARACTER INTO REGISTER A
        COMPR   A,S     TEST FOR END OF RECORD (X'00')
        JEQ     EXIT    EXIT LOOP IF EOR
        +STCH   BUFFER,X        STORE CHARACTER IN BUFFER
        TIXR    T       LOOP UNLESS MAX LENGTH
        JLT     RLOOP   HAS BEEN REACHED
EXIT    +STX    LENGTH  SAVE RECORD LENGTH
        RSUB            RETURN TO CALLER
INPUT   BYTE    X'F1'   CODE FOR INPUT DEVICE
MAXLEN  WORD    BUFEND-BUFFER
WRREC   CSECT
.
.       SUBROUTINE TO WRITE RECORD FROM BUFFER
.
        EXTREF  LENGTH,BUFFER
        CLEAR   X       CLEAR LOOP COUNTER
        +LDT    LENGTH
WLOOP   TD      =X'05'  TEST OUTPUT DEVICE
        JEQ     WLOOP   LOOP UNTIL READY
        +LDCH   BUFFER,X        GET CHARACTER FROM BUFFER
        WD      =X'05'  WRITE CHARACTER
        TIXR    T       LOOP UNTIL ALL CHARACTERS
        JLT     WLOOP   HAVE BEEN WRITTEN
        RSUB            RETURN TO CALLER
        END     FIRST
```

<input2_error.txt>



<System.out>

4장 결론 및 보충할 점

Assembler, LiteralTable, SymbolTable, InstTable, Instruction, TokenTable, Token, ObjectProgram, Text, Header, Modification 클래스들을 활용해서 pass1과 pass2를 거치면서 필요한 데이터들을 저장하고 최종적으로는 object code로의 변환을 마쳤다. 이 프로젝트를 완성하고 나서 이미 정의된 변수와 함수에서 보충할 점을 느꼈다.

4.1. Object Code 변수

object code를 저장하는 Assembler.codeList와 Token.objectCode는 String보단 Integer에 적합

해 보인다. 본 프로그램에서 나올 수 있는 object code는 최대 4바이트이고 비트 연산과 합 연산을 쉽게 할 수 있기 때문이다. 하지만 데이터를 추가할 때 매번 비트 연산을 하기에는 가독성과 사용성이 떨어지므로 관련 편의성 함수와 위치와 길이 정보까지 포함한 새로운 클래스를 만드는 게 나아 보인다. 본 프로그램에서는 Text 클래스를 만들어 사용했다.

4.2. Assembler의 symtabList, TokenList

Assembler.symtabList와 Assembler.TokenList는 ArrayList 자료형보다 HashMap 자료형이 더 유용해 보인다. ArrayList로 구현하면 Section index로 Section의 심볼 테이블과 토큰 테이블에 접근하기 때문에 Section name에 맞는 Section index 변환 과정이 하나 더 필요하다. 하지만 HashMap<String, Table>으로 구현하면 Section name에 맞는 테이블에 바로 접근할 수 있다.

4.3. LiteralTable과 SymbolTable

LiteralTable과 SymbolTable에는 ArrayList 형식의 dataList와 locationList를 놓는 대신 각각 Literal과 Symbol 클래스를 만들고 HashMap<String, Literal>, HashMap<String, Symbol>를 테이블 변수로 넣는 게 나을 것 같다. Literal과 Symbol에는 기존 테이블에서 ArrayList로 저장하던 값들을 멤버변수로 저장한다. 그리고 ArrayList 대신에 HashMap을 사용하면 찾고자 하는 값을 O(1)에 가까운 시간복잡도로 구할 수 있다.

5장 소스코드(+주석)

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

import javax.sound.midi.Soundbank;

import java.io.*;


/**
 * Assembler :
 * 이 프로그램은 SIC/XE 머신을 위한 Assembler 프로그램의 메인 루틴이다.
 * 프로그램의 수행 작업은 다음과 같다. <br>
 * 1) 처음 시작하면 Instruction 명세를 읽어들여서 assembler를 세팅한다. <br>
 * 2) 사용자가 작성한 input 파일을 읽어들인 후 저장한다. <br>
 * 3) input 파일의 문장들을 단어별로 분할하고 의미를 파악해서 정리한다. (pass1) <br>
 * 4) 분석된 내용을 바탕으로 컴퓨터가 사용할 수 있는 object code를 생성한다. (pass2) <br>
 *
```

```
 * <br><br>
 * 작성중의 유의사항 : <br>
 *  1) 새로운 클래스, 새로운 변수, 새로운 함수 선언은 얼마든지 허용됨. 단, 기존의 변수와 함수들을 삭제하거나 완전히 대체하는 것은 안된다.<br>
 *  2) 마찬가지로 작성된 코드를 삭제하지 않으면 필요에 따라 예외처리, 인터페이스 또는 상속 사용 또한 허용됨.<br>
 *  3) 모든 void 타입의 리턴값은 유저의 필요에 따라 다른 리턴 타입으로 변경 가능.<br>
 *  4) 파일, 또는 콘솔창에 한글을 출력시키지 말 것. (채점상의 이유. 주석에 포함된 한글은 상관 없음)<br>
 *
 * <br><br>
 *  + 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<SymbolTable> symtabList;
	/** 프로그램의 section별로 프로그램을 저장하는 공간*/
	ArrayList<TokenTable> TokenList;
	/** 프로그램의 section별로 literal table을 저장하는 공간*/
	ArrayList<LiteralTable> littabList;
	/**
	 * Token, 또는 지시어에 따라 만들어진 오브젝트 코드들을 출력 형태로 저장하는 공간. <br>
	 * 필요한 경우 String 대신 별도의 클래스를 선언하여 ArrayList를 교체해도 무방함.
	 */
	//section별 ObjectCode 리스트
	ArrayList<ObjectProgram> codeList;
	//symtabList, littabList, TokenList에 접근하기 위한 section index 해시맵
	HashMap<String, Integer> sectionList;
	//현재 section 이름
	String section;
	//section 개수
	int section_counter;
	//location counter
	int locctr;
```

```
/**
 * 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>();
        littabList = new ArrayList<LiteralTable>();
        TokenList = new ArrayList<TokenTable>();
        codeList = new ArrayList<ObjectProgram>();
        sectionList = new HashMap<String, Integer>();
        Token.instTable = instTable;
}


//코드 변환 시 에러가 발생하면 에러 내용을 출력하고 즉시 프로그램 종료
public static void exitError(String errorString) {
        System.out.println("----- " + errorString + " -----");
        System.exit(1);
}

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

        if(assembler.pass1() == false) {
                exitError("pass1 error");
        }
        assembler.printSymbolTable("output_symtab.txt");
        assembler.printLiteralTable("output_littab.txt");
        //assembler.printSymbolTable(null);
        //assembler.printLiteralTable(null);

        assembler.pass2();
```

```
        assembler.printObjectCode("output_objectcode.txt");
        //assembler.printObjectCode(null);
}


/**
 * inputFile을 읽어들여서 lineList에 저장한다.<br>
 * @param inputFile : input 파일 이름.
 */
private void loadInputFile(String inputFile) {
        // TODO Auto-generated method stub
 try (BufferedReader br = new BufferedReader(new FileReader(inputFile))) {
     String line;
     while ((line = br.readLine()) != null) {
         lineList.add(line);
     }
 } catch (IOException e) {
     e.printStackTrace();
 }
}

/**
 * 작성된 SymbolTable들을 출력형태에 맞게 출력한다.<br>
 * @param fileName : 저장되는 파일 이름
 */
private void printSymbolTable(String fileName) {
        PrintStream ps = getPrintStream(fileName);
        for(int i = 0; i < section_counter; ++i) {
                if(symtabList.get(i).symbolList.size() <= 0)
                        continue;
                symtabList.get(i).print(ps);
                ps.println();
        }
}

/**
 * 작성된 LiteralTable들을 출력형태에 맞게 출력한다.<br>
 * @param fileName : 저장되는 파일 이름
 */
private void printLiteralTable(String fileName) {
```

```
            PrintStream ps = getPrintStream(fileName);
            for(int i = 0; i < section_counter; ++i) {
                    if(littabList.get(i).literalList.size() <= 0)
                            continue;
                    littabList.get(i).print(ps);
                    ps.println();
            }
    }

    /**
     * pass1 과정을 수행한다.<br>
     *   1) 프로그램 소스를 스캔하여 토큰단위로 분리한 뒤 토큰테이블 생성<br>
     *   2) label을 symbolTable에 정리<br>
     *   <br><br>
     *    주의사항 : SymbolTable과 TokenTable은 프로그램의 section별로 하나씩 선언되
어야 한다.
     */
    private boolean pass1() {
            section_counter = 0;
            // TODO Auto-generated method stub
            for(String line : lineList) {
                    Token token = new Token(line, locctr);
                    if(token.getParseSucceed() == false) {
                            return false;
                    }
                    locctr = nextLocctr(token);
                    String token_section = nextSection(token);
                    if(token_section != section) {
                            if(section != null) {
                                    locctr = getLiteralTable().ltorg(locctr);
