20171969

박건호

과제명 : 시스템 프로그래밍 Project #1(b) SIC/XE assembler Programming

출석번호 : 110

수업 구분 : 가

1. 동기 / 목적

수업시간에 배운 control sectiions 방식의 어셈블리어 SIC/XE 프로그램을 Object Program Code로 변환해주는 어셈블러를 JAVA로 완성하는 것이 목적이다.

어셈블러를 구현하기 위해, 어셈블리어 프로그램 소스코드를 입력 받아, 소스코드의 명령어 정보가 담긴 inst_table.txt를 토대로 Location Counter 할당, Symbol Table 작성, Literal Tabel 작성을 수행하여 결과적으로 소스코드의 한 라인마다 object code를 만들어낸다.

최종적으로 만들어진 어셈블러 오브젝트 코드는 어셈블리 언어로 작성된 소스 코드를 기계어로 번역한 실행 가능한 프로그램으로 변환된 코드이다.

본 과제 프로그램은 어셈블리어 소스코드와 기계어 목록을 읽어오고, 어셈블리 언어로 작성된 소스 코드를 토큰화하고, 각 소스코드 라인에 로케이션 카운터를 할당하고, 심볼을 심볼 테이블에 추가하고, 리터럴을 리터럴 테이블에 추가하고, nixbpe 값을 계산하고, 결과적으로 op코드의 값을 판단하고, 타겟 어드레스 계산을 수행하여 라인마다 알맞은 기계와 코드를 만들어내는 게 목적이다.

Pass1에서는 로케이션 카운터를 알맞게 증가시키고, nixbpe 할당을 수행하고, 등장하는 심볼을 심볼 테이블에 추가하고, 등장하는 리터럴을 리터럴 테이블에 추가하여 관리하는 기능을 수행한다.

Pass2 에서는 Pass1에서 만들었던 항목들을 통해 실제로 라인마다 기계어 코드를 계산하고 Directive 연산자에 따른 오브젝트 코드 생성을 수행한다.

본 JAVA 프로그램의 주요 기능은 크게 1. 소스코드 파싱, 2. 심볼 테이블의 관리, 3. 리터럴의 처리, 4. 명령어 해석, nixbpe 할당 5. 오브젝트 코드의 생성 6. 결과 출력 등이 있다.

2. 설계 / 구현 아이디어

Assembler.java

PASS1 과 PASS2를 호출하며, 처음 생성자에서 기계어 명령어 목록 파일을 읽어들여 _instTable 필드로 관리한다.

입력받을 소스코드를 divideInput 메소드를 통해 섹션별로 나누어 오브젝트 코드 생성과정을 거친다. 각 섹션별로 따로 심볼테이블, 리터럴테이블을 생성하며, 결과적인 오브젝트 코드 생성도 각각 따로 수행하게 된다.

ControlSection.java

PASS1과 PASS2의 과정이 담긴 클래스이다.

PASS1 ControlSection 에서는 먼저 소스코드를 StringToken을 통해 라인별로 파싱한다.

그리고 심볼테이블과 리터럴 테이블과 토큰리스트를 생성한다.

연산자 오류처리를 하고 난후, 영산자를 기준으로 PASS1을 수행한다.

연산자가 기계어명령 테이블에 존재한다면 instruction PASS1 을 수행,

존재하지 않는다면 Directive이기에 Directive PASS1 을 수행한다.

각각의 결과에 따라 Token을 상속하여 instruction토큰과 directive토큰으로 구분된다.

한 섹션의 모든 라인마다 반복하여 모든 PASS1 과정이 끝나면

토큰리스트 심볼테이블 리터럴테이블이 작성되었다.

PASS2 buildObjectCode 에서는 먼저 오브젝트 코드 인스턴스를 생성하고 (여기에 실제 오브젝트 코드가 담긴다)

PASS1에서 결과적으로 만들어진 토큰리스트에 대해 하나하나의 토큰을 instruction토큰인지 directive토큰인지 판단하여 PASS2를 수행한다.

PASS1

handlePass1InstructionStep

로케이션 카운터 할당을 위해 본 소스코드 라인의 명령어 형식을 판단하여 크기를 결정하고, 레이블과 피연산자에서 등장하는 심볼들을 심볼테이블에 추가하고, 리터럴이 등장

했다면 리터럴테이블에 추가하고, 피연산자가 레지스터인 경우에 대해 처리한다. 그리고 피연산자와 형식들에 따라 nixbpe 비트 구분을 위한 변수들도 적절히 조정한다.

handlePass1DirectiveStep

디렉티브마다 적절한 PASS1을 수행한다. 레이블이 있는 경우 심볼테이블에 추가하고 결과값 리턴인 디렉티브토큰에 넘겨줄 피연산자를 관리한다.

각 디렉티브마다 적절한 수행을 하였으며, EQU의 연산은 완벽하게 구현하지 못하였다.

현재는 예시 input에 존재하는 "*" 와 "Symbol-Symbol" 피연산자만 수행이 가능하다. 이는 추후 Numeric 클래스를 수정하여 더 많은 경우의 피연산자 처리가 가능하도록 해야 한다.

END 디렉티브의 경우에는 만약 현재 섹션의 심볼테이블에 END의 피연산자가 존재한다면 토큰에 피연산자로서 넘겨주도록 구현하였다. 이를 통해 실제 END 디렉티브를 통해 돌아가야하는 시작지점으로 돌아가는 E 오브젝트코드를 필요한 곳에만 부여 가능하다.

PASS2

handlePass2InstructionStep

명령어 토큰에 대해 실제 오브젝트코드와 modification record(여기선 4형식의 경우)를 계산하여 Textinfo로 관리한다. 이 정보를 오브젝트코드 인스턴스에 넘겨서 오브젝트코드 인스턴스에 오브젝트코드가 쌓여간다.

handlePass2DirectiveStep

디렉티브 토큰에 대해 적절한 수행을 한다.

주요한 기능은 오브젝트코드 인스턴스에 실제로 적힐 오브젝트 코드 정보를 넘겨주는 것이다. WORD의 경우 수식이 나오는 경우 PASS1 에서 넘겨받은 피연산자를 판단하여 modification 레코드에 추가해야 한다.

LTORG가 나오면 지금까지 있었던 계산되지 않은 모든 리터럴에 대해 계산을 수행하고 오브젝트코드 인스턴스에 넘긴다

END가 나오면 LTORG와 같은 작업을 수행하고, 만약 오퍼랜드로 넘겨받은 것이 있다면 이를 심볼테이블에서 검색하여 해당하는 심볼의 address를 E코드에 적힐 값은 설정해준다. 또한 해당 섹션의 총 크기를 설정한다.

RESB와 RESW가 나오면 빈 공간이 생긴것이기 때문에 줄바꿈을 수행하는 코드를 호출한

다.

ObjectCode.java

PASS2에서 넘겨받은 모든 오브젝트 코드 정보들을 가지고 한 섹션의 전체 오브젝트 코드를 통합하는 과정을 수행한다. 이중에서 Text 부분은 길이가 30B를 넘기거나, 줄바꿈 정보가 주어졌다면 줄바꿈을 수행하여 다시 적어나간다.

그 외에 자바 파일들은 위 세 주요한 클래스 들을 구현하기 위해 정의되었다.

Token -> DirectiveToken / InstructionToken으로 상속된다.

StringToken : 소스코드를 ₩t 기준으로 파싱한다.

Operand -> LiteralOperand / NumericOperand / RegisterOperand 로 상속된다.

Symbol / SymbolTable : 심볼과 심볼테이블에 관한 클래스이다.

Literal / LiteralTable : 리터럴과 리터럴테이블에 관한 클래스이다.

Instruction / InstructionTable : 처음으로 수행되는 명령어 목록에 관한 클래스이다.

Directive : 디렉티브들에 관한 구조체이다.

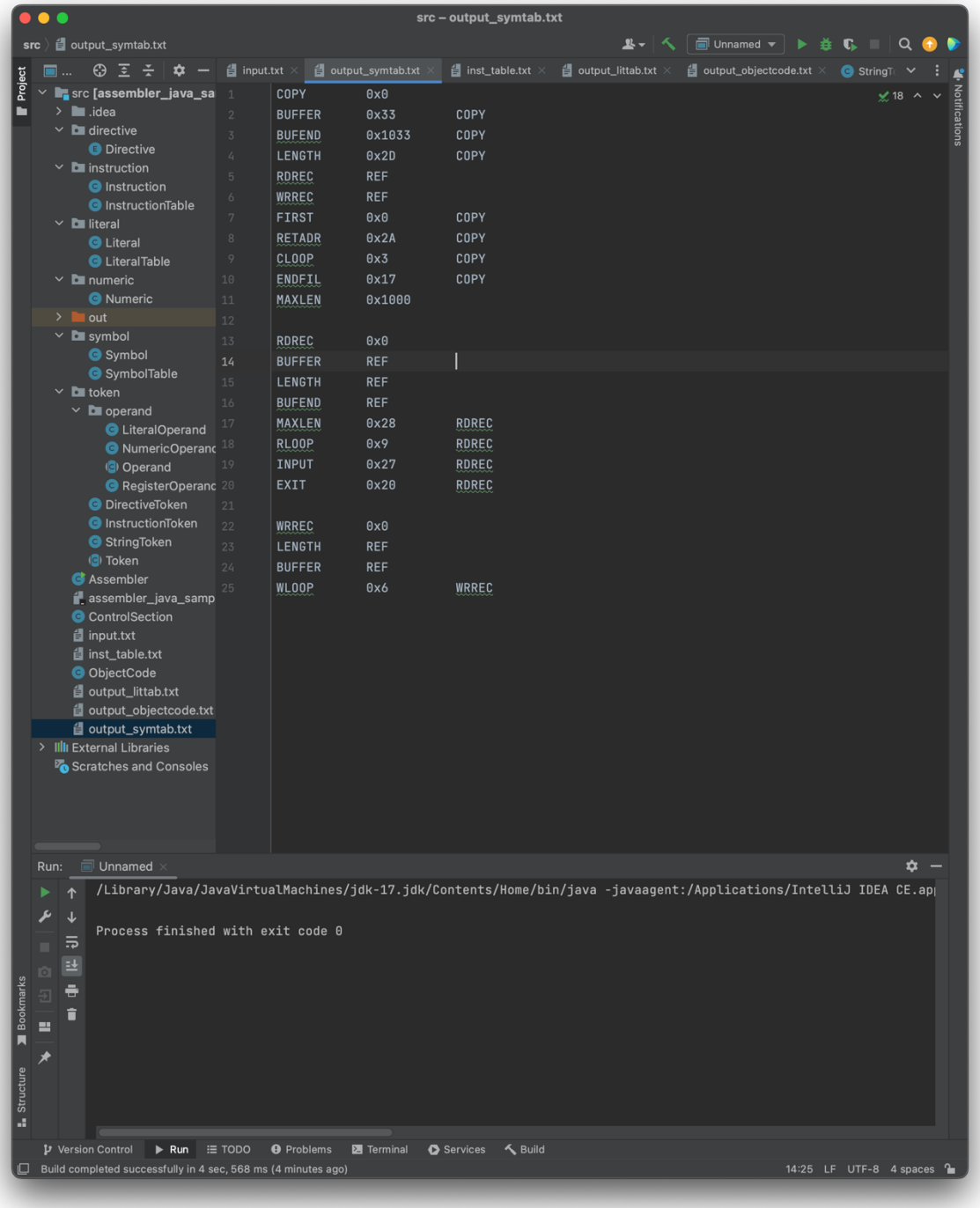
Numeric : 주소값과 수식계산을 용이하게 하고자 정의된 클래스이다. 특히 Symbol은 Numeric과 매우 관련이 깊다. Numeric은 value와 <심볼, 계수> 해시맵의 필드를 가지는데 이를 통해 수식연산과 modification record 관리 등등 여러 작업에 편의성을 준다.

특히 EQU의 피연산자로 복잡한 수식 Symbol - Constant / Constant + Constant / Symbol - Symbol 등의 연산을 Numeric을 완전하게 구현하면 쉽게 수행이 가능하다.

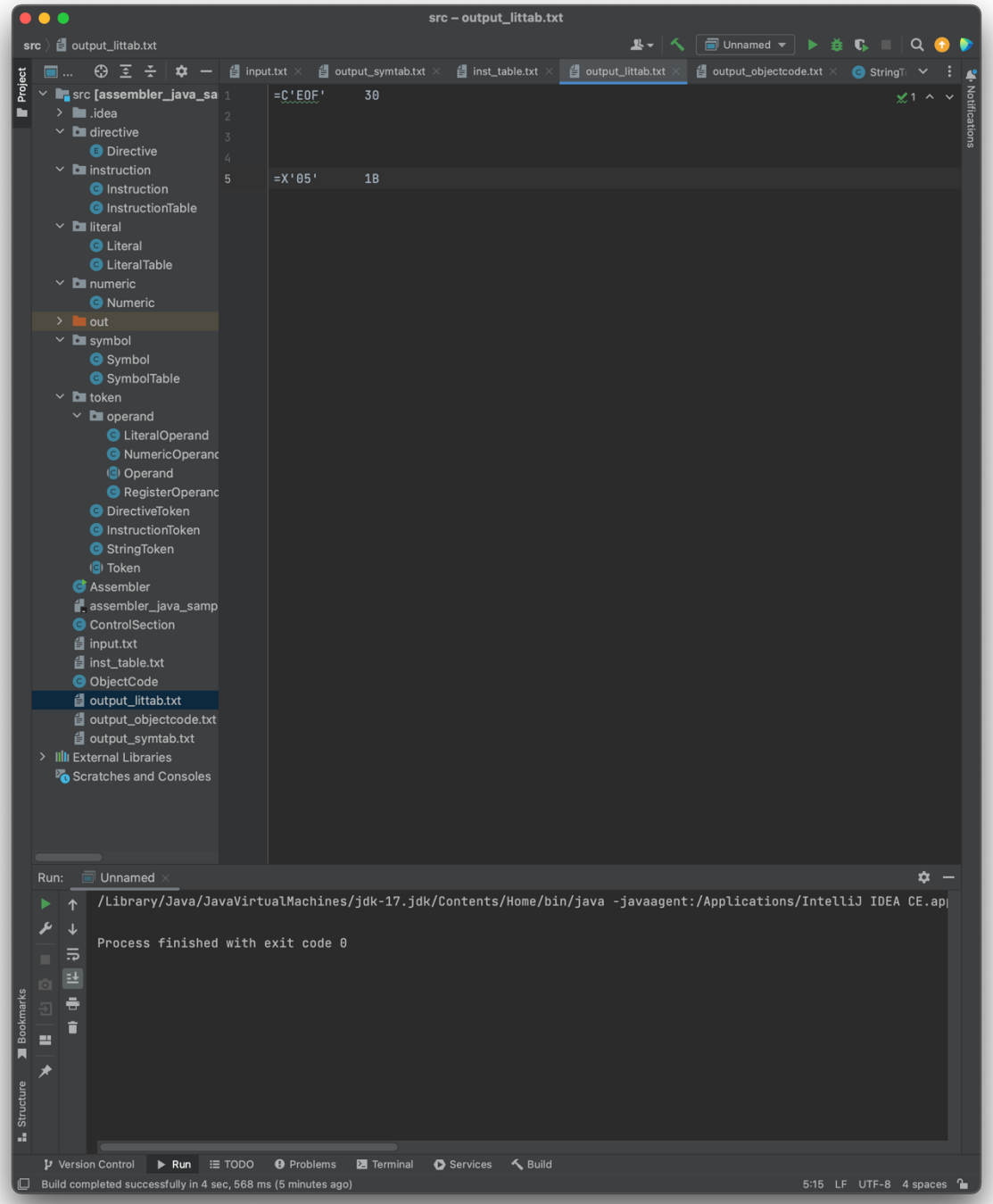
그러나 본 과제 수행을 하며, 아직 완성하지 못하였으며, 현재는 "*" 와 "Symbol-Symbol" 의 EQU 피연산자만 수행이 가능하다.

이외의 세부적인 구현사항은 주석으로 상세한 설명을 달아놓았다.

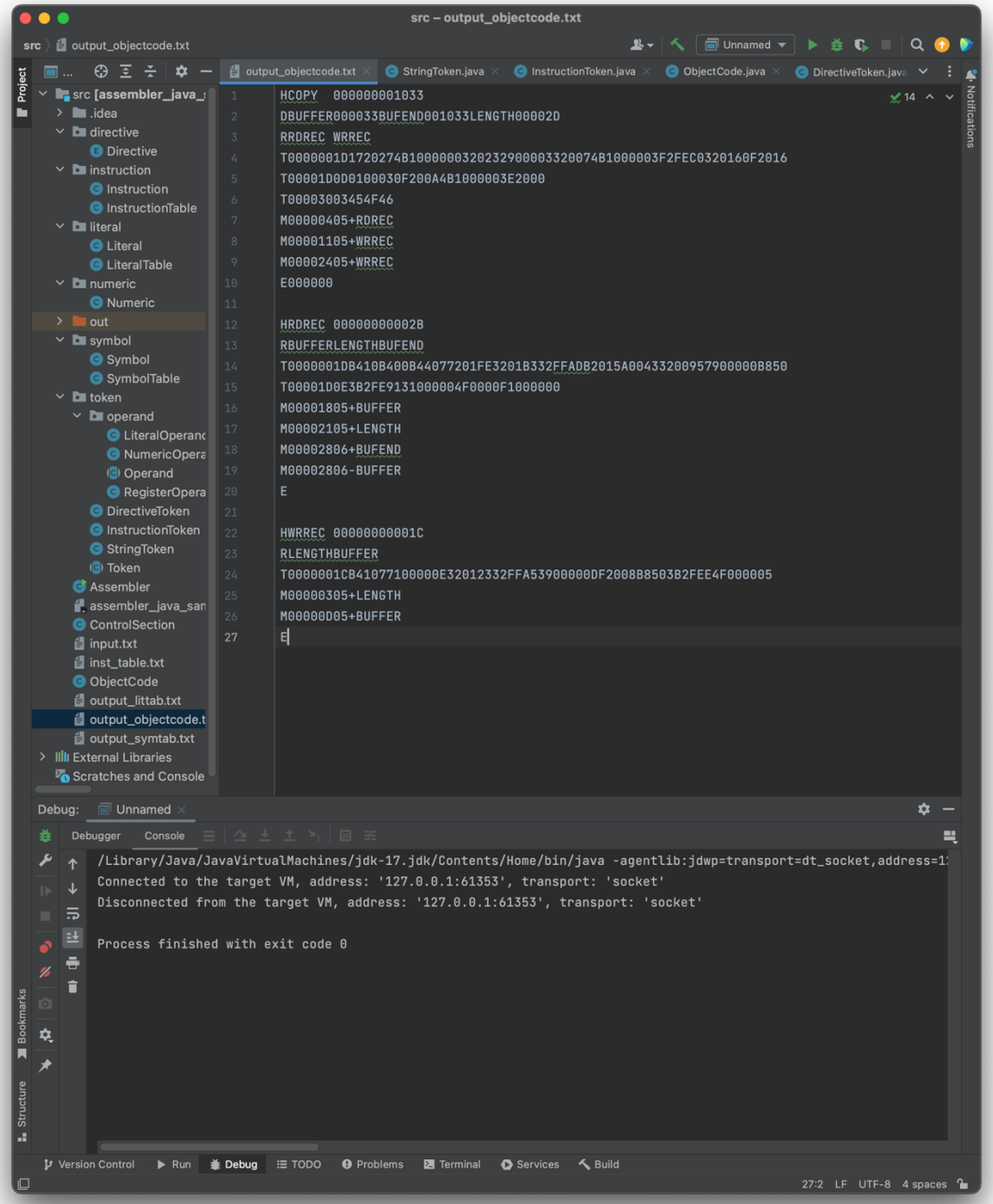
3. 수행결과



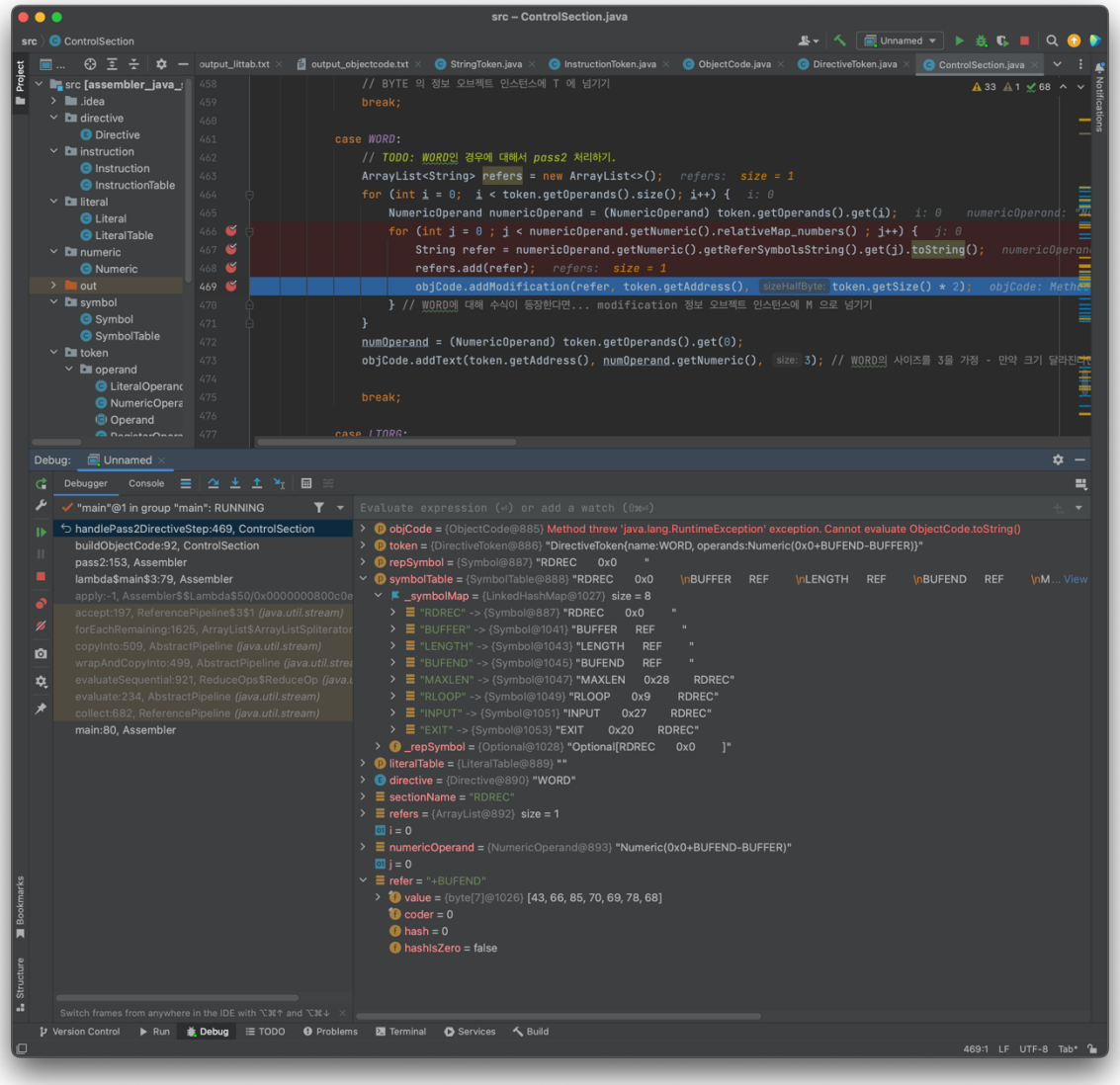
input.txt에 대한 심볼테이블 출력 결과이다.



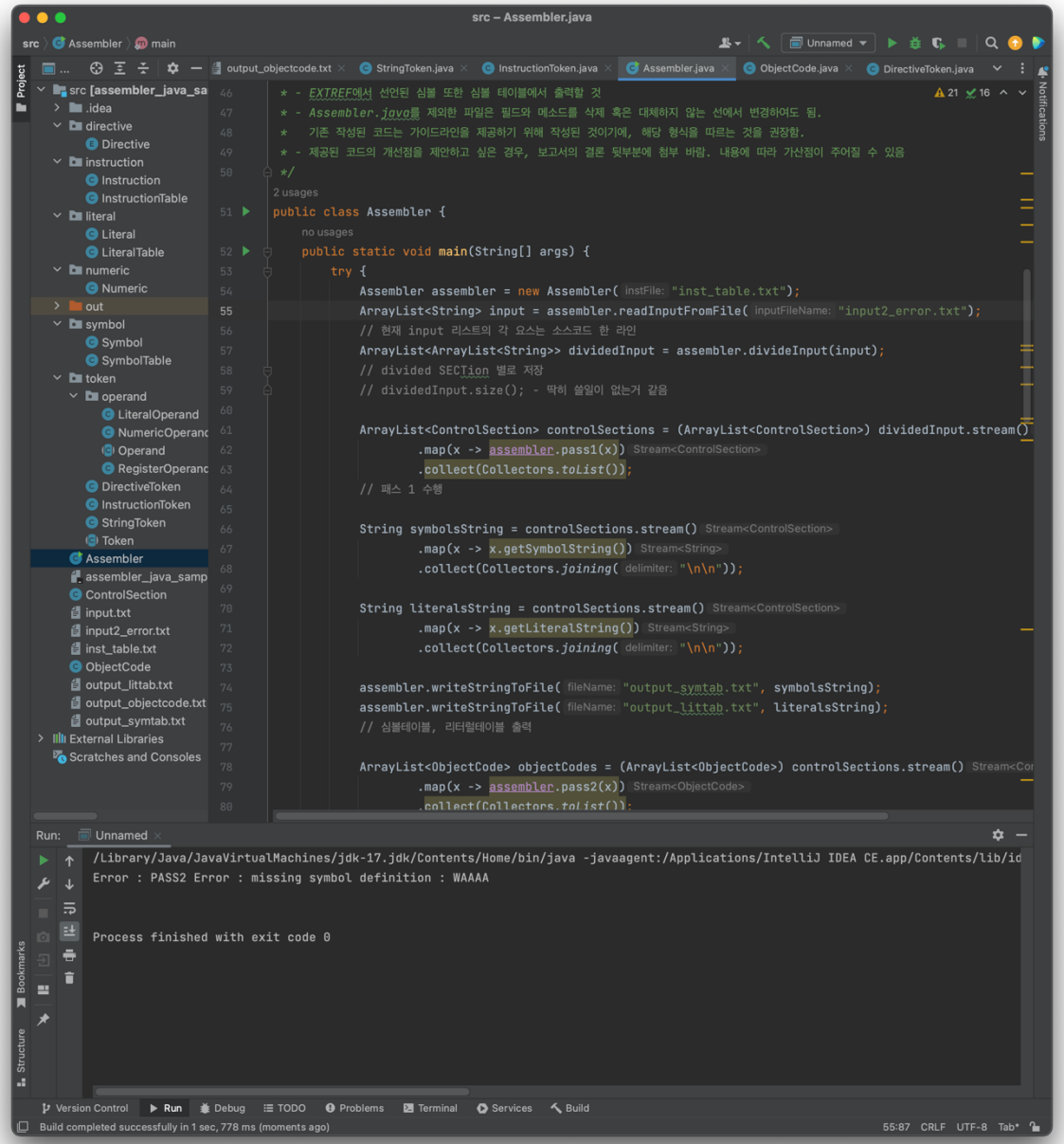
input.txt에 대한 리터럴테이블 출력 결과이다.