                            }
                            section = token_section;
                            createNewSection(token_section);
                    }
                    getTokenTable().putToken(token);
                    addSymbol(token);
                    addLiteral(token);
                    check_ltorg(token);
                    check_exts(token);
                    getObjectProgram().setLength(locctr);
```

```
                //token.print(line);
        }
        locctr = getLiteralTable().ltorg(locctr);
        getObjectProgram().setLength(locctr);
        return true;
}

/**
 * pass2 과정을 수행한다.<br>
 *   1) 분석된 내용을 바탕으로 object code를 생성하여 codeList에 저장.
 */
private void pass2() {
        // TODO Auto-generated method stub
        for(int i = 0; i < section_counter; ++i) {
                int token_list_size = TokenList.get(i).tokenList.size();
                for(int j = 0; j < token_list_size; ++j) {
                        Text object_text = TokenList.get(i).makeObjectCode(j);
                        codeList.get(i).addObjectText(object_text);
                        //System.out.println(object_text);
                }

                int littab_size = littabList.get(i).literalList.size();
                //System.out.println("littab_size : " + littab_size);
                for(int j = 0; j < littab_size; ++j) {
                        Text object_text = littabList.get(i).getCodeText(j);
                        codeList.get(i).addObjectText(object_text);
                        //System.out.println(object_text);
                }

                for(Token token : TokenList.get(i).tokenList) {
                        codeList.get(i).appendMods(token.modRecords);
                }
                codeList.get(i).setExtdefLocation(symtabList.get(i));
                codeList.get(i).sortTexts();
                //codeList.get(i).printTexts();
        }
}
```

```
//fileName으로 null을 주면 표준출력, 이외에는 해당 파일의 출력스트림으로 변환
private PrintStream getPrintStream(String fileName) {
	PrintStream ps = System.out;
	try {
		if(fileName != null)
			ps = new PrintStream(fileName);
	} catch(FileNotFoundException e) {
		e.printStackTrace();
	}
	return ps;
}

/**
 * 작성된 codeList를 출력형태에 맞게 출력한다.<br>
 * @param fileName : 저장되는 파일 이름
 */
private void printObjectCode(String fileName) {
	// TODO Auto-generated method stub
	PrintStream ps = getPrintStream(fileName);
	for(int i = 0; i < section_counter; ++i) {
		codeList.get(i).print(ps);
		ps.println();
	}
}



//현재 section의 토큰 테이블 반환
private TokenTable getTokenTable() {
	int sec_idx = sectionList.get(section);
	return TokenList.get(sec_idx);
}



//현재 section의 리터럴 테이블 반환
private LiteralTable getLiteralTable() {
	int sec_idx = sectionList.get(section);
	return littabList.get(sec_idx);
```

```
	}


	//현재 section의 심볼 테이블 반환
	private SymbolTable getSymbolTable() {
		int sec_idx = sectionList.get(section);
		return symtabList.get(sec_idx);
	}


	//현재 section의 ObjectProgram 반환
	private ObjectProgram getObjectProgram() {
		int sec_idx = sectionList.get(section);
		return codeList.get(sec_idx);
	}


	//토큰의 명령어가 ltorg인지 확인하고 맞으면 리터럴을 object code로 변환하고 배치
	private void check_ltorg(Token token) {
		if(token.operator != null && token.operator.equals("LTORG")) {
			locctr = getLiteralTable().ltorg(locctr);
		}
	}


	//토큰의 명령어가 extref, extdef인지 검사하고 맞으면 extref, extdef를 필요로하는 곳에 레이블 전달
	private void check_exts(Token token) {
		if(token == null || token.operator == null)
			return;

		if(token.operator.equals("EXTREF")) {
			for(String operand : token.operands) {
				getObjectProgram().addExtref(operand);
				getSymbolTable().addExtref(operand);
			}
```

```
		}
		else if(token.operator.equals("EXTDEF")) {
			for(String operand : token.operands) {
				getObjectProgram().addExtdef(operand, 0);
			}
		}
	}



	//토큰을 분석해서 레이블이 있을 경우에는 심볼 테이블에 추가
	private void addSymbol(Token token) {
		if(token.label == null || token.label.equals(""))
			return;

		if(token.operator.equals("EQU")) {
			getSymbolTable().putSymbol(token.label,
token.getOperandsResult(getSymbolTable(), getLiteralTable()));
			return;
		}
		getSymbolTable().putSymbol(token.label, token.location);
	}



	//토큰을 분석해서 리터럴이 있을 경우에는 리터럴 테이블에 추가
	private void addLiteral(Token token) {
		if(token.operands.length <= 0)
			return;
		if(token.operands[0].charAt(0) != '=')
			return;
		getLiteralTable().putLiteral(token.operands[0], token.location);
	}



	//새로운 section에 진입할 때 리터럴 테이블, 심볼 테이블, ObjectProgram 등을 할당
	private void createNewSection(String new_section) {
		sectionList.put(new_section, section_counter++);
		symtabList.add(new SymbolTable(new_section));
```

```
            littabList.add(new LiteralTable());
            codeList.add(new ObjectProgram(new_section, locctr, section_counter == 1));

            int sec_idx = sectionList.get(new_section);
            TokenList.add(new TokenTable(symtabList.get(sec_idx), littabList.get(sec_idx),
instTable));
        }



        //토큰의 연산자가 START, CSECT인지를 확인해서 다음 section을 설정
        private String nextSection(Token token) {
            String operator = token.operator;
            String[] operands = token.operands;
            if(operator == null || operands == null)
                return section;

            if(operator.equals("START") || operator.equals("CSECT")) {
                if(token.label != null)
                    return token.label;
            }
            return section;
        }



        //토큰의 형식과 연산자를 분석해서 다음 locctr를 설정
        private int nextLocctr(Token token) throws NumberFormatException {
            String operator = token.operator;
            String[] operands = token.operands;
            if(operator == null || operands == null)
                return locctr;

            if(operator.equals("RESB") || operator.equals("RESW")) {
                int locgap = Integer.parseInt(operands[0]) * token.instruction.format;
                token.setLocIncrement(locgap);
                return locctr + locgap;
            }
            else if(operator.equals("BYTE")) {
                int locgap = LiteralTable.getLiteralLength(operands[0]) / 2;
```

```
                token.setLocIncrement(locgap);
                return locctr + locgap;
            }
            else if(operator.equals("START")) {
                int start_loc = Integer.parseInt(operands[0]);
                token.setLocation(start_loc);
                return start_loc;
            }
            else if(operator.equals("CSECT")) {
                token.setLocation(0);
                return 0;
            }

            token.setLocIncrement(token.instruction.format);
            return locctr + token.instruction.format;
        }
}
```