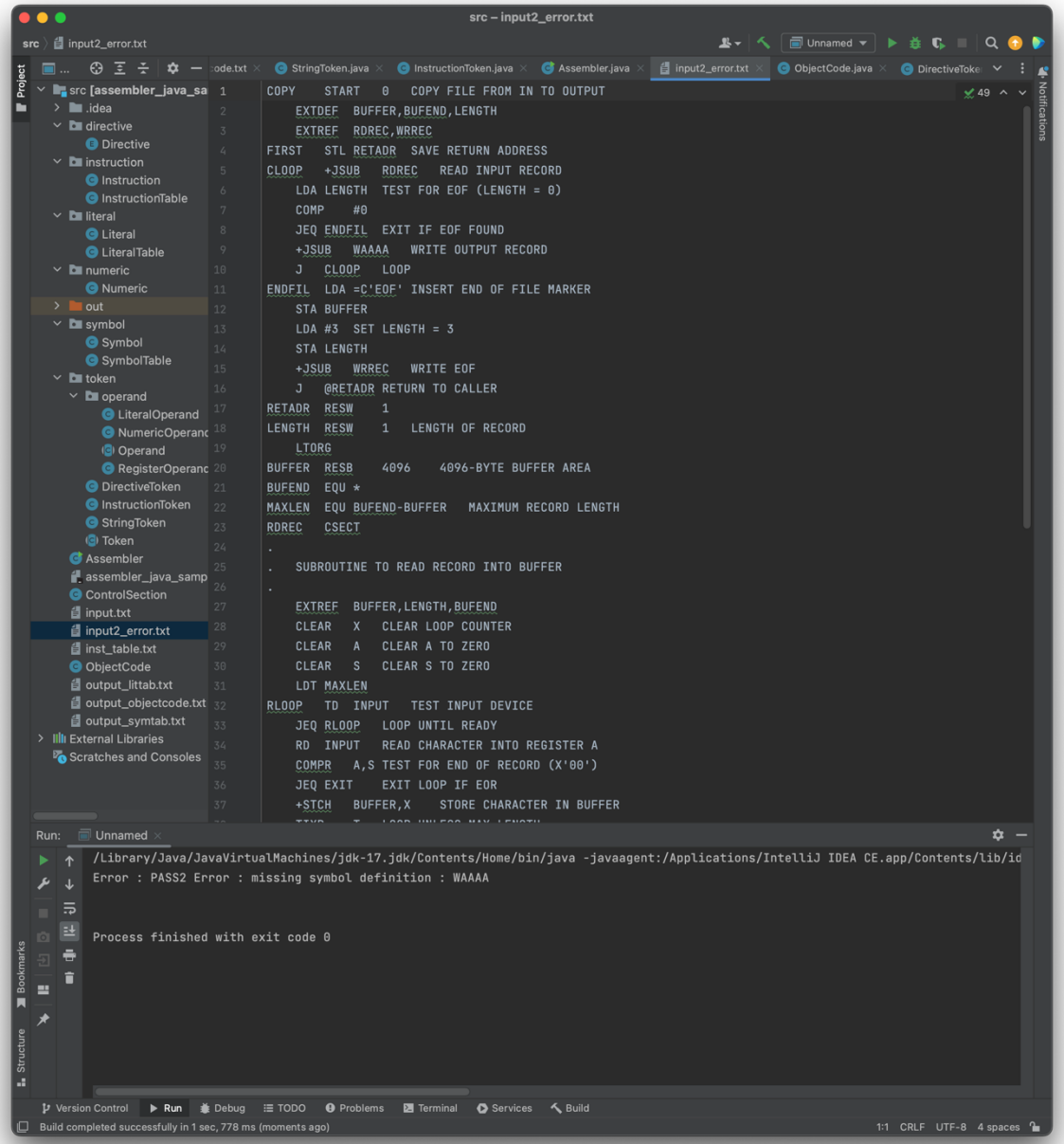
input.txt에 대한 오브젝트코드 출력 결과이다.



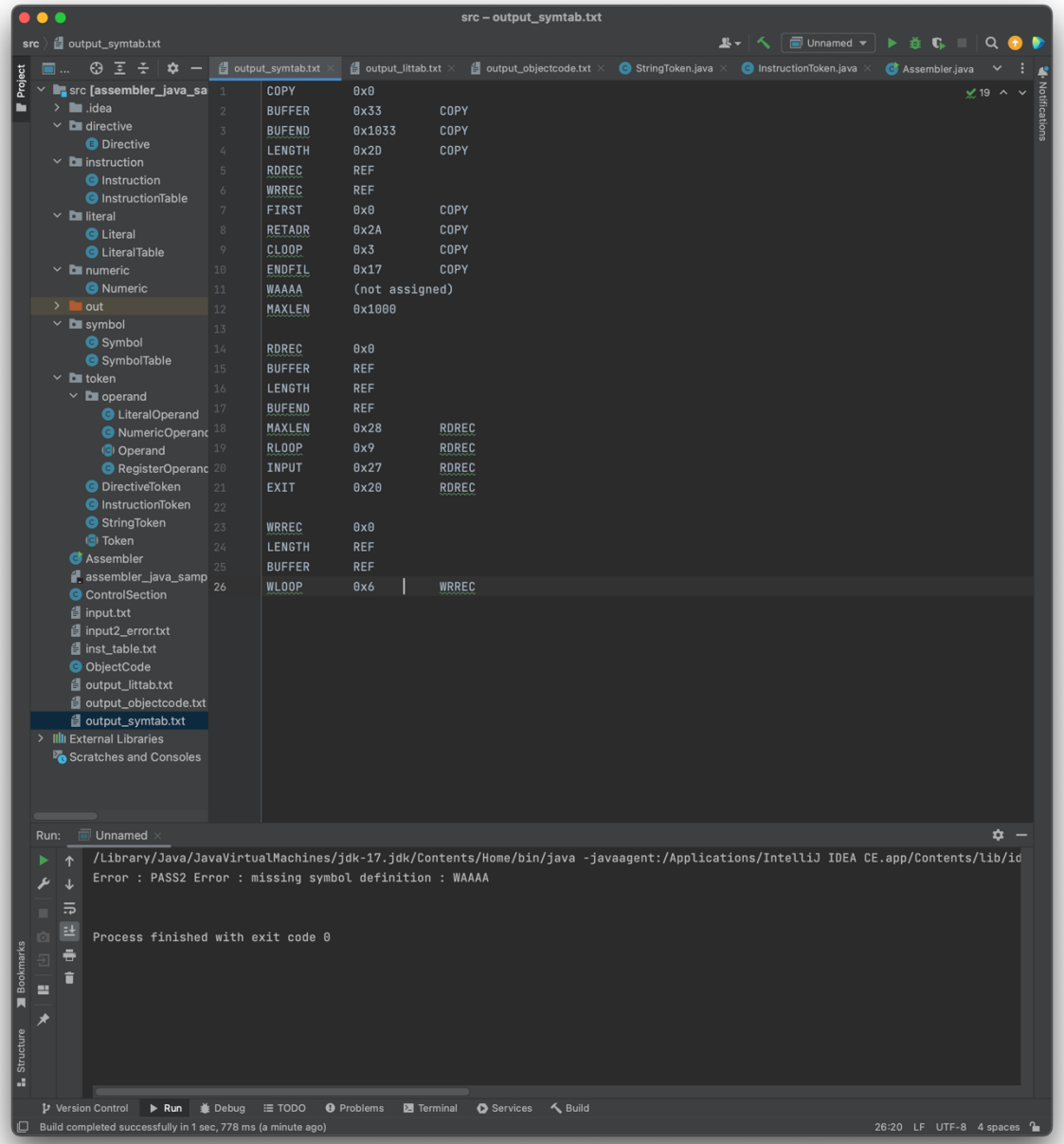
디버깅 중단점을 설정하여 섹션의 심볼테이블과 변수 값들을 확인하며 작업을 수행하였다.



input2_error.txt로 입력파일을 변경하여 작업을 수행하였을 때, Exception 처리가 된 모습이다.



input2_error.txt는 WAAA가 레이블에서 등장하지 않아 어드레스가 할당되지 않았다.



심볼테이블 출력 결과를 보면 할당되지 않았음을 알 수 있다.

실행환경 : MacOS Sonoma 14.4.1 / JAVA jdk-17 / intelliJ

4. 결론 및 보충할 점

현재 소스코드에서는 Exception처리를 대표적인 항목만 수행하였다.

명세서에서 주어진 주소가 할당되지 않은 심볼의 주소에 접근하려할 때의 처리,

존재하지 않는 연산자 (명령어도, 디렉티브도 아님)에 대한 처리,

심볼테이블에 심볼 추가시 심볼 명칭으로 유효하지 않은 String이 들어온 경우에 대한 처리,

심볼테이블에 Section 대표 심볼과 EXTDEF 심볼의 중복 추가에 대한 처리,

심볼의 주소 할당을 여러번 하는 경우에 대한 처리.

등을 수행하였다.

더 예외처리를 해야하는 점은 다음과 같다.

SymTab에 접근시에는 Optional isPresent isEmpty 메소드들을 사용하여, 존재하는지의 여부를 확인하고 상황에 따른 예외처리를 수행한다.

FILE IO 에 실패한 경우에 대해 예외처리를 수행한다.

input을 성공적으로 파싱하지 못한 경우에 대해 예외처리를 수행한다.

등등

현재 코드에서는 EQU에 대한 처리가 미흡하다. 현재는 "*" , "Symbol-Symbol" 에 대한 경우만 처리가 가능하다.

Numeric은 피연산자에 복잡한 수식이 나오는 경우를 원활하게 처리하기 위한 클래스이다. 해당 클래스에 있는 메소드들을 더 정확하게 완성하고 적절하게 사용하여 모든 피연산자의 경우에 대해 처리가 가능하도록 수정해야 한다. (시간관계상 수행하지 못하였다)

추가적으로 제안할 점은 파싱과정이다.

현재 파싱에서 nixbpe중 xBit 에 대한 처리를 하기 위해 X가 포함되어있는지 확인하여 _xBit를 설정하고 있다.

그러나 만약 피연산자로 X가 나오는 경우 그것이 레지스터 X라면 문제가 발생할 수 있

다. 내가 제안하는 것은 파싱과정에도 instruction 테이블을 가지고 와서 해당 연산자가 필요로 하는 피연산자의 개수, 형태 정보를 사용하여 파싱을 진행하는 것이다.

하지만 본 과제에서는 그냥 하드코딩으로 라인파싱을 수행하였다. 내 생각에는 파싱에서도 명령어 정보를 사용하여 파싱을 함이 더 정확한 파싱 수행이 가능할 것 같다.

5. 소스코드 (+주석)

예시 코드에서 추가 작성이 있었던 코드만 포함하였다.

보고서에 포함하지 않은 java파일은 예시 코드에서 달라진 부분이 없다.

Assembler.java

```
/**
 * @author Enoch Jung (github.com/enochjung)
 * @file Assembler.java
 * @date 2024-05-05
 * @version 1.0.0
 *
 * @brief 조교가 구현한 SIC/XE 어셈블러 코드 구조 샘플
 */

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.stream.Collectors;

import instruction.InstructionTable;
```

```
/**
 * SIC/XE 머신을 위한 Assembler 프로그램의 메인 루틴이다.
 *
 * 작성 중 유의 사항
 * 1) Assembler.java 파일의 기존 코드를 변경하지 말 것. 다른 소스 코드의 구조는 본인의 편의에 따라 변경해도 됨.
 * 2) 새로운 클래스, 새로운 필드, 새로운 메소드 선언은 허용됨. 단, 기존의 필드와 메소드를 삭제하거나 대체하는 것은 불가함.
 * 3) 예외 처리, 인터페이스, 상속 사용 또한 허용됨.
 * 4) 파일, 또는 콘솔창에 한글을 출력하지 말 것. (채점 상의 이유)
 *
 * 제공하는 프로그램 구조의 개선점을 제안하고 싶은 학생은 보고서의 결론 뒷부분에 첨부 바람. 내용에 따라 가산점이 있을 수 있음.
 *
 * 필드와 메소드를 삭제 혹은 대체하지 않는 선에서 변경하여도 됨'의 의미는
 * 주어진 필드와 메소드를 적극적으로 사용하되, 이들의 역할을 바꾸지 말아 달라는 의미입니다.
 * 기준이 상당히 주관적이므로 적절히 판단해 주시길 바랍니다.
 *
 * 심볼 테이블 및 리터럴 테이블 출력에서 각 요소 출력 순서는 상관없습니다.
 * 로더가 같은 동작을 수행함을 보장한다면 오브젝트 코드의 출력 순서도 상관없습니다..만
 * 보장하지 않는 경우가 대부분이므로, 오브젝트 코드는 출력 예시에 맞추는 것을 권장합니다.
 *
 * - 주어진 java 파일을 참고하여 과제 코드를 구현할 것
```

```
 * - ControlSection.java 파일에서 각 control section에 해당하는 소스 코드가 독립적으로 컴파일됨.
 *    독립적인 컴파일을 보장하기 위하여, Assembler.java 에서 각 control section 소스 코드에 END directive를 추가하였음
 * - Exception Handling을 통해 SIC/XE 소스 코드의 문법적 오류를 탐지할 수 있도록 구현할 것.
 *    최소한 exception handling을 통해 정의되지 않은 심볼 사용 오류를 감지하여야 하며, 다른 오류 감지를 추가적으로 수행할 경우 가산점이 주어질 수 있음
 * - EXTREF에서 선언된 심볼 또한 심볼 테이블에서 출력할 것
 * - Assembler.java를 제외한 파일은 필드와 메소드를 삭제 혹은 대체하지 않는 선에서 변경하여도 됨.
 *    기존 작성된 코드는 가이드라인을 제공하기 위해 작성된 것이기에, 해당 형식을 따르는 것을 권장함.
 * - 제공된 코드의 개선점을 제안하고 싶은 경우, 보고서의 결론 뒷부분에 첨부 바람. 내용에 따라 가산점이 주어질 수 있음
 */
public class Assembler {
	public static void main(String[] args) {
		try {
			Assembler assembler = new Assembler("inst_table.txt");
			ArrayList<String> input = assembler.readInputFromFile("input.txt");
			// 현재 input 리스트의 각 요스는 소스코드 한 라인
			ArrayList<ArrayList<String>> dividedInput = assembler.divideInput(input);
			// divided SECTion 별로 저장
			// dividedInput.size(); - 딱히 쓸일이 없는거 같음
```

```
                ArrayList<ControlSection> controlSections =
(ArrayList<ControlSection>) dividedInput.stream()
                        .map(x -> assembler.pass1(x))
                        .collect(Collectors.toList());
                // 패스 1 수행

                String symbolsString = controlSections.stream()
                        .map(x -> x.getSymbolString())
                        .collect(Collectors.joining("₩n₩n"));

                String literalsString = controlSections.stream()
                        .map(x -> x.getLiteralString())
                        .collect(Collectors.joining("₩n₩n"));

                assembler.writeStringToFile("output_symtab.txt", symbolsString);
                assembler.writeStringToFile("output_littab.txt", literalsString);
                // 심볼테이블, 리터럴테이블 출력

                ArrayList<ObjectCode> objectCodes = (ArrayList<ObjectCode>)
controlSections.stream()
                        .map(x -> assembler.pass2(x))
                        .collect(Collectors.toList());
                // 패스 2 수행

                String objectCodesString = objectCodes.stream()
                        .map(x -> x.toString())
```

```
                                        .collect(Collectors.joining("₩n₩n"));

                        assembler.writeStringToFile("output_objectcode.txt",
objectCodesString);
                        // 오브젝트 코드 출력
                } catch (Exception e) {
                        System.out.println("Error : " + e.getMessage());
                }

        }

        public Assembler(String instFile) throws FileNotFoundException, IOException {
                _instTable = new InstructionTable(instFile);
                // Assembler.java 생성하면서 기계어 명령어 테이블 목록 만든다
        }

        private ArrayList<ArrayList<String>> divideInput(ArrayList<String> input) {
                ArrayList<ArrayList<String>> divided = new
ArrayList<ArrayList<String>>();
                String lastStr = input.get(input.size() - 1);

                ArrayList<String> tmpInput = new ArrayList<String>();
                for (String str : input) { // 한 라인씩 검사 수행
                        if (str.contains("CSECT")) { // 라인에 CSECT가 있다면
                                if (!tmpInput.isEmpty()) { // tmpInput이 비어있지 않은 경
우
```

```
                                        tmpInput.add(lastStr); // END FIRST 삽입!! -> 나
중에 심볼테이블에서 확인 가능
                                        divided.add(tmpInput);
                                        tmpInput = new ArrayList<String>();
                                        tmpInput.add(str);
                                }
                        } else {
                                tmpInput.add(str);
                        }
                }

                if (!tmpInput.isEmpty()) {
                        divided.add(tmpInput);
                }

                return divided; // 섹션별로 독립적인 컴파일 위해 나누었음
        }

        private ArrayList<String> readInputFromFile(String inputFileName) throws
FileNotFoundException, IOException {
                ArrayList<String> input = new ArrayList<String>();

                File file = new File(inputFileName);
                BufferedReader bufReader = new BufferedReader(new FileReader(file));

                String line = "";
```

```
            while ((line = bufReader.readLine()) != null) // 한 라인을 읽어서 null이 아니라면 line으로 읽어오기
                input.add(line); // input에 line을 추가

            bufReader.close();

            return input;
        }

        private void writeStringToFile(String fileName, String content) throws IOException {
            File file = new File(fileName);

            BufferedWriter writer = new BufferedWriter(new FileWriter(file));
            writer.write(content);
            writer.close();
        }

        private ControlSection pass1(ArrayList<String> input) throws RuntimeException {
            return new ControlSection(_instTable, input); // inst_table , input을 받아서 pass1 수행
        }

        private ObjectCode pass2(ControlSection controlSection) throws RuntimeException {
            return controlSection.buildObjectCode(); // PASS2 수행
```

```
        }


        private InstructionTable _instTable;
}
```

ControlSection.java

```
/**
 * @author Enoch Jung (github.com/enochjung)
 * @file ControlSection.java
 * @date 2024-05-05
 * @version 1.0.0
 *
 * @brief 조교가 구현한 SIC/XE 어셈블러 코드 구조 샘플
 */

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

import directive.Directive;
import instruction.*;
import literal.*;
import symbol.*;
import token.*;
import token.operand.*;
import numeric.Numeric;

public class ControlSection {
        /**
```

```
	* pass1 작업을 수행한다. 기계어 목록 테이블을 통해 소스 코드를 토큰화하고, 심볼 테이블 및 리터럴 테이블을 초기화환다.
	*
	* @param instTable 기계어 목록 테이블
	* @param input 하나의 control section에 속하는 소스 코드. 마지막 줄은 END directive를 강제로
	* 추가하였음.
	* @throws RuntimeException 소스 코드 컴파일 오류.
	*/
	public ControlSection(InstructionTable instTable, ArrayList<String> input) throws RuntimeException {
		List<StringToken> stringTokens = input.stream()
				.map(x -> new StringToken(x))
				.collect(Collectors.toList());
		// StringToken 의 List -> StringToken 인스턴스는... 레이블 오퍼레이터 오퍼랜드 코멘트 / nixpe 포함

		SymbolTable symTab = new SymbolTable(); // 심볼테이블 생성
		LiteralTable litTab = new LiteralTable(); // 리터럴테이블 생성

		ArrayList<Token> tokens = new ArrayList<Token>(); // 토큰 리스트 생성

		int locctr = 0;
		for (StringToken stringToken : stringTokens) { // 라인마다 수행
			if (stringToken.getOperator().isEmpty()) { // 연산자가 없는 경우
				boolean isLabelEmpty = stringToken.getLabel().isEmpty(); // true : 레이블 없음
```

```
                    boolean isOperandEmpty =
stringToken.getOperands().isEmpty(); // true : operand 없음
                    if (!isLabelEmpty || !isOperandEmpty) // 연산자가 없는데,
레이블이나 피연산자가 있음
                        throw new RuntimeException("missing
operator₩n₩n" + stringToken.toString());
                    continue;
                }
                String operator = stringToken.getOperator().get(); // 연산자

                Optional<Instruction> optInst = instTable.search(operator); //
instTable에서 찾아보고
                // 있다 -> instruction | 없다 -> directive
                boolean isOperatorInstruction = optInst.isPresent();
                if (isOperatorInstruction) { // instruction인 경우 토큰 처리하고 리
스트에 삽입
                    Token token = handlePass1InstructionStep(optInst.get(),
stringToken, locctr, symTab, litTab);
                    locctr += token.getSize();
                    tokens.add(token);
                    // System.out.println(token.toString()); /** 디버깅 용도 */
                } else { // directive인 경우 토큰 처리하고 리스트에 삽입
                    Token token = handlePass1DirectiveStep(stringToken,
locctr, symTab, litTab);
                    locctr += token.getSize();
                    tokens.add(token);
                    // System.out.println(token.toString()); /** 디버깅 용도 */
```

```
                }
            } // 라인마다 수행 끝

            _tokens = tokens;
            _symbolTable = symTab;
            _literalTable = litTab;
        }

        /**
         * pass2 작업을 수행한다. pass1에서 초기화한 토큰 테이블, 심볼 테이블 및 리터럴 테이블을 통해 오브젝트 코드를 생성한다.
         *
         * @return 해당 control section에 해당하는 오브젝트 코드 객체
         * @throws RuntimeException 소스 코드 컴파일 오류.
         */
        public ObjectCode buildObjectCode() throws RuntimeException {
            ObjectCode objCode = new ObjectCode();
            Optional<Symbol> optRepSymbol = _symbolTable.getRepSymbol();
            if (optRepSymbol.isEmpty())
                throw new RuntimeException("invalid operation");
            Symbol repSymbol = optRepSymbol.get();

            for (Token token : _tokens) {
                if (token instanceof InstructionToken) { // inst PASS2
                    handlePass2InstructionStep(objCode, (InstructionToken) token, _symbolTable, _literalTable);
```

```
                } else if (token instanceof DirectiveToken) { // direc PASS2
                        handlePass2DirectiveStep(objCode, (DirectiveToken)
token, repSymbol, _symbolTable, _literalTable);
                } else
                        throw new RuntimeException("invalid operation");
        }

        return objCode;
    }

    /**
     * 심볼 테이블 객체의 정보를 문자열로 반환한다. Assembler.java에서 심볼 테이
블 출력 용도로 사용한다.
     *
     * @return 심볼 테이블의 정보를 담은 문자열
     */
    public String getSymbolString() {
        return _symbolTable.toString();
    }

    /**
     * 리터럴 테이블 객체의 정보를 문자열로 반환한다. Assembler.java에서 리터럴
테이블 출력 용도로 사용한다.
     *
     * @return 리터럴 테이블의 정보를 담은 문자열
     */
```

```
	public String getLiteralString() {
		return _literalTable.toString();
	}

	/**
	 * pass1에서 operator가 instruction에 해당하는 경우에 대해서 처리한다. label 및 operand에 출현한 심볼 및
	 * 리터럴을 심볼 테이블 및 리터럴 테이블에 추가하고, 문자열 형태로 파싱된 토큰을 InstructionToken으로 가공하여 반환한다.
	 *
	 * @param inst    기계어 정보
	 * @param token   문자열로 파싱된 토큰
	 * @param locctr location counter 값
	 * @param symTab 심볼 테이블
	 * @param litTab 리터럴 테이블
	 * @return 가공된 InstructionToken 객체
	 * @throws RuntimeException 잘못된 명령어 사용 방식.
	 */
	private static InstructionToken handlePass1InstructionStep(Instruction inst, StringToken token, int locctr,
			SymbolTable symTab, LiteralTable litTab) throws RuntimeException {
		Instruction.Format format = inst.getFormat();
		Instruction.OperandType operandType = inst.getOperandType();

		int size = 0;
```

```
ArrayList<Operand> operands = new ArrayList<>();
boolean isN = token.isN();
boolean isI = token.isI();
boolean isX = token.isX();
boolean isP = token.isP();
boolean isE = token.isE();

switch (format) {
    case TWO:
        // TODO: size = 2?;
        size = 2;
        break;

    case THREE_OR_FOUR:
        // TODO: size = 3 or 4?;
        if (token.isE()) size = 4;
        else size = 3;
        break;

    default:
        throw new UnsupportedOperationException("not fully support InstructionInfo.Format");
}

// TODO: label을 심볼 테이블에 추가하기.
```

```
token.getLabel().ifPresent(label -> {
        symTab.put(label, locctr);
});

switch (operandType) {
        case NO_OPERAND:
                // TODO: operand가 없어야 하는 경우에 대해서 처리하기.
                // like ₩tJSUB₩tCOMMENT
                isP = false;
                break;

        case MEMORY:
                // TODO: operand로 MEMORY 하나만 주어져야 하는 경우에 대해서 처리하기.
                String opd0 = token.getOperands().get(0);
                Operand.MemoryType memoryType = Operand.MemoryType.distinguish(opd0);
                switch (memoryType) {
                        case NUMERIC:
                                // TODO: operand로 상수 혹은 심볼이 주어지는 경우에 대해서 처리하기.
                                // 심볼이고, 처음 나왔으면 삽입 | 처음 나오는게 아니면 주소 할당
                                // 문자로 시작하면 심볼 |
                                Numeric numericOperand;
                                // 숫자 판별: 숫자로만 구성되어 있으면
```

true, 아니면 false

if (opd0.matches("-?₩₩d+")) { // 정규 표현식을 사용하여 숫자인지 판단

// 숫자 리터럴로 Numeric 객체 생성

numericOperand = new Numeric(opd0);

} else {

// 심볼이면 심볼 테이블에서 검색 또는 새로 추가

if (symTab.search(opd0).isPresent()) {

}

else {

symTab.put(opd0);

}

Symbol operand_symbol = symTab.search(opd0).get();

numericOperand = new Numeric(0, operand_symbol);

}

NumericOperand token_numericOperand = new NumericOperand(numericOperand);

operands.add(token_numericOperand);

break;