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


public class LiteralTable {
    ArrayList<String> literalList;
    ArrayList<Integer> locationList;
    //리터얼의 길이 저장
    ArrayList<Integer> lengthList;
    //리터럴이 LTORG나 프로그램 끝을 만나 ㅊ
    ArrayList<Boolean> placementFlags;
    //리터럴의 원본 저장
    ArrayList<String> originalList;



    public LiteralTable() {
       originalList = new ArrayList<String>();
```

```
        literalList = new ArrayList<String>();
        locationList = new ArrayList<Integer>();
        lengthList = new ArrayList<Integer>();
        placementFlags = new ArrayList<Boolean>();
        }



    public void putLiteral(String literal, int location) {
        originalList.add(literal);
        literalList.add(getLiteralString(literal));
        locationList.add(location);
        lengthList.add(getLiteralLength(literal));
        placementFlags.add(false);
    }



    //제어문이 LTORG나 프로그램 끝을 만나서 object code로 안 변환된 리터럴들을 전부 변
환
    public int ltorg(int location) {
        for(int i = 0; i < literalList.size(); ++i) {
                if(placementFlags.get(i) == false) {
                        String original = originalList.get(i);
                        String literal = literalList.get(i);
                        int length = lengthList.get(i);
                        while (literalList.contains(literal)) {
                                int index = literalList.indexOf(literal);
                                originalList.remove(index);
                                literalList.remove(literal);
                                locationList.remove(index);
                                lengthList.remove(index);
                                placementFlags.remove(index);
                        }
                        originalList.add(original);
                        literalList.add(literal);
                        locationList.add(location);
                        lengthList.add(length);
                        placementFlags.add(true);
                        location += length / 2;
```

```
            }
        }
        return location;
    }



    //리터럴의 바이트 수 반환
    public static int getLiteralLength(String literal) {
        int start = literal.indexOf('\'');
         int end = literal.lastIndexOf('\'');

         if (start == -1 || end == -1 || start >= end) {
             return 0;
         }
         int length = (end - start - 1) * 2;
         length = (literal.charAt(start - 1) == 'X') ? length / 2 : length;
         return length;
    }



    //따옴표(')로 둘러싸여진 리터럴의 데이터 추출(ex. =C'EOF' -> EOF, =X'05' -> 05)
    public static String getLiteralString(String literal) {
        int start = literal.indexOf('\'');
         int end = literal.lastIndexOf('\'');

         if (start == -1 || end == -1 || start >= end) {
             return null;
         }
         String extracted = literal.substring(start + 1, end);
         return extracted;
    }



    //리터럴을 object code로 변환
    public static int getLiteralCode(String literal) {;
                try {
                        int byte_data = Integer.parseInt(literal, 16);
```

```
                return byte_data;
            } catch(NumberFormatException e) {
                StringBuilder sb = new StringBuilder();
                for (char c : literal.toCharArray()) {
                    sb.append(String.format("%02X", (int)c));
                }
                return Integer.parseUnsignedInt(sb.toString(), 16);
            }
    }
```

```
//리터럴 테이블의 n번째 리터럴을 Text(object code)로 변환
public Text getCodeText(int index) {
    int literal_code = getLiteralCode(literalList.get(index));
    Text code_text = new Text(locationList.get(index), lengthList.get(index), (byte)0xFF);
    code_text.addData(literal_code, 0);
    return code_text;
}
```

```
//리터럴 테이블에서 리터럴 검색, 없으면 -1 반환
public int searchLiteral(String literal) {
            //...
            for(int i = 0; i < literalList.size(); ++i) {
                if(literalList.get(i).equals(literal))
                    return locationList.get(i);
            }
            return -1;
    }
```

```
//printStream(System.out or file)으로 리터럴 테이블의 레코드 정보를 출력
public void print(PrintStream ps) {
    for(int i = 0; i < literalList.size(); ++i) {
            ps.printf("%s\t\t0x%04X\n", originalList.get(i), locationList.get(i));
    }
    }
```

```
    // 필요 메서드 추가 구현
}




import java.io.PrintStream;
import java.util.ArrayList;

/**
 * symbol과 관련된 데이터와 연산을 소유한다.
 * section 별로 하나씩 인스턴스를 할당한다.
 */
public class SymbolTable {
        ArrayList<String> symbolList;
        ArrayList<Integer> locationList;
        //테이블이 속한 section의 이름
        String section;
        //EXTREF 리스트
        ArrayList<String> extrefList;



        public SymbolTable(String section) {
                this.section = section;
                symbolList = new ArrayList<String>();
                locationList = new ArrayList<Integer>();
                extrefList = new ArrayList<String>();
        }

        /**
         * 새로운 Symbol을 table에 추가한다.
         * @param symbol : 새로 추가되는 symbol의 label
         * @param location : 해당 symbol이 가지는 주소값
         * <br><br>
         * 주의 : 만약 중복된 symbol이 putSymbol을 통해서 입력된다면 이는 프로그램 코드
에 문제가 있음을 나타낸다.
         * 매칭되는 주소값의 변경은 modifySymbol()을 통해서 이루어져야 한다.
         */
```

```
public void putSymbol(String symbol, int location) {
        symbolList.add(symbol);
        locationList.add(location);
}



/**
 * 기존에 존재하는 symbol 값에 대해서 가리키는 주소값을 변경한다.
 * @param symbol : 변경을 원하는 symbol의 label
 * @param newLocation : 새로 바꾸고자 하는 주소값
 */
public void modifySymbol(String symbol, int newLocation) {

}

/**
 * 인자로 전달된 symbol이 어떤 주소를 지칭하는지 알려준다.
 * @param symbol : 검색을 원하는 symbol의 label
 * @return symbol이 가지고 있는 주소값. 해당 symbol이 없을 경우 -1 리턴
 */
public int searchSymbol(String symbol) {
        //...
        for(int i = 0; i < symbolList.size(); ++i) {
                if(symbolList.get(i).equals(symbol))
                        return locationList.get(i);
        }
        return -1;
}



//EXTREF 레코드 추가
public void addExtref(String extref) {
        extrefList.add(extref);
}



//EXTREF 레코드 검색
```

```
	public int searchExtref(String symbol) {
		for(String extref : extrefList) {
			if(extref.equals(symbol))
				return 0;
		}
		return -1;
	}