case LITERAL:

```
                                        // TODO: operand로 리터럴이 주어지는
경우에 대해서 처리하기.
                                        // 리터럴테이블 삽입
                                        Optional<Literal> optionalLiteral =
litTab.search(opd0);   // LiteralTable에서 리터럴 검색
                                        if (optionalLiteral.isPresent()) {
                                        }
                                        else {
                                                litTab.putLiteral(opd0);
                                        }
                                        Literal literaloperand =
litTab.search(opd0).get();
                                        LiteralOperand token_literaloperand =
new LiteralOperand(literaloperand);
                                        operands.add(token_literaloperand);
                                        break;

                                default:
                                        throw new
UnsupportedOperationException("not fully support Operand.MemoryType");
                                }
                                break;

                        case REG:
                                // TODO: operand로 REGISTER 하나만 주어져야 하는 경
우에 대해서 처리하기.
                                RegisterOperand registerOperand = new
```

```
RegisterOperand(Operand.Register.stringToRegister(token.getOperands().get(0)));
                                operands.add(registerOperand);
                                break;

                        case REG1_REG2:
                                // TODO: operand로 REGISTER 두개가 주어져야 하는 경우에 대해서 처리하기.
                                String regName1 = token.getOperands().get(0);
                                String regName2 = token.getOperands().get(1);
                                RegisterOperand regOperand1 = new RegisterOperand(Operand.Register.stringToRegister(regName1));
                                RegisterOperand regOperand2 = new RegisterOperand(Operand.Register.stringToRegister(regName2));
                                operands.add(regOperand1);
                                operands.add(regOperand2);
                                break;

                        default:
                                throw new UnsupportedOperationException("not fully support InstructionInfo.OperandType");
                }

                return new InstructionToken(token.toString(), locctr, size, inst, operands, isN, isI, isX, isP, isE);
        }

        /**
```

```
	* pass1에서 operator가 directive에 해당하는 경우에 대해서 처리한다. label 및 operand에 출현한 심볼을 심볼

	* 테이블에 추가하고, 주소가 지정되지 않은 리터럴을 리터럴 테이블에서 찾아 주소를 할당하고, 문자열 형태로 파싱된 토큰을

	* DirectiveToken으로 가공하여 반환한다.

	*

	* @param token  문자열로 파싱된 토큰

	* @param locctr location counter 값

	* @param symTab 심볼 테이블

	* @param litTab 리터럴 테이블

	* @return 가공된 DirectiveToken 객체

	* @throws RuntimeException 잘못된 지시어 사용 방식.

	*/

	private static DirectiveToken handlePass1DirectiveStep(StringToken token, int locctr, SymbolTable symTab,

			LiteralTable litTab) throws RuntimeException {

		String operator = token.getOperator().get();


		Directive directive;

		try {

			directive = Directive.stringToDirective(operator);

		} catch (RuntimeException e) {

			throw new RuntimeException(e.getMessage() + "₩n₩n" + token.toString());

		} // illegal directive name
```

```
int size = 0;
ArrayList<Operand> operands = new ArrayList<>();

switch (directive) {
    case START:
        // TODO: START인 경우에 대해서 pass1 처리하기.
        size = 0; // START does not occupy any space.
        symTab.putRep(token.getLabel().orElseThrow(() -> new
RuntimeException("Label required for START")), locctr);
        NumericOperand numericOperand = new
NumericOperand(new Numeric(token.getOperands().get(0)));
        operands.add(numericOperand);
        break;

    case CSECT:
        // TODO: CSECT인 경우에 대해서 pass1 처리하기.
        size = 0; // CSECT does not occupy any space.
        symTab.putRep(token.getLabel().orElseThrow(() -> new
RuntimeException("Label required for CSECT")), locctr);
        break;

    case EXTDEF:
        // TODO: EXTDEF인 경우에 대해서 pass1 처리하기.
        // operand 전부 심볼에 때려박기
        for (String operand : token.getOperands()) {
            symTab.put(operand); //
```

```
ADDRESS_NOT_ASSIGNED;
                                        Symbol operand_symbol =
symTab.search(operand).get();
                                        NumericOperand token_numericOperand = new
NumericOperand(new Numeric(0, operand_symbol));
                                        operands.add(token_numericOperand);
                                }
                                size = 0;
                                break;

                        case EXTREF:
                                // TODO: EXTREF인 경우에 대해서 pass1 처리하기.
                                for (String operand : token.getOperands()) {
                                        symTab.putRefer(operand);
                                        Symbol operand_symbol =
symTab.search(operand).get();
                                        NumericOperand token_numericOperand = new
NumericOperand(new Numeric(0, operand_symbol));
                                        operands.add(token_numericOperand);
                                }
                                size = 0;
                                break;

                        case BYTE:
                                // TODO: BYTE인 경우에 대해서 pass1 처리하기.
                                String byte_operand = token.getOperands().get(0);
```

```
                    symTab.put(token.getLabel().orElseThrow(() -> new
RuntimeException("Label required for BYTE")), locctr);
                    operands.add(new NumericOperand(new
Numeric(byte_operand)));
                    if (byte_operand.startsWith("C'") &&
byte_operand.endsWith("'")) {
                        size =   byte_operand.length() - 3;
                    } else if (byte_operand.startsWith("X'") &&
byte_operand.endsWith("'")) {
                        size = (byte_operand.length() - 3) / 2;
                    } else if (byte_operand.matches("-?₩₩d+")) {
                        size = 1;
                    } else {
                        throw new RuntimeException("Invalid format for
BYTE operand: " + byte_operand);
                    }
                    break;

                case WORD:
                    // TODO: WORD인 경우에 대해서 pass1 처리하기.
                    String word_operand = token.getOperands().get(0);
                    symTab.put(token.getLabel().orElseThrow(() -> new
RuntimeException("Label required for WORD")), locctr);
                    operands.add(new NumericOperand(new
Numeric(word_operand, symTab, locctr)));
                    size = 3;
```

```
                    break;

                case RESB:
                    // TODO: RESB인 경우에 대해서 pass1 처리하기.
                    symTab.put(token.getLabel().orElseThrow(() -> new
RuntimeException("Label required for RESB")), locctr);
                    size = Integer.parseInt(token.getOperands().get(0));
                    operands.add(new NumericOperand(new
Numeric(token.getOperands().get(0))));
                    break;

                case RESW:
                    // TODO: RESW인 경우에 대해서 pass1 처리하기.
                    symTab.put(token.getLabel().orElseThrow(() -> new
RuntimeException("Label required for RESW")), locctr);
                    size = 3 * Integer.parseInt(token.getOperands().get(0));
                    operands.add(new NumericOperand(new
Numeric(token.getOperands().get(0))));
                    break;

                case LTORG:
                    // TODO: LTORG인 경우에 대해서 pass1 처리하기.
                    size = litTab.assignAddress(locctr);
                    break;

                case EQU:
```

```
                    // TODO: EQU인 경우에 대해서 pass1 처리하기.
                    String equValue = token.getOperands().get(0); // "*" "BUFEND-BUFFER"
                    // 레이블은 심볼테이블에 넣기
                    symTab.put(token.getLabel().orElseThrow(() -> new RuntimeException("Label required for EQU")), equValue, locctr);
                    operands.add(new NumericOperand(new Numeric(equValue, symTab, locctr)));
                    size = 0;
                    break;

                case END:
                    // TODO: END인 경우에 대해서 pass1 처리하기.
                    // 모든 섹션의 마지막에 ₩tEND₩tFIRST삽입했음
                    size = litTab.assignAddress(locctr);
                    if (symTab.search(token.getOperands().get(0)).isPresent()) { // 만약 symTab에 FIRST 존재하면 삽입
                        operands.add(new NumericOperand(new Numeric(0, symTab.search(token.getOperands().get(0)).get())));
                    }
                    // FIRST
                    break;

                default:
                    throw new UnsupportedOperationException("not fully support Directive");
            }
```

```
            return new DirectiveToken(token.toString(), locctr, size, directive,
operands);
        }

        /**
         * pass2에서 operator가 instruction인 경우에 대해서 오브젝트 코드에 정보를
추가한다.
         *
         * @param objCode 오브젝트 코드 객체
         * @param token     InstructionToken 객체
         * @throws RuntimeException 잘못된 심볼 객체 변환 시도.
         */
        private static void handlePass2InstructionStep(ObjectCode objCode,
InstructionToken token, SymbolTable symTab, LiteralTable litTab) throws
RuntimeException {
            Token.TextInfo textInfo = token.getTextInfo(symTab, litTab); // 그러면 지
금 Token에 Text info가 있다
            // 그냥 symTab, litTab 넘겨서 offset 계산했음
            // 라인 정보랑 mod 정보 textInfo 에 담았음
            objCode.addText(textInfo.address, textInfo.code, textInfo.size);
            // Text 배열에 추가
            if (textInfo.mod.isEmpty())
                return; // mod 없으면 리턴
            Token.ModificationInfo modInfo = textInfo.mod.get();
            // mod 돌면서 추가
            for (String refer : modInfo.refers)
```

```
                    objCode.addModification(refer, modInfo.address,
modInfo.sizeHalfByte);
        }

        /**
         * pass2에서 operator가 directive인 경우에 대해서 오브젝트 코드에 정보를 추
가한다.
         *
         * @param objCode         오브젝트 코드 객체
         * @param token           DirectiveToken 객체
         * @param repSymbol       control section 명칭 심볼
         * @param literalTable 리터럴 테이블
         * @throws RuntimeException 잘못된 지시어 사용 방식.
         */
        private static void handlePass2DirectiveStep(ObjectCode objCode, DirectiveToken
token, Symbol repSymbol,

            SymbolTable symbolTable, LiteralTable literalTable) throws RuntimeException {
                // 그냥 귀찮아서 심볼테이블도 가져왔음
                // 어차피 SECT마다 독립적으로 심볼테이블 가짐
                Directive directive = token.getDirective();
                String sectionName = repSymbol.getName();

                ArrayList<Operand> operands;
                NumericOperand numOperand;
                Numeric num;
```

```
                int address;
                int size;

                switch (directive) {
                        case START:
                                numOperand = (NumericOperand)
token.getOperands().get(0);
                                objCode.setSectionName(sectionName);

        objCode.setStartAddress(numOperand.getNumeric().getInteger());
                                break;

                        case CSECT:
                                objCode.setSectionName(sectionName);
                                objCode.setStartAddress(0);
                                break;

                        case EXTDEF:
                                // TODO: EXTDEF인 경우에 대해서 pass2 처리하기.
                                // objCode.addDefineSymbol(?, ?);
                                operands = token.getOperands();
                                for (Operand operand : operands) {
                                        NumericOperand op = (NumericOperand)
operand;

        objCode.addDefineSymbol(op.getNumeric().getName().orElseThrow(
```

```
                                                        () -> new
RuntimeException("Expected numeric operand with name for EXTDEF")),

            symbolTable.search(op.getNumeric().getName().get()).get().getAddress().get().getIn
teger());
                                } // 피연산자에 대해 EXTDEF 심볼들을 오브젝트 인스턴
스 D 에 넘기기
                                break;

                        case EXTREF:
                                operands = token.getOperands();
                                for (Operand operand : operands) {
                                        numOperand = (NumericOperand) operand;
                                        num = numOperand.getNumeric();
                                        String symbolName = num.getName().get();
                                        objCode.addReferSymbol(symbolName);
                                } // EXTREF 심볼들을 오브젝트 인스턴스 R 에 넘기기
                                break;

                        case BYTE:
                                // TODO: BYTE인 경우에 대해서 pass2 처리하기.
                                // objCode.addText(?, ?, ?);
                                numOperand = (NumericOperand)
token.getOperands().get(0);
                                objCode.addText(token.getAddress(),
numOperand.getNumeric(), numOperand.getNumeric().getSize());
                                // BYTE 의 정보 오브젝트 인스턴스에 T 에 넘기기
```

```
                    break;


                case WORD:
                    // TODO: WORD인 경우에 대해서 pass2 처리하기.
                    ArrayList<String> refers = new ArrayList<>();
                    for (int i = 0;  i < token.getOperands().size(); i++) {
                        NumericOperand numericOperand =
(NumericOperand) token.getOperands().get(i);
                        for (int j = 0 ; j <
numericOperand.getNumeric().relativeMap_numbers() ; j++) {
                            String refer =
numericOperand.getNumeric().getReferSymbolsString().get(j).toString();
                            refers.add(refer);
                            objCode.addModification(refer,
token.getAddress(), token.getSize() * 2);
                        } // WORD에 대해 수식이 등장한다면...
modification 정보 오브젝트 인스턴스에 M 으로 넘기기
                    }
                    numOperand = (NumericOperand)
token.getOperands().get(0);
                    objCode.addText(token.getAddress(),
numOperand.getNumeric(), 3); // WORD의 사이즈를 3을 가정 - 만약 크기 달라진다면...?


                    break;


                case LTORG:
                    // TODO: LTORG인 경우에 대해서 pass2 처리하기.
```

```
                    // objCode.addText(?, ?, ?);
                    for (Literal literal : literalTable.getAllLiterals()) {
                        if (literal.getAddress().isPresent()
&& !literal.Is_Written()) {
                            int litAddress = literal.getAddress().get();
                            Numeric litValue = literal.getValue();
                            int litSize = literal.getSize();

                            objCode.addText(litAddress, litValue,
litSize);
                            literal.set_written_flag();
                        } // 리터럴 중 아직 쓰이지 않은것을 T에 쓰고
written flag를 true로 설정
                          // true인 경우 - T에 쓰지 않음
                    }
                    break;

                case END:
                    // TODO: END인 경우에 대해서 pass2 처리하기.
                    for (Literal literal : literalTable.getAllLiterals()) { // LTORG
못한 리터럴 처리
                        if (literal.getAddress().isPresent()
&& !literal.Is_Written()) {
                            int litAddress = literal.getAddress().get();
                            Numeric litValue = literal.getValue();
                            int litSize = literal.getSize();
```

```
                                        objCode.addText(litAddress, litValue,
litSize);
                                        literal.set_written_flag();
                                    } // 모든 코드의 끝 부분에서 LTORG와 동일작업
수행
                                }
                                if (!token.getOperands().isEmpty()) {
                                    NumericOperand END_operand_numeric =
(NumericOperand) token.getOperands().get(0);
                                    String END_operand =
END_operand_numeric.getNumeric().getName().get();

        objCode.setInitialPC(symbolTable.search(END_operand).get().getAddress().get().get
Integer());
                                } // 만약 END Directive의 피연산자가 PASS1에서 심볼테
이블에 존재했을 경우...
                                  // 현재 섹션에 END의 피연산자 심볼이 존재해서 토큰
피연산자로 넘겨받았음
                                  // 따라서 오브젝트 인스턴스 E에 붙여질 InitialPC 를
해당 심볼의 주소로 설정
                                objCode.setProgramLength(token.getAddress() +
token.getSize());
                                // 섹션의 총 사이즈 설정 크기를 계산

                                break;

                            case RESB:
                                objCode.add_clear_Text();
```

```
                        case RESW:
                                objCode.add_clear_Text();
                                // 줄바꿈에 해당하는 더미 Text 배열 생성
                        case EQU:
                                // 처리할 동작이 없음.
                                break;

                        default:
                                throw new UnsupportedOperationException("not fully
support Directive");
                }
        }

        private final List<Token> _tokens;
        private final SymbolTable _symbolTable;
        private final LiteralTable _literalTable;
}
```

ObjectCode.java

```
/**
 * @author Enoch Jung (github.com/enochjung)
 * @file ObjectCode.java
 * @date 2024-05-05
 * @version 1.0.0
 *
 * @brief 조교가 구현한 SIC/XE 어셈블러 코드 구조 샘플
 */

import java.util.Optional;
import java.util.ArrayList;

import numeric.Numeric;

public class ObjectCode {
	public ObjectCode() {
		_sectionName = Optional.empty();
		_startAddress = Optional.empty();
		_programLength = Optional.empty();
		_initialPC = Optional.empty();

		_defines = new ArrayList<Define>();
		_refers = new ArrayList<String>();
		_texts = new ArrayList<Text>();
```

```
            _mods = new ArrayList<Modification>();
        }

        /**
         * ObjectCode 객체를 String으로 변환한다. Assembler.java에서 오브젝트 코드를 출력하는 데에 사용된다.
         */
        @Override
        public String toString() {
            if (_sectionName.isEmpty() || _startAddress.isEmpty() || _programLength.isEmpty())
                throw new RuntimeException("illegal operation");

            String sectionName = _sectionName.get();
            int startAddress = _startAddress.get();
            int programLength = _programLength.get();

            String header = String.format("H%-6s%06X%06X₩n", sectionName, startAddress, programLength);

            // TODO: 오브젝트 코드 문자열 생성하기.
            String define = "D";
            // 이어붙이기
            define += _defines.stream().map(d -> String.format("%-6s%06X", d.symbolName, d.address))
                    .reduce("", (acc, x) -> acc + x);
```

```
define += "₩n";
// 마지막에 줄바꿈 붙이기
if (_defines.isEmpty()) define = "";

String refer = "R";
refer += _refers.stream().map(r -> String.format("%-6s", r))
                        .reduce("", (acc, x) -> acc + x);
refer += "₩n";
if (_refers.isEmpty()) refer = "";

// Text는 최대 70B -> 앞에 9 빼고 30B -> 총 60글자로...
// 첫 녀석은 "T%06X%02X%s", t.address, t.value.length() / 2, t.value 다
넣기
// 그 뒤로 오는 녀석들은 t.value만 넣어가면서 크기 계산
// 만약 크기가 꽉 찼으면 줄바꾸기
// 첫 녀석은 "T%06X%02X%s", t.address, t.value.length() / 2, t.value 다
넣기
// 이런식으로 진행...
final int MAX_CHARS = 60; // 최대 글자 수 (30바이트)
String text = "";
int currentLineStartAddress = 0;
String currentLineValues = "";
int currentLength = 0;

for (Text t : _texts) {
        if (!currentLineValues.isEmpty() && t.line_clear) {
```

```
                text += String.format("T%06X%02X%s₩n",
                                currentLineStartAddress,
                                currentLength / 2,
                                currentLineValues);
                currentLineValues = ""; // 내용 초기화
                currentLength = 0;
                continue;
        }

        if (currentLineValues.isEmpty()) {
                // 새로운 라인을 시작할 때 시작 주소 설정
                currentLineStartAddress = t.address;
        }

        // 현재 값 추가 전에 길이를 확인
        if (currentLength + t.value.length() > MAX_CHARS) {
                // 현재 줄을 종료하고 새로운 줄을 시작
                text += String.format("T%06X%02X%s₩n",
                                currentLineStartAddress,
                                currentLength / 2,
                                currentLineValues);
                currentLineValues = ""; // 내용 초기화
                currentLineStartAddress = t.address; // 새로운 시작 주소
업데이트
                currentLength = 0;
```

```
                }

                // 현재 줄에 값을 추가
                currentLineValues += t.value;
                currentLength += t.value.length();
            }

            // 마지막 라인 처리
            if (!currentLineValues.isEmpty()) {
                text += String.format("T%06X%02X%s₩n",
                                currentLineStartAddress,
                                currentLength / 2,
                                currentLineValues);
            }

            String modification = _mods.stream()
                                .map(m -> String.format("M%06X%02X%s", m.address,
m.sizeHalfByte, m.symbolNameWithSign))
                                .reduce("", (acc, x) -> acc + x + "₩n");

            String end = "E" + _initialPC
                                .map(x -> String.format("%06X", x))
                                .orElse("");

            return header + define + refer + text + modification + end;
```

```
}

public void setSectionName(String sectionName) {
        _sectionName = Optional.of(sectionName);
}

public void setStartAddress(int address) {
        _startAddress = Optional.of(address);
}

public void setProgramLength(int length) {
        _programLength = Optional.of(length);
}

public void addDefineSymbol(String symbolName, int address) {
        _defines.add(new Define(symbolName, address));
}

public void addReferSymbol(String symbolName) {
        _refers.add(symbolName);
}

public void addText(int address, Numeric context, int size) {
        String value = context.getValue(size * 2);
        _texts.add(new Text(address, value));
```

```
	}

	public void add_clear_Text() {
		Text clear_Text = new Text(0, "");
		clear_Text.setLine_clear();
		_texts.add(clear_Text);
	}

	public void addModification(String symbolNameWithSign, int address, int sizeHalfByte) {
		_mods.add(new Modification(address, sizeHalfByte, symbolNameWithSign));
	}

	public void setInitialPC(int address) {
		_initialPC = Optional.of(address);
	}

	class Define {
		Define(String symbolName, int address) {
			this.symbolName = symbolName;
			this.address = address;
		}

		String symbolName;
		int address;
```

```
	}

	class Text {
		Text(int address, String value) {
			this.address = address;
			this.value = value;
			this.line_clear = false;
		}
		void setLine_clear (){ // 개행 필요하면 true 설정
			line_clear = true;
		}

		int address;
		String value;
		boolean line_clear;
	}

	class Modification {
		Modification(int address, int sizeHalfByte, String symbolNameWithSign) {
			this.address = address; // 위치
			this.sizeHalfByte = sizeHalfByte; // 변경할 개수
			this.symbolNameWithSign = symbolNameWithSign; // ex)
+RDREC -BUFFER
		}
```

```
            int address;
            int sizeHalfByte;
            String symbolNameWithSign;
        }

        private Optional<String> _sectionName;
        private Optional<Integer> _startAddress;
        private Optional<Integer> _programLength;
        private Optional<Integer> _initialPC;

        private ArrayList<Define> _defines;
        private ArrayList<String> _refers;
        private ArrayList<Text> _texts; // Text Array
        private ArrayList<Modification> _mods;
}
```

Literal.java

```
/**
 * @author Enoch Jung (github.com/enochjung)
 * @file Literal.java
 * @date 2024-05-05
 * @version 1.0.0
 *
 * @brief 조교가 구현한 SIC/XE 어셈블러 코드 구조 샘플
 */

package literal;

import java.util.Optional;
import numeric.Numeric;

public class Literal {
	/**
	 * 리터럴 객체를 초기화한다. 주소는 할당하지 않는다.
	 *
	 * @param literal 리터럴 문자열
	 * @throws RuntimeException 잘못된 리터럴 문자열 포맷
	 */
	Literal(String literal) throws RuntimeException {
		// TODO: 리터럴 객체 초기화하기.
```

```
            _literal = literal;
            _address = Optional.empty();
            _value = new Numeric(literal.substring(1));
            written_flag = false;
        }

        /**
         * 리터럴 String을 반환한다.
         *
         * @return 리터럴 String
         */
        public String getLiteral() {
            return _literal;
        }

        public Numeric getValue() {
            return _value;
        }

        /**
         * 리터럴의 주소를 반환한다.
         *
         * @return 리터럴의 주소. 주소가 지정되지 않은 경우 empty
<code>Optional</code>
         */
```

```
	public Optional<Integer> getAddress() {
		return _address;
	}

	/**
	 * 리터럴의 수치값을 저장하기 위해 필요한 크기를 반환한다.
	 *
	 * @return 수치값 크기
	 */
	public int getSize() {
		return _value.getSize();
	}

	/**
	 * 리터럴 객체의 정보를 문자열로 반환한다. 리터럴 테이블 출력 용도로 사용한다.
	 */
	@Override
	public String toString() {
		String literal = _literal;
		String address = _address
				.map(x -> String.format("%X", x))
				.map(x -> x.replaceAll("[+].*$", ""))
				.orElse("(not assigned)");
		String formatted = String.format("%-12s%s", literal, address);
```

```
            return formatted;
        }

        /**
         * 리터럴의 주소를 지정한다.
         *
         * @param address 리터럴의 주소
         */
        void assignAddress(int address) {
            _address = Optional.of(address);
        }
        public void set_written_flag() { written_flag = true; }
        public boolean Is_Written() { return written_flag; }

        private final String _literal;

        /** 리터럴 주소. 주소가 지정되지 않은 경우 empty <code>Optional</code> */
        private Optional<Integer> _address;
        private final Numeric _value;
        private boolean written_flag;
        // 리터럴을 PASS2에서 다룰때, 이 플래그를 통해 중복 처리 되지 않도록 한다.
}
```

LiteralTable.java

```
/**
 * @author Enoch Jung (github.com/enochjung)
 * @file LiteralTable.java
 * @date 2024-05-05
 * @version 1.0.0
 *
 * @brief 조교가 구현한 SIC/XE 어셈블러 코드 구조 샘플
 */

package literal;

import java.util.LinkedHashMap;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.Collection;

public class LiteralTable {
	/**
	 * 리터럴 테이블을 초기화한다.
	 */
	public LiteralTable() {
		_literalMap = new LinkedHashMap<String, Literal>();
	}
```

```
	/**
	 * 리터럴을 리터럴 테이블에 추가한다.
	 *
	 * @param literal 추가할 리터럴
	 * @throws RuntimeException 비정상적인 리터럴 서식 혹은 이미 존재하는 리터럴 추가를 시도
	 */
	public Literal putLiteral(String literal) throws RuntimeException {
		// TODO: 리터럴 객체를 생성하고, 이를 리터럴 테이블에 추가하기.
		if (_literalMap.containsKey(literal)) {
			// 리터럴은 값이 할당이 되지 않았어도 여러번 보게 됨...
			// throw new RuntimeException("Literal already exists: " + literal);
		}
		Literal newLiteral = new Literal(literal);
		_literalMap.put(literal, newLiteral);
		return newLiteral; // 새로운 리터럴 추가
	}

	/**
	 * 리터럴 문자열을 통해 리터럴을 찾는다.
	 *
	 * @param literal 찾을 리터럴 문자열
	 * @return 리터럴. 없을 경우 empty <code>Optional</code>
	 */
```

```
public Optional<Literal> search(String literal) {
        // TODO: 리터럴을 검색하고, 결과를 반환하기
        return Optional.ofNullable(_literalMap.get(literal));
} // 리터럴 String으로 검색 후 반환

/**
 * 리터럴 주소값을 통해 리터럴을 찾는다.
 *
 * @param address 찾을 리터럴의 시작 주소
 * @return 리터럴. 없을 경우 empty <code>Optional</code>
 */
public Optional<Literal> search(int address) {
        // TODO: 리터럴 주소값으로 리터럴을 검색하고, 결과를 반환하기.
        return _literalMap.values().stream()
                        .filter(literal -> literal.getAddress().orElse(-1) == address)
                        .findFirst();
} // 리터럴을 주소값으로 찾기 - 그러나 구현하면서 쓸일은 없었던거같음

/**
 * 리터럴 테이블에서 주소가 할당되지 않은 리터럴에 대해 주소를 할당하고, 해당 리터럴들의 전체 크기를 반환한다.
 *
 * @param address 할당을 시작할 주소
 * @return 할당된 리터럴들의 총 크기
 */
```

```
	public int assignAddress(int address) {
		// TODO: 리터럴 주소값 할당하기.
		// 매우 중요!!! 리터럴을 전부 할당 후에... 크기 계산해서 호출부에 알려야함
		// 그래야 정상적 LC 할당 작업 수행 가능
		int currentAddress = address;
		for (Literal lit : _literalMap.values()) {
			if (!lit.getAddress().isPresent()) {
				lit.assignAddress(currentAddress);
				currentAddress += lit.getSize();
			}
		}
		return currentAddress - address;
	}

	/**
	 * 리터럴 테이블 객체의 정보를 문자열로 반환한다. 리터럴 테이블 출력 용도로 사용한다.
	 */
	@Override
	public String toString() {
		String literals = _literalMap.entrySet().stream()
				.map(x -> x.getValue().toString())
				.collect(Collectors.joining("₩n"));

		return literals;
```

```
        }
        public Collection<Literal> getAllLiterals() {
                return _literalMap.values();
        }
        // 모든 리터럴을 하나씩 확인하는 함수 정의
        private LinkedHashMap<String, Literal> _literalMap;
}
```