	//printStream(System.out or file)으로 심볼 테이블의 레코드 정보를 출력
	public void print(PrintStream ps) {
	for(int i = 0; i < symbolList.size(); ++i) {
		String section_stream = symbolList.get(i).equals(section) ? "" : section;
		ps.printf("%s\t0x%04X\t%s\n", symbolList.get(i), locationList.get(i),
section_stream);
	}
	for(String extref : extrefList) {
		ps.printf("%s\tREF\n", extref);
	}
	}
}




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


/**
 * 모든 instruction의 정보를 관리하는 클래스. instruction data들을 저장한다. <br>
 * 또한 instruction 관련 연산, 예를 들면 목록을 구축하는 함수, 관련 정보를 제공하는 함수
```

등을 제공 한다.
 */
```
public class InstTable {
	/**
	 * inst.data 파일을 불러와 저장하는 공간.
	 *  명령어의 이름을 집어넣으면 해당하는 Instruction의 정보들을 리턴할 수 있다.
	 */
	HashMap<String, Instruction> instMap;

	/**
	 * 클래스 초기화. 파싱을 동시에 처리한다.
	 * @param instFile : instuction에 대한 명세가 저장된 파일 이름
	 */
	public InstTable(String instFile) {
		instMap = new HashMap<String, Instruction>();
		openFile(instFile);
	}

	/**
	 * 입력받은 이름의 파일을 열고 해당 내용을 파싱하여 instMap에 저장한다.
	 */
	public void openFile(String fileName) {
		//...
		try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
	    String line;
	    while ((line = br.readLine()) != null) {
		Instruction inst = new Instruction(line);
		instMap.put(inst.operator, inst);
	    }
	 } catch (IOException e) {
	    e.printStackTrace();
	 }
	}
	//get, set, search 등의 함수는 자유 구현



	//명령어 테이블에서 명령어 검색, 없으면 null 반환
	public Instruction search(String operator) {
		Instruction instFound = instMap.get(operator);
```

```
			if(instFound == null)
				return null;
			return new Instruction(instFound);
		}

}
/**
 * 명령어 하나하나의 구체적인 정보는 Instruction클래스에 담긴다.
 * instruction과 관련된 정보들을 저장하고 기초적인 연산을 수행한다.
 */
class Instruction {
		/*
		 * 각자의 inst.data 파일에 맞게 저장하는 변수를 선언한다.
		 *
		 * ex)
		 * String instruction;
		 * int opcode;
		 * int numberOfOperand;
		 * String comment;
		 */

		String operator;
		byte opcode;
		/** instruction이 몇 바이트 명령어인지 저장. 이후 편의성을 위함 */
		int format;
		//음수면 무한 인자(ex. EXTREF, EXTDEF)
		int operandCount;

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



		//복사 생성자
		public Instruction(Instruction other) {
```

```
		if(other == null) return;
		this.operator = other.operator;
		this.opcode = other.opcode;
		this.format = other.format;
		this.operandCount = other.operandCount;
	}

	/**
	 * 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[] splits = line.split("\t");
		if(splits.length > 4)
			return;
		operator = splits[0];
		opcode = (byte)Integer.parseInt(splits[1], 16);  // 16진수 -> byte
		format = Integer.parseInt(splits[2]);
		operandCount = Integer.parseInt(splits[3]);
	}

}
```

```
import java.util.*;

import javax.sound.midi.Soundbank;

/**
 * 사용자가 작성한 프로그램 코드를 단어별로 분할 한 후, 의미를 분석하고, 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 littab;
	InstTable instTab;


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

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

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

	/**
	 * Pass2 과정에서 사용한다.
	 * instruction table, symbol table 등을 참조하여 objectcode를 생성하고, 이를 저장한다.
	 * @param index
	 */
	public Text makeObjectCode(int index){
		//...
		return tokenList.get(index).makeObjectCode(symTab, littab);
	}

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

}

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

```
String label;
String operator;
String[] operands;
String comment;
char nixbpe;
//locctr 증가치
int locIncrement;
//파싱 성공 여부
boolean isParseSucceed;
//토큰의 operator에 해당하는 명령어 정보
Instruction instruction;
//modification record 리스트
ArrayList<Modification> modRecords;

//명령어 검색을 위해 참조할 명령어 테이블
static InstTable instTable;
//레지스터 번호 해시맵
static HashMap<String, Integer> registerMap;
static {
    registerMap = new HashMap<String, Integer>();
    registerMap.put("A", 0);
    registerMap.put("X", 1);
    registerMap.put("L", 2);
    registerMap.put("B", 3);
    registerMap.put("S", 4);
    registerMap.put("T", 5);
    registerMap.put("F", 6);
    registerMap.put("PC", 8);
    registerMap.put("SW", 9);
}

// 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, int locctr) {
```

```
		//initialize 추가
		location = locctr;
		operands = new String[] {};
		modRecords = new ArrayList<Modification>();
		setFlag(TokenTable.nFlag, 1);
		setFlag(TokenTable.iFlag, 1);
		setFlag(TokenTable.pFlag, 1);
		parsing(line);
	}

	/**
	 * line의 실질적인 분석을 수행하는 함수. Token의 각 변수에 분석한 결과를 저장한다.
	 * @param line 문장단위로 저장된 프로그램 코드.
	 */
	public void parsing(String line) {
		List<String> splits = Arrays.asList(line.split("\t+"));
		isParseSucceed = false;
		if(splits.size() <= 0)
			return;
		isParseSucceed = true;
		if(splits.get(0).equals("."))
			return;
		Iterator<String> iter = splits.iterator();
		if(instTable.search(splits.get(0)) == null) {
			isParseSucceed = isParseSucceed && iter.hasNext() && parseLabel(iter.next());
		}
		isParseSucceed = isParseSucceed && iter.hasNext() && parseOperator(iter.next());
		if(iter.hasNext()) {
			isParseSucceed = isParseSucceed && parseOperands(iter.next());
		}
		setByteSize();
	}

	/**
	 * n,i,x,b,p,e flag를 설정한다. <br><br>
	 *
	 * 사용 예 : setFlag(nFlag, 1); <br>
```

```
     *   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 != 0 && value != 1) return;
            int exponent = (int) (Math.log(flag) / Math.log(2));
            if(value == 1) {
                    nixbpe |= 1 << exponent;
            }
            else {
                    nixbpe = (char)(nixbpe & ~(1 << exponent));
            }
    }

    /**
     * 원하는 flag들의 값을 얻어올 수 있다. flag의 조합을 통해 동시에 여러개의 플래그를 얻는 것 역시 가능하다 <br><br>
     *
     * 사용 예 : getFlag(nFlag) <br>
     *   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;
    }



    //locIncrement setter 함수
    public void setLocIncrement(int locIncrement) {
            this.locIncrement = locIncrement;
    }
```

```
	//isParseSucceed getter 함수
	public boolean getParseSucceed() {
		return isParseSucceed;
	}



	//레이블 파싱
	private boolean parseLabel(String label) {
		this.label = label;
		return true;
	}



	//명령어 파싱
	private boolean parseOperator(String operator) {
		this.operator = operator;
		String searchTarget = operator;
		if(operator.length() > 0 && operator.charAt(0) == '+') {
			instruction = instTable.search(operator.substring(1));
			setFlag(TokenTable.eFlag, 1);
			setFlag(TokenTable.pFlag, 0);
			setFlag(TokenTable.bFlag, 0);
			instruction.operator = operator;
			instruction.format = 4;
		}
		else {
			instruction = instTable.search(operator);
		}

		if(operator.equals("BYTE") || operator.equals("WORD") || instruction.format <=
2) {
			nixbpe = 0;
		}
		return (instruction != null);
	}
```

```
	//쉼표(,)를 기준으로 피연산자 파싱, @ # X 유뮤에 따라 n i x 비트 설정
	private boolean parseOperands(String operands) {
		if(instruction.operandCount == 0) {
			setFlag(TokenTable.pFlag, 0);
			setFlag(TokenTable.bFlag, 0);
			return true;
		}
		this.operands = operands.split(",");
		for (int i = 0; i < this.operands.length; i++) {
			this.operands[i] = this.operands[i].trim();
	 }

		if(this.operands.length <= 0)
			return false;

		if(this.operands[0].charAt(0) == '#') {
			setFlag(TokenTable.nFlag, 0);
			this.operands[0] = this.operands[0].substring(1);
		}
		else if(this.operands[0].charAt(0) == '@') {
			setFlag(TokenTable.iFlag, 0);
			this.operands[0] = this.operands[0].substring(1);
		}

		if(instruction.operandCount == -1 || instruction.operandCount == this.operands.length)
			return true;
		String lastOperand = this.operands[this.operands.length - 1];
		if((instruction.operandCount == this.operands.length - 1) && lastOperand.equals("X")) {
			setFlag(TokenTable.xFlag, 1);
			this.operands = Arrays.copyOf(this.operands, this.operands.length - 1);
			return true;
		}
		return false;
	}
```

```
	//토큰을 object code 로 변환 시에 차지할 바이트 수
	public void setByteSize() {
		if(operator == null || instruction.format <= 0 || operator.equals("RESW") ||
operator.equals("RESB")) {
			byteSize = 0;
		}
		else if(operator.equals("BYTE")) {