Symbol.java

```
/**
 * @author Enoch Jung (github.com/enochjung)
 * @file Symbol.java
 * @date 2024-05-05
 * @version 1.0.0
 *
 * @brief 조교가 구현한 SIC/XE 어셈블러 코드 구조 샘플
 */

package symbol;

import java.util.Optional;

import numeric.Numeric;

public class Symbol {
	/**
	 * 문자열이 심볼 문자열 형태인지 판별한다.
	 *
	 * @param symbol 판별할 문자열
	 * @return 심볼 문자열로 사용 가능한 형태인지 여부
	 */
	public static boolean isSymbol(String symbol) {
		String symbolRegex = "^[a-zA-Z][a-zA-Z0-9]*$";
```

```
        return symbol.matches(symbolRegex) && symbol.length() <= 6;
    } // 심볼문자열로 사용 가능하다면 true ex) Xyz123 - true / 123abc false

    /**
     * 심볼 명칭을 반환한다.
     *
     * @return 심볼 명칭
     */
    public String getName() {
        return _name;
    }

    /**
     * 상대 주소로 표현될 수 없는 심볼인지 (START,CSECT,EXTREF로 생성된 심볼인지) 판별한다.
     *
     * @return 상대 주소로 표현될 수 없는지 여부
     */
    public boolean isBaseSymbol() {
        return _state == State.REP_SECTION || _state == State.EXTERNAL;
    }

    /**
     * EXTREF로 생성된 심볼인지 판별한다.
     *
```

```
     * @return EXTREF로 생성된 심볼인지 여부
     */
    public boolean isReferSymbol() {
        return _state == State.EXTERNAL;
    }

    /**
     * 심볼의 정보를 문자열로 반환한다. 디버그 용도로 사용한다.
     */
    @Override
    public String toString() {
        String name = _name;
        String address = _address
                .map(x -> x.toString())
                .map(x -> x.replace("+" + name, ""))
                .map(x -> x.replace("+", " ₩t+ "))
                .orElse("(not assigned)");
        String section;
        if (isBaseSymbol() || isReferSymbol() || _section == null) {
            section = ""; // 섹션 대표심볼, 상대주소 아닌 것들은 ""
        } else {
            section = _section;
        }
        String formatted = String.format("%-12s%-12s%s", name, _state ==
State.EXTERNAL ? "REF" : address,
```

```
                                section); // ex) COPY
                return formatted;
        }

        public Optional<Numeric> getAddress() {
                return _address;
        }

        /**
         * 대표 심볼 객체를 초기화한다.
         *
         * @param name      심볼 명칭
         * @param address 절대 주소
         * @return 대표 심볼 객체
         * @throws RuntimeException 부적절한 심볼 명칭
         */
        static Symbol createRepSymbol(String name, int address) throws
RuntimeException {
                // TODO: symbol 객체의 address 할당하기.
                if (!isSymbol(name)) {
                        throw new RuntimeException("Illegal symbol name: " + name);
                } // 대표심볼 할당을 진행한다. State REP_SECTION
                Numeric numeric = new Numeric(String.valueOf(address));
                Optional<Numeric> optionalNumeric = Optional.of(numeric);
                Symbol symbol = new Symbol(name, optionalNumeric,
State.REP_SECTION);
```

```
        return symbol;
}

/**
 * 외부 심볼 객체를 초기화한다.
 *
 * @param name 심볼 명칭
 * @return 외부 심볼 객체
 * @throws RuntimeException 부적절한 심볼 명칭
 */
static Symbol createExternalSymbol(String name) throws RuntimeException {
        // TODO: 외부 심볼 객체 생성하기.
        // Symbol symbol = new Symbol(?, ?, ?);
        // and something more...?
        if (!isSymbol(name)) {
                throw new RuntimeException("Illegal symbol name: " + name);
        } // state EXTERNAL
        Optional<Numeric> optionalNumeric = Optional.empty();
        Symbol symbol = new Symbol(name, optionalNumeric, State.EXTERNAL);
        return symbol;
}

/**
 * 주소값이 주어진 일반 심볼 객체를 초기화한다.
 *
```

```
     * @param name      심볼 명칭
     * @param address 주소값
     * @return 일반 심볼 객체
     * @throws RuntimeException 부적절한 심볼 명칭
     */
    static Symbol createAddressAssignedSymbol(String name, Numeric address)
throws RuntimeException {
        if (!isSymbol(name)) {
            throw new RuntimeException("Illegal symbol name: " + name);
        }
        Symbol symbol = new Symbol(name, Optional.of(address),
State.ADDRESS_ASSIGNED);
        return symbol;
    } // 심볼을 새로 인식할때, 레이블에 나온 심볼은 locctr통해 주소계산 바로 가능

    /**
     * 주소값이 주어지지 않은 일반 심볼 객체를 초기화한다.
     *
     * @param name 심볼 명칭
     * @return 일반 심볼 객체
     * @throws RuntimeException 부적절한 심볼 명칭
     */
    static Symbol createAddressNotAssignedSymbol(String name) throws
RuntimeException {
        if (!isSymbol(name)) {
            throw new RuntimeException("Illegal symbol name: " + name);
```

```
                }
                Symbol symbol = new Symbol(name, Optional.empty(),
State.ADDRESS_NOT_ASSIGNED);
                return symbol;
        }

        /**
         * 주소값이 주어지지 않은 일반 심볼 객체의 주소를 설정한다.
         *
         * @param address 주소값
         * @throws RuntimeException 주소값이 주어지지 않은 일반 심볼이 아님
         */
        void assign(Numeric address) throws RuntimeException {
                // TODO: 주소값 설정하기.
                if (_state != State.ADDRESS_NOT_ASSIGNED) {
                        throw new RuntimeException("Address can only be assigned to
symbols with not assigned state.");
                } // 주소가 설정 안되어있던 심볼에 대해 주소 할당
                _address = Optional.of(address);
                _state = State.ADDRESS_ASSIGNED;
        }
        void assign_section(String _repsect) throws RuntimeException {
                _section = _repsect;
        } // 상대주소를 명시적으로 넣는 함수 ex) COPY

        public boolean state_ADDRESS_NOT_ASSIGNED() {
```

```
            if (_state == State.ADDRESS_NOT_ASSIGNED) return   true;
            else return false;
        } // 예외처리 위한 > 주소 할당 안되어있다면 true

        private Symbol(String name, Optional<Numeric> address, State state) throws
RuntimeException {
            if (!isSymbol(name))
                throw new RuntimeException("illegal symbol name");
            _name = name;
            _address = address;
            _state = state;
        }

        /**
         * 심볼의 상태값
         *
         * <ul>
         * <li><code>State.REP_SECTION</code>: control section의 명칭으로 선언한 대
표 심볼
         * <li><code>State.EXTERNAL</code>: EXTREF로 선언한 외부 심볼
         * <li><code>State.ADDRESS_ASSIGNED</code>: label에서 등장하여 주소값이
결정된 일반 심볼
         * <li><code>State.ADDRESS_NOT_ASSIGNED</code>: operand에서 등장하였으
나, 아직 label에서는
         * 등장하지 않아 주소값이 결정되지 않은 일반 심볼
         * </ul>
```

```
 */
private enum State {
        /**
         * control section의 명칭으로 선언한 대표 심볼
         */
        REP_SECTION,

        /**
         * EXTREF로 선언한 외부 심볼
         */
        EXTERNAL,

        /**
         * label에서 등장하여 주소값이 결정된 일반 심볼
         */
        ADDRESS_ASSIGNED,

        /**
         * operand에서 등장하였으나, 아직 label에서는 등장하지 않아 주소값이 결정되지 않은 일반 심볼
         */
        ADDRESS_NOT_ASSIGNED;
}

private final String _name;
```

```
        private Optional<Numeric> _address;
        private State _state;
        private String _section;
        // 상대주소 명시 String 추가
}
```

SymbolTable.java

```
/**
 * @author Enoch Jung (github.com/enochjung)
 * @file SymbolTable.java
 * @date 2024-05-05
 * @version 1.0.0
 *
 * @brief 조교가 구현한 SIC/XE 어셈블러 코드 구조 샘플
 */

package symbol;

import java.util.LinkedHashMap;
import java.util.Optional;
import java.util.stream.Collectors;

import numeric.Numeric;

public class SymbolTable {
	/**
	 * 심볼 테이블 객체를 초기화한다.
	 */
	public SymbolTable() {
		_symbolMap = new LinkedHashMap<String, Symbol>();
		_repSymbol = Optional.empty();
```

```
}

/**
 * 주소값이 정해지지 않은 심볼을 추가한다.
 *
 * @param name 심볼 명칭
 * @return 심볼 객체
 * @throws RuntimeException 잘못된 심볼 생성 시도
 */
public Symbol put(String name) throws RuntimeException {
        // TODO: 예외 처리하기 (exception)
        // 만약 이미 심볼이 존재한다면?
        if (_symbolMap.containsKey(name)) {
                // throw new RuntimeException("Symbol already exist: " + name);
        }
        Symbol symbol = Symbol.createAddressNotAssignedSymbol(name);
        _symbolMap.put(name, symbol);
        return symbol;
}

/**
 * 주소값이 정해진 심볼을 추가한다.
 *
 * @param name      심볼 명칭
 * @param address 심볼 주소
```

```
     * @return 심볼 객체
     * @throws RuntimeException 잘못된 심볼 생성 시도
     */
    public Symbol put(String name, int address) throws RuntimeException {
        // TODO: 심볼 추가하기. 만약 심볼이 이미 존재하고 해당 심볼이 주소가 지정되지 않은 심볼일 경우, 주소값 할당하기.

        Symbol symbol;
        Optional<Symbol> optSymbol = search(name);
        if (optSymbol.isPresent()) {
            // TODO: 해당 심볼이 주소가 지정되지 않은 심볼일 경우 주소값 할당하기.
            symbol = optSymbol.get();
            if (!symbol.getAddress().isPresent()) { // address가 assign 안 되어 있으면
                Numeric numeric = new Numeric(String.valueOf(address));
                symbol.assign(numeric); // Assign
                symbol.assign_section(_repSymbol.get().getName());
            } else {
                throw new RuntimeException("Symbol address already assigned: " + name);
            }
        } else {
            // TODO: 심볼 추가하기.
            Numeric numeric = new Numeric(String.valueOf(address));
            symbol = Symbol.createAddressAssignedSymbol(name, numeric);
```

```
                symbol.assign_section(_repSymbol.get().getName());
                _symbolMap.put(name, symbol);
            }

            return symbol;
        }

        /**
         * EQU label에 해당하는 심볼을 추가한다.
         *
         * @param name      심볼 명칭
         * @param formula 수식 문자열
         * @param locctr   location counter 값
         * @return 심볼 객체
         * @throws RuntimeException 잘못된 심볼 생성 시도 혹은 잘못된 수식 포맷
         */
        public Symbol put(String name, String formula, int locctr) throws
RuntimeException {
            // TODO: 심볼 추가하기. 만약 심볼이 이미 존재하고 해당 심볼이 주소
가 지정되지 않은 심볼일 경우, 주소값 할당하기.

            Symbol symbol;
            Numeric addr = new Numeric(formula, this, locctr);
            Optional<Symbol> optSymbol = search(name);

            if (optSymbol.isPresent()) {
```

```
                    // TODO: 해당 심볼이 주소가 지정되지 않은 심볼일 경우 주소
값 할당하기.
                    symbol = optSymbol.get();
                    if (!symbol.getAddress().isPresent()) { // address가 assign 안 되어
있으면
                        symbol.assign(addr); // Assign
                        if (formula.equals("*")) {

    symbol.assign_section(_repSymbol.get().getName());
                        } // 지금은 하드코딩으로 *만 상대주소 할당했지만... 수
정이 좀 필요함
                    }
                } else {
                    // TODO: 심볼 추가하기.
                    symbol = Symbol.createAddressAssignedSymbol(name, addr);
                    if (formula.equals("*")) {
                        symbol.assign_section(_repSymbol.get().getName());
                    } // 지금은 하드코딩으로 *만 상대주소 할당했지만... 수정이 좀
필요함
                    _symbolMap.put(name, symbol);
                }

                return symbol;
            }

            /**
             * control section 명칭에 해당하는 심볼을 추가한다.
```

```
     *
     * @param name      심볼 명칭
     * @param address 심볼 주소
     * @return 심볼 객체
     * @throws RuntimeException 잘못된 심볼 생성 시도
     */
    public Symbol putRep(String name, int address) throws RuntimeException {
        // TODO: control section 명칭에 해당하는 심볼을 추가하기.
        // with Symbol.createRepSymbol(?, ?);

        if (_repSymbol.isPresent()) {
            throw new RuntimeException("Representative symbol is already
defined.");
        }

        Symbol symbol = Symbol.createRepSymbol(name, address);
        _repSymbol = Optional.of(symbol);
        _symbolMap.put(name, symbol);
        return symbol;
    }

    /**
     * EXTERN operand에 주어지는 외부 심볼을 추가한다.
     *
     * @param name 심볼 명칭
```

```
     * @return 심볼 객체
     * @throws RuntimeException 잘못된 심볼 생성 시도
     */
    public Symbol putRefer(String name) throws RuntimeException {
            if (_symbolMap.containsKey(name)) {
                    throw new RuntimeException("Symbol already exists: " + name);
            }
            Symbol symbol = Symbol.createExternalSymbol(name);
            _symbolMap.put(name, symbol);

            // TODO: EXTERN operand에 주어지는 외부 심볼을 추가하기.

            return symbol;
    }

    /**
     * 심볼 테이블에서 심볼을 찾는다.
     *
     * @param name 찾을 심볼 명칭
     * @return 심볼. 없을 경우 empty <code>Optional</code>
     */
    public Optional<Symbol> search(String name) {
            return Optional.ofNullable(_symbolMap.get(name));
    }
```

```
	/**
	 * control section 명칭에 해당하는 심볼을 반환한다.
	 *
	 * @return 심볼. 없을 경우 empty <code>Optional</code>
	 */
	public Optional<Symbol> getRepSymbol() {
		return _repSymbol;
	}

	/**
	 * 심볼 테이블 객체의 정보를 문자열로 반환한다. 심볼 테이블 출력 용도로 사용한다.
	 */
	@Override
	public String toString() {
		String symbols = _symbolMap.entrySet().stream()
				.map(x -> x.getValue().toString())
				.collect(Collectors.joining("₩n"));

		return symbols;
	}

	private final LinkedHashMap<String, Symbol> _symbolMap;
	private Optional<Symbol> _repSymbol;
}
```