			byteSize = LiteralTable.getLiteralLength(operands[0]) / 2;
		}
		else {
			byteSize = instruction.format;
		}
	}



	//location setter 함수
	public void setLocation(int location) {
		this.location = location;
	}



	//2형식 레지스트 연산 토큰의 피연산자 레지스터 값 반환
	private int getFormat2Result() {
		int result = 0;
		for(int i = 0; i < 2 && i < operands.length; ++i) {
			int registerNum = registerMap.get(operands[i]);
			result |= registerNum << ((1 - i) * 4);
		}
		return result;
	}



	//피연산자를 정수 변환 시도, 성공시 정수값 반환
	private int tryParseInt(String part) {
		try {
			int num = Integer.parseInt(part);
			setFlag(TokenTable.pFlag, 0);
```

```
            return num;
        } catch (NumberFormatException e) {
            return -1;
        }
    }


    //피연산자를 바이트 문자열 변환 시도(ex. C'EOF', X'05'), 성공시 문자열 바이트 반환
    private int tryParseBytes(String part) {
        if(operator.equals("BYTE")) {
            String byte_string = LiteralTable.getLiteralString(part);
            return LiteralTable.getLiteralCode(byte_string);
        }
        return -1;
    }


    //리터럴 테이블을 검색해서 피연산자를 리터럴로 변환 시도, 성공시 리터럴 위치 반환
    private int trySearchLiteral(String part, LiteralTable littab) {
        if(part.charAt(0) == '=') {
            String part_literal = LiteralTable.getLiteralString(part);
            return littab.searchLiteral(part_literal);
        }
        return -1;
    }


    //심볼 테이블을 검색해서 피연산자를 심볼로 변환 시도, 성공시 심볼 위치 반환
    private int trySearchSymbol(String part, SymbolTable symtab) {
        return symtab.searchSymbol(part);
    }


    //피연산자를 정수변환, 문자열 변환, 심봄 변환, 리터럴 변환, EXTREF 레이블 검색을 시도하면서 값을 계산
    private int getOperandPartValue(String part, SymbolTable symtab, LiteralTable littab,
```

```
boolean is_negative) {
                int value;
                if(part.equals("*"))
                        return location;
            value = tryParseInt(part);
            if (value >= 0)
                return value;
            value = tryParseBytes(part);
            if (value >= 0)
                return value;
            value = trySearchLiteral(part, littab);
            if (value >= 0)
                return value;
            value = trySearchSymbol(part, symtab);
            if (value >= 0)
                return value;

            value = symtab.searchExtref(part);
            if(value >= 0) {
                createModification(part, is_negative);
                return value;
            }
            Assembler.exitError("invalid operand : " + part);
            return -1;
        }



        //부호와 참조 이름을 받아서 토큰 내부의 Modification record 리스트에 추가
        private void createModification(String part, boolean is_negative) {
                int mod_length = 2 * byteSize - 3;
                if(operator.equals("BYTE") || operator.equals("WORD")) {
                        mod_length = 2 * byteSize;
                }
                int mod_offset = (2 * byteSize - mod_length) / 2;
                char operator = is_negative ? '-' : '+';
                Modification mod_record = new Modification(location + mod_offset, mod_length,
operator, part);
                modRecords.add(mod_record);
        }
```

```
//+-가 포함된 수식 피연산자의 전체값을 계산
public int getOperandsResult(SymbolTable symtab, LiteralTable littab) {
	if(instruction.format == 2)
		return getFormat2Result();
	if(operands.length != 1)
		return 0;

	int result = 0;
	boolean is_negative = false;
	int start_index = 0;
	String formula = operands[0];
	for(int i = 0; i < formula.length(); ++i) {
		if(formula.charAt(i) != '+' && formula.charAt(i) != '-')
			continue;
		String part = formula.substring(start_index, i);

		int part_value = getOperandPartValue(part, symtab, littab, is_negative);
		result += is_negative ? -part_value : part_value;
		is_negative = (formula.charAt(i) == '-');
		start_index = i + 1;

	}
	String part = formula.substring(start_index);
	int part_value = getOperandPartValue(part, symtab, littab, is_negative);
	result += is_negative ? -part_value : part_value;
	return result;
}



//피연산자의 전체값을 구하고 relative address 여부에 따라 locctr와의 차이를 계산
public int getTextAddtion(SymbolTable symtab, LiteralTable littab) {
	int operandResult = getOperandsResult(symtab, littab);
	if(getFlag(TokenTable.pFlag) == TokenTable.pFlag) {
		operandResult = operandResult - (location + locIncrement);
		operandResult &= (1 << 12) - 1;
	}
```

```
			return operandResult;
		}



		//opcode, nixbpe, 피연산자나 주소 차이 등을 더해서 object code를 계산
		public Text makeObjectCode(SymbolTable symtab, LiteralTable littab) {
			if(byteSize <= 0)
				return null;

			Text object_text = new Text(location, byteSize * 2, instruction.opcode);
			object_text.addData(getTextAddtion(symtab, littab), 0);
			object_text.addNixbpe(nixbpe);
			return object_text;
		}
}
```

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


/*Section별 헤더 정보, object code 목록, modification 목록, EXTREF EXTDEF 정보 등을
저장하는 클래스다.
 *Section별로 하나씩 할당된다.
 */
public class ObjectProgram {
		//메인 section 여부
		boolean isMain;
		//헤더 정보
		Header header;
		//object code 리스트
		ArrayList<Text> textList;
		//modification record 리스트
		ArrayList<Modification> modList;
		//extref 레이블 리스트
		ArrayList<String> extrefList;
		//extdef 레이블과 위치의 리스트
		ArrayList<Map.Entry<String, Integer>> extdefList;



		//생성자, 멤버변수 초기화
		public ObjectProgram() {
			textList = new ArrayList<Text>();
```

```
        modList = new ArrayList<Modification>();
        header = new Header();
        extrefList = new ArrayList<String>();
        extdefList = new ArrayList<>();
}


//생성자, section 이름과 프로그램 시작 위치를 저장
public ObjectProgram(String section, int start) {
        this();
        header.name = section;
        header.start = start;
        this.isMain = false;
}


//생성자, 해당 section의 메인 여부를 저장
public ObjectProgram(String section, int start, boolean isMain) {
        this(section, start);
        this.isMain = isMain;
}



//object code 리스트에 object code 추가
public void addObjectText(Text text) {
        if(text == null)
                return;
        textList.add(text);
}



//modification record 리스트에 modification record 추가
public void appendMods(ArrayList<Modification> mods) {
        if(mods == null)
                return;
        for(Modification mod : mods) {
                modList.add(mod);
        }
}



//object code의 위치를 기준으로 object code 리스트 오름차순 정렬
public void sortTexts() {
        Iterator<Text> iterator = textList.iterator();
        while (iterator.hasNext()) {
            if (iterator.next() == null) {
                iterator.remove();
            }
        }
        Collections.sort(textList);
}



//object program의 EXTREF 레코드 문자열
private String getExtrefString() {
        StringBuilder sb = new StringBuilder();
        for(String extref : extrefList) {
                sb.append(String.format("%-6s", extref));
```

```
        }
        return sb.toString();
}



//object program의 EXTDEF 레코드 문자열
private String getExtdefString() {
        StringBuilder sb = new StringBuilder();
        for (Map.Entry<String, Integer> entry : extdefList) {
            String extdef = entry.getKey();
            Integer location = entry.getValue();
            sb.append(String.format("%-6s%06X", extdef, location));
        }
        return sb.toString();
}