StringToken.java

```
/**
 * @author Enoch Jung (github.com/enochjung)
 * @file StringToken.java
 * @date 2024-05-05
 * @version 1.0.0
 *
 * @brief 조교가 구현한 SIC/XE 어셈블러 코드 구조 샘플
 */

package token;

import java.util.ArrayList;
import java.util.Optional;
import java.util.stream.Collectors;

import java.util.Arrays;
import instruction.*;

public class StringToken {
	/**
	 * 소스 코드 한 줄에 해당하는 토큰을 초기화한다.
	 *
	 * @param input 소스 코드 한 줄에 해당하는 문자열
	 * @throws RuntimeException 잘못된 형식의 소스 코드 파싱 시도.
```

```
 */
public StringToken(String input) throws RuntimeException {
        // TODO: 소스 코드를 파싱하여 토큰을 초기화하기.

        /*
         * 1. input 의 첫 문자가 .이라면 주석으로 처리
         * 2. 첫 문자가 ₩t 가 아니라면 레이블로 설정
         * 3. 첫 문자가 ₩t 이라면 오퍼레이터부터 설정
         *
         * 4. 오퍼레이터 설정이 끝나면 피연산자를 체크하는데
         *      여기서 ₩tOPERAND₩tCOMMENT 의 경우와
         *            ₩t₩tCOMMENT 의 경우와
         *            아무것도 없는 경우가 존재
         *       각각에 대해 제대로 파싱 진행해야함
         */

        _label = Optional.empty();
        _operator = Optional.empty();
        _operands = new ArrayList<>();
        _comment = Optional.empty();

        _nBit = true;
        _iBit = true;
        _xBit = false;
        _pBit = true;
```

```
_eBit = false;

// 1. .으로 시작하면 주석 처리
if (input.startsWith(".")) {
        _comment = Optional.of(input.substring(1).trim());
        return;
}

int pos = 0; // 현재 위치 0 에서 시작
int length = input.length(); // 입력 문자열의 길이

// 2. 레이블 추출
if (Character.isLetter(input.charAt(pos))) { // 첫 문자가 문자인 경우
        int start = pos;
        while (pos < length && input.charAt(pos) != '₩t') pos++;
        _label = Optional.of(input.substring(start, pos));
        pos++; // 탭을 넘어서 연산자를 가리키도록 함
}
else {
        pos++; // 첫 문자가 ₩t 인 경우 건너 뜀
}

// 3. 연산자 추출
if (pos < length) {
        int start = pos;
```

```
                while (pos < length && input.charAt(pos) != '₩t') pos++;
                String operator = input.substring(start, pos);
                if (operator.startsWith("+")) {
                        _eBit = true;
                        _pBit = false;
                        operator = operator.substring(1);
                }
                _operator = Optional.of(operator);
                pos++;
        }

        // 4. 피연산자 추출
        if (pos < length && input.charAt(pos) != '₩t') { // 다음 문자가 탭이 아니라면 피연산자가 존재
                int start = pos;
                while (pos < length && input.charAt(pos) != '₩t') pos++;
                String operands = input.substring(start, pos);
                // _operands.addAll(Arrays.asList(operands.split(",")));

                String[] operandArray = operands.split(",");
                ArrayList<String> processedOperands = new ArrayList<>();

                // 피연산자 처리 필요함
                // 만약 첫 피연산자의 첫 문자가 @이라면 _nBit true, _iBit false
                // 만약 첫 피연산자의 첫 문자가 #이라면 _nBit false, _iBit false
```

```
                    // 그리고 substring(1)


                    // 만약 필요한 피연산자 수보다 많은 경우 "X"인지 확인
                    // X 라면 _xBit true
                    // 그리고 해당 피연산자 삭제
                    // 이 작업을 위해서는 _operator 를 instTable 에서 search 해서 피연산자 형식을 봐야 할 듯


                    for (int i = 0; i < operandArray.length; i++) { // 하나씩 돌면서 확인
                        String operand = operandArray[i].trim(); // 혹시모르니 공백 제거 (BUFFER, X) 이런식으로 쓸수도?
                        if (i == 0) {
                            if (operand.charAt(0) == '@') {
                                _nBit = true;
                                _iBit = false;
                                operand = operand.substring(1); // '@' 제거
                            }
                            else if (operand.charAt(0) == '#') {
                                _nBit = false;
                                _iBit = true;
                                _pBit = false;
                                operand = operand.substring(1); // '#' 제거
                            }
                        }
```

```
                    processedOperands.add(operand);
                }

                // _xBit 검사 및 처리
                // (no label)      <CLEAR>      (no operand)X      <CLEAR LOOP
COUNTER>
                // 예시 처리는 그냥 X 나오면 _xBit true 로 처리? 아마?
                // 조건 1. Operand 존재 2. 마지막 Operand 가 X 3. Operand 는
2 개 이상 (CLEAR X 의 경우)
                // 그런데 COMR A,X 는???
                // 더 정확한 처리를 위해서는 instTable 에서 검색 필요...
                if (!processedOperands.isEmpty()
                        &&
processedOperands.get(processedOperands.size() - 1).equals("X")
                        && processedOperands.size() > 1) {
                    _xBit = true; // 인덱싱 비트 설정
                    processedOperands.remove(processedOperands.size() -
1); // 'X' 제거
                }

                _operands.addAll(processedOperands); // 처리된 피연산자들을
최종 목록에 추가
                pos++;
            }

            // 5. 주석 처리
```

```
        if (pos < length) { // 남은 문자열은 모두 주석으로 처리
                _comment = Optional.of(input.substring(pos).trim());
        }

        // System.out.println(this.toString()); /** 디버깅 용도 */
}

/**
 * label 문자열을 반환한다.
 *
 * @return label 문자열. 없으면 empty <code>Optional</code>.
 */
public Optional<String> getLabel() {
        return _label;
}

/**
 * operator 문자열을 반환한다.
 *
 * @return operator 문자열. 없으면 empty <code>Optional</code>.
 */
public Optional<String> getOperator() {
        return _operator;
}
```

```
/**
 * operand 문자열 배열을 반환한다.
 *
 * @return operand 문자열 배열
 */
public ArrayList<String> getOperands() {
	return _operands;
}

/**
 * comment 문자열을 반환한다.
 *
 * @return comment 문자열. 없으면 empty <code>Optional</code>.
 */
public Optional<String> getComment() {
	return _comment;
}

/**
 * 토큰의 iNdirect bit 가 1 인지 여부를 반환한다.
 *
 * @return N bit 가 1 인지 여부
 */
public boolean isN() {
	return _nBit;
```

```
}

/**
 * 토큰의 Immediate bit 가 1 인지 여부를 반환한다.
 *
 * @return I bit 가 1 인지 여부
 */
public boolean isI() {
	return _iBit;
}

/**
 * 토큰의 indeX bit 가 1 인지 여부를 반환한다.
 *
 * @return X bit 가 1 인지 여부
 */
public boolean isX() {
	return _xBit;
}

/**
 * 토큰의 Pc relative bit 가 1 인지 여부를 반환한다.
 *
 * @return P bit 가 1 인지 여부
 */
```

```
	public boolean isP() {
		return _pBit;
	}

	/**
	 * 토큰의 Extra bit 가 1 인지 여부를 반환한다.
	 *
	 * @return E bit 가 1 인지 여부
	 */
	public boolean isE() {
		return _eBit;
	}

	/**
	 * StringToken 객체의 정보를 문자열로 반환한다. 디버그 용도로 사용한다.
	 */
	@Override
	public String toString() {
		String label = _label.map(x -> "<" + x + ">").orElse("(no label)");
		String operator = (isE() ? "+" : "") + _operator.map(x -> "<" + x +
">").orElse("(no operator)");
		String operand = (isN() && !isI() ? "@" : "") + (isI() && !isN() ? "#" : "")
				+ (_operands.isEmpty() ? "(no operand)"
						: "<" +
_operands.stream().collect(Collectors.joining("/")) + ">")
				+ (isX() ? (_operands.isEmpty() ? "X" : "/X") : "");
```

```
                String comment = _comment.map(x -> "<" + x + ">").orElse("(no
comment)");

                String formatted = String.format("%-12s₩t%-12s₩t%-18s₩t%s", label,
operator, operand, comment);
                return formatted;
        }

        private Optional<String> _label;
        private Optional<String> _operator;
        private ArrayList<String> _operands;
        private Optional<String> _comment;

        private boolean _nBit;
        private boolean _iBit;
        private boolean _xBit;
        // private boolean _bBit; /** base relative 는 구현하지 않음 */
        private boolean _pBit;
        private boolean _eBit;
}
```

InstructionToken.java

```
/**
 * @author Enoch Jung (github.com/enochjung)
 * @file InstructionToken.java
 * @date 2024-05-05
 * @version 1.0.0
 *
 * @brief 조교가 구현한 SIC/XE 어셈블러 코드 구조 샘플
 */

package token;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Collectors;
import java.util.Optional;

import instruction.Instruction;
import literal.Literal;
import literal.LiteralTable;
import numeric.Numeric;
import symbol.SymbolTable;
import token.operand.*;

public class InstructionToken extends Token {
```

```
	public InstructionToken(String tokenString, int address, int size, Instruction inst,
ArrayList<Operand> operands,
			boolean nBit, boolean iBit, boolean xBit, boolean pBit, boolean
eBit) {
		super(tokenString, address, size);
		_inst = inst;
		_operands = operands;
		_nBit = nBit;
		_iBit = iBit;
		_xBit = xBit;
		_pBit = pBit;
		_eBit = eBit;
	}

	/**
	 * 토큰의 iNdirect bit 가 1 인지 여부를 반환한다.
	 *
	 * @return N bit 가 1 인지 여부
	 */
	public boolean isN() {
		return _nBit;
	}

	/**
	 * 토큰의 Immediate bit 가 1 인지 여부를 반환한다.
	 *
```

```
     * @return I bit 가 1 인지 여부
     */
    public boolean isI() {
            return _iBit;
    }

    /**
     * 토큰의 indeX bit 가 1 인지 여부를 반환한다.
     *
     * @return X bit 가 1 인지 여부
     */
    public boolean isX() {
            return _xBit;
    }

    /**
     * 토큰의 Base relative bit 가 1 인지 여부를 반환한다.
     *
     * @return B bit 가 1 인지 여부
     */
    public boolean isB() {
            return false;
    }

    /**
```

```
     * 토큰의 Pc relative bit 가 1 인지 여부를 반환한다.
     *
     * @return P bit 가 1 인지 여부
     */
    public boolean isP() {
            return _pBit;
    }

    /**
     * 토큰의 Extra bit 가 1 인지 여부를 반환한다.
     *
     * @return E bit 가 1 인지 여부
     */
    public boolean isE() {
            return _eBit;
    }

    /**
     * InstructionToken 객체의 정보를 문자열로 반환한다. 디버그 용도로 사용한다.
     */
    @Override
    public String toString() {
            String instName = _inst.getName();
            String operands = _operands.isEmpty() ? "(empty)"
                            : (_operands.stream()
```

```
                                        .map(x -> x.toString())
                                        .collect(Collectors.joining("/")));
                String nixbpe = String.format("0b%d%d%d%d%d%d", _nBit ? 1 : 0, _iBit ? 1 : 0, _xBit ? 1 : 0, 0, _pBit ? 1 : 0,
                                _eBit ? 1 : 0);
                return "InstructionToken{name:" + instName + ", operands:" + operands + ", nixbpe:" + nixbpe + "}";
        }

        /**
         * object code 에 관한 정보를 반환한다.
         *
         * @return 텍스트 레코드 정보가 담긴 객체
         * @throws RuntimeException 잘못된 심볼 객체 변환 시도.
         */
        public TextInfo getTextInfo(SymbolTable symTab, LiteralTable litTab) throws RuntimeException {
                int address = this.getAddress();
                Numeric code;
                int size = this.getSize();
                Optional<ModificationInfo> modInfo = Optional.empty();

                // TODO: pass2 과정 중, 오브젝트 코드 생성을 위한 정보를 TextInfo 객체에 담아서 반환하기.
                int opcode = _inst.getOpcode() & 0xFF;
                int operandCode = 0;
```

```
if (size == 2) { // 2 형식
    operandCode += (opcode << 8);
    if (_inst.getOperandType() == Instruction.OperandType.REG) {
        RegisterOperand registerOperand = (RegisterOperand)
_operands.get(0);
        operandCode += registerOperand.getValue() << 4;
    } else if (_inst.getOperandType() ==
Instruction.OperandType.REG1_REG2) {
        RegisterOperand registerOperand1 = (RegisterOperand)
_operands.get(0);
        RegisterOperand registerOperand2 = (RegisterOperand)
_operands.get(1);
        operandCode += registerOperand1.getValue() << 4;
        operandCode += registerOperand2.getValue();
    }
    // no operand-> 아무 행동 안한다
} else if (size == 3) { // 3 형식
    int PC = address + size;
    // PC 값 계산
    int displacement;
    operandCode += (opcode << 16);
    if (_nBit) { operandCode += 32 << 12; }
    if (_iBit) { operandCode += 16 << 12; }
    if (_xBit) { operandCode += 8 << 12; }
    if (_pBit) { operandCode += 2 << 12; }
```

```
                    if (_eBit) { operandCode += 1 << 12; }


                    if (_inst.getOperandType() ==
Instruction.OperandType.NO_OPERAND) {
                    } else if (_inst.getOperandType() ==
Instruction.OperandType.MEMORY) {
                        String opd0 = _operands.get(0).toString().substring(0,7);
                        switch (opd0) {
                            case "Numeric":
                                NumericOperand numericOperand =
(NumericOperand) _operands.get(0);
                                if
(numericOperand.getNumeric().isAbsolute()) {
                                    displacement =
numericOperand.getNumeric().getInteger();
                                    operandCode += displacement;
                                } else {
                                    String Target_symbol =
numericOperand.getNumeric().getReferSymbolsString().get(0).toString().substring(1);
                                    int Target_address =
symTab.search(Target_symbol).get().getAddress().get().getInteger();
                                    displacement = Target_address -
PC;
                                    operandCode += displacement;
                                    if (displacement < 0) { // !!! 만약
계산 결과가 음수 -> TA 가 PC 보다 작은 값이었다
                                        operandCode += 4096; //
만약 계산 했는데 음수이면 자릿수 하나 빌려온것이다
```

```
                                        } // 그래서 결과에 0x1000
추가해준다
                                    }
                                    break;
                                case "Literal":
                                    LiteralOperand literalOperand =
(LiteralOperand) _operands.get(0);
                                    displacement =
literalOperand.getAddress().get() - PC;
                                    operandCode += displacement;
                                    if (displacement < 0) { // !!! 만약 계산
결과가 음수 -> TA 가 PC 보다 작은 값이었다
                                        operandCode += 4096; // 만약
계산 했는데 음수이면 자릿수 하나 빌려온것이다
                                    } // 그래서 결과에 0x1000 추가해준다
                                    break;
                            }
                        }
                    } else if (size == 4) {
                        // 4 형식
                        int PC = address + size;
                        operandCode += (opcode << 24);
                        if (_nBit) { operandCode += 32 << 20; }
                        if (_iBit) { operandCode += 16 << 20; }
                        if (_xBit) { operandCode += 8 << 20; }
                        if (_pBit) { operandCode += 2 << 20; }
                        if (_eBit) { operandCode += 1 << 20; }
```

```
                        // modification 장보 추가
                        ArrayList<String> refers = new ArrayList<>();
                        for (int i = 0;   i < _operands.size(); i++) { // 예제에선 굳이
반복할 필요 없지만.. 그레도?
                                NumericOperand numericOperand = (NumericOperand)
_operands.get(i);
                                String refer =
numericOperand.getNumeric().getReferSymbolsString().get(0).toString();
                                refers.add(refer);
                                String check = refer.substring(1);
                                if
(symTab.search(check).get().state_ADDRESS_NOT_ASSIGNED()) {
                                        throw new RuntimeException("PASS2 Error :
missing symbol definition : " + check + "₩n");
                                } // 에러 처리 Ex) WAAAA
                        }
                        modInfo = Optional.of(new ModificationInfo(refers, address + 1,
5));
                } else {
                        throw new RuntimeException("wrong size instruction₩n");
                }
                code = new Numeric(Integer.toString(operandCode));

                TextInfo textInfo = new TextInfo(address, code, size, modInfo);

                return textInfo;
        }
```

```
	private Instruction _inst;

	private ArrayList<Operand> _operands;

	private boolean _nBit;
	private boolean _iBit;
	private boolean _xBit;
	// private boolean _bBit; /** base relative 는 구현하지 않음 */
	private boolean _pBit;
	private boolean _eBit;
}
```