//EXTDEF 레이블을 심볼 테이블에서 찾아서 주소를 EXTDEF 리스트에 저장
public void setExtdefLocation(SymbolTable symtab) {
        for (Map.Entry<String, Integer> entry : extdefList) {
            String extdef = entry.getKey();
            entry.setValue(symtab.searchSymbol(extdef));
        }
}



//object program의 맨 끝 줄의 레코드 문자열
private String getEndString() {
        if(isMain)
                return String.format("%06X", header.start);
        return "";
}



//object program의 Text 영역 문자열
private String[] getTextStrings() {
        ArrayList<String> text_strings = new ArrayList<String>();

        int locctr = 0;
        int line_bytes = 0;
        boolean line_break = true;
        StringBuilder sb = null;

        for(Text text : textList) {
                int text_bytes = text.length / 2;
                if(line_bytes + text_bytes > 30 || locctr != text.location) {
                        locctr = text.location;
                        line_break = true;
                }
                if(line_break) {
                        if(sb != null) {
                                sb.insert(6, String.format("%02X", line_bytes));
                                text_strings.add(sb.toString());
                        }
                        sb = new StringBuilder(String.format("%06X", text.location));
                        line_bytes = 0;
                        line_break = false;
                }
                sb.append(text);
```

```
                line_bytes += text_bytes;
                locctr += text_bytes;
            }

            if(sb != null) {
                sb.insert(6, String.format("%02X", line_bytes));
                text_strings.add(sb.toString());
            }

            return text_strings.toArray(new String[0]);
        }


        //헤더, 텍스트, modification record 등으로 이루어진 object program을 printStream으로 작성
        public void print(PrintStream ps) {
            ps.println("H" + header);
            String extdef_string = getExtdefString();
            if (extdef_string != null && !extdef_string.isEmpty()) {
                ps.println("D" + extdef_string);
            }
            String extref_string = getExtrefString();
            if (extref_string != null && !extref_string.isEmpty()) {
                ps.println("R" + extref_string);
            }
            for(String text_string : getTextStrings()) {
                ps.println("T" + text_string);
            }
            for(Modification mod : modList) {
                ps.println("M" + mod);
            }
            ps.println("E" + getEndString());
        }



        //프로그램의 길이 setter 함수
        public void setLength(int length) {
            header.length = length;
        }



        //EXTDEF 리스트에 레이블과 위치 추가
        public void addExtdef(String extdef, int location) {
            extdefList.add(new AbstractMap.SimpleEntry<>(extdef, location));
        }



        //EXTREF 리스트에 레이블 추가
        public void addExtref(String extref) {
            extrefList.add(extref);
        }
}



/*토큰이랑 리터럴 테이블에서 변환된 object code 클래스다. */
class Text implements Comparable<Text> {
        //object code data
        int data;
```

```
    //object code 16진법 변환 시의 길이
    int length;
    //object code의 위치
    int location;



    //생성자, 위치, 길이를 저장하고 object code(data)를 opcode로 초기화
    public Text(int location, int length, byte opcode) {
            this.data = 0;
            this.location = location;
            this.length = length;
            if((opcode & 0xFF) != 0xFF) {
                    addData(opcode, length - 2);
            }
    }



    //16진법으로 오른쪽에서 shifts번째 자리를 addition으로 세팅
    public void addData(int addition, int shifts) {
            data |= addition << (shifts * 4);
    }



    //object code에 nixbpe 값 세팅
    public void addNixbpe(char nixbpe) {
            if(length == 6 || length == 8)
                    addData(nixbpe, length - 3);
    }



    //object code에 피연산자 값 혹은 주소 차이 값 세팅
    public void addOperandValue(int operandValue) {
            int mask = (1 << ((length - 3) * 4)) - 1;
        int lower_bits = operandValue & (int)mask;
            addData(lower_bits, 0);
    }



    //object code를 바꿈
    public void setData(int new_data) {
            data = new_data;
    }



    //위치 기준 정렬을 위한 비교 함수
    @Override
public int compareTo(Text other) {
     return Integer.compare(this.location, other.location);
}



    //저장된 16진법 길이만큼 object code를 문자열로 변환
    @Override
    public String toString() {
            long mask = (1L << (length * 4)) - 1;
        int lower_bits = data & (int)mask;
```

```
            return String.format("%0" + length + "X", lower_bits);
        }
}
```

```
/*Section의 헤더 작성을 위한 프로그램의 시작 위치, 길이, 이름 등을 저장하는 클래스다.*/
class Header {
        int start;
        int length;
        String name;


        //기본 생성자
        public Header() { }


        //object program의 헤더 정보 문자열
        @Override
        public String toString() {
                return String.format("%-6s%06X%06X", name, start, length);
        }
}
```

```
/*Section의 Modification record 작성을 위한 수정 위치, 길이, 연산자(+,-), 대상 심 심볼 등을 저장하는 클래스다.*/
class Modification {
        int location;
        int length;
        char operator;
        String symbol;


        //생성자, 수정위치, 수정길이, 연산자, 대상 심볼 저장
        public Modification(int location, int length, char operator, String symbol) {
                this.location = location;
                this.length = length;
                this.operator = operator;
                this.symbol = symbol;
        }


        //object program의 modification record 정보 문자열
        @Override
        public String toString() {
                return String.format("%06X%02X%c%s", location, length, operator, symbol);
        }
}
```