시스템 프로그래밍 프로젝트 1b

20142385 엄재식

출석번호 19

1. 목적

- SIC/XE 소스를 Object Program Code 로 변환하여 봄으로써 SIC/XE 어셈블러의 동작을 이해한다.

- 주어진 자바파일을 이용하여 SIC/XE 소스를 Object Program Code로 변환하는 과정을 복습하고 C와 비교한다.

1. 설계/구현 아이디어
2. Parsing을 통해 instable 생성
3. input파일을 읽어 lineList 생성
4. pass 1

토큰리스트와 심볼리스트를 생성해 토큰테이블을 만든다.

* 토큰 테이블에 문자열 배열로 만든 ltorg리스트와 새로 만든 extref 클래스 배열로 만든 extref리스트를 추가함
* ltorg리스트는 토큰 operand 첫문자가 =인 경우 심볼리스트에 저장해 나중에 해당 주소를 찾기 쉽도록 하였고 해당 문자열을 ltorg리스트에 추가하고 나중에 object가 끝나는 operator인 end나 csect를 만났을 때 그리고 operator가 ltorg인 토큰을 만났을 때 ltorg리스트를 이용해 ltorg 토큰을 새로 만드는 역할을 한다.
* extref리스트는 extref 토큰을 만났을 때 그 문자열을 포함하는 extref 클래스를 생성해 extref리스트에 추가하여 나중에 extref를 사용하는 토큰을 만날 때마다 해당 extref 클래스에 주소와 + or – 정보를 추가하여 modify 부분을 만들 때 사용하는 역할을 한다.

라인마다 토큰을 만들어 토큰리스트에 추가. 토큰 클래스를 생성할 때 문자열 파라미터로 받아 파싱하여 label operator operand comment 별로 구분해 저장

putToken 함수를 통해 파싱한 토큰 클래스를 해석함

* 토큰 리스트의 static변수 pc를 통해 토큰을 생성할 때 토큰의 주소 값을 구함
* 토큰을 생성할 때 토큰을 해석하여 토큰의 오브젝트코드 크기만큼 pc값을 증가시킴
* 토큰 operator가 inst테이블에 없을 경우 조건문을 통해 operator 별로 다른 토큰해석 방법을 사용
* 토큰 operator가 inst테이블에 있을 경우 inst를 통해 format을 구해 토큰에 저장하고 토큰을 해석하여 I,n,x flag를 설정함
* 토큰 operator가 end거나 csect일 경우 return값을 달리하여 pass1 함수에 object코드가 끝났음을 알림

Pass2은 pass1에서 해석한 토큰을 이용해 오브젝트코드 문자열을 생성하고 코드의 바이트크기를 토큰에 저장한다.

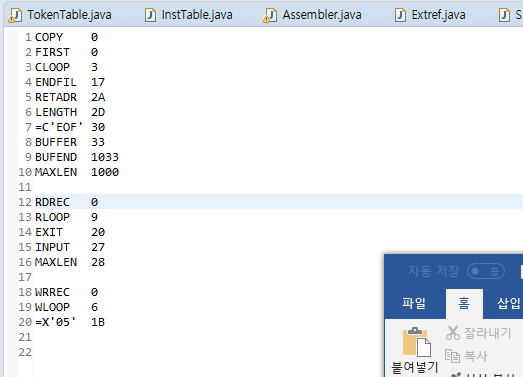
* 오브젝트가 끝나는 토큰을 만나면 extref리스트를 이용해 modify코드를 만들고 해당 토큰의 오브젝트코드를 modify코드로 사용한다.

printObjectCode 함수를 이용해 Pass2로 만든 오브젝트코드 문자열을 파일에 출력한다

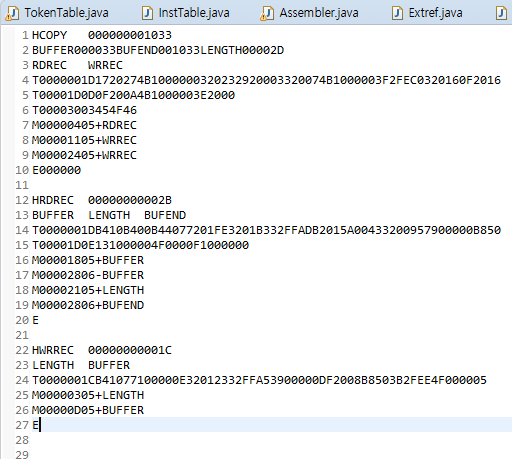
* 출력한 토큰의 바이트크기를 누적해 그 크기가 0x1D 이상일 경우 텍스트 라인의 정보부분을 생성해 출력하고 개행한다.
* Resw resb인경우도 텍스트 라인의 정보부분을 생성해 출력하고 개행한다.
* 오브젝트가 끝난 경우 pass2에서 생성한 modify코드를 출력하고 end부분을 출력함

1. 수행결과

Symtab\_20142385



Output\_20142385



1. 결론 및 보충할 점

객체별로 프로그램을 만들어 쓰는 자바의 이점을 사용하지 못하고 프로젝트 1-a에 사용했던 코드를 그대로 가져와서 프로그램을 만들었다. 최대한 제공된 모듈을 지켜 c코드를 자바 코드에 옮겼다.

1. 소스코드

Assembler.java

**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.HashMap;

**import** java.util.Map;

/\*\*

\* 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;

/\*\*

\* Token, 또는 지시어에 따라 만들어진 오브젝트 코드들을 출력 형태로 저장하는 공간. <br>

\* 필요한 경우 String 대신 별도의 클래스를 선언하여 ArrayList를 교체해도 무방함.

\*/

ArrayList<String> codeList;

/\*\*

\* 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>();

TokenList = **new** ArrayList<TokenTable>();

codeList = **new** ArrayList<String>();

}

/\*\*

\* 어셐블러의 메인 루틴

\*/

**public** **static** **void** main(String[] args) {

Assembler assembler = **new** Assembler("inst.data");

assembler.loadInputFile("input.txt");

assembler.pass1();

assembler.printSymbolTable("symtab\_20142385");

assembler.pass2();

assembler.printObjectCode("output\_20142385");

}

/\*\*

\* 작성된 codeList를 출력형태에 맞게 출력한다.<br>

\* **@param** fileName : 저장되는 파일 이름

\*/

**private** **void** printObjectCode(String fileName) {

// **TODO** Auto-generated method stub

File file = **new** File(fileName);

BufferedWriter bufferedWriter;

Token token = **null**;

String str = **null**;

String text = "";

**int** start = 0; //line의 text 시작

**int** end = 0; //line의 text 끝

**try** {

bufferedWriter = **new** BufferedWriter(**new** FileWriter(file));

**for** (**int** i = 0; i < TokenList.size(); i++) {

**for** (**int** j = 0; j < TokenList.get(i).size(); j++) {

//출력할 토큰 저장

token = TokenList.get(i).getToken(j);

//head 출력, start end 값 설정

**if** (token.getOperator().equals("START")) {

start = token.getLocation();

end = start;

**if**(file.isFile() && file.canWrite()){

bufferedWriter.write(token.getObjectCode());

bufferedWriter.newLine();

}

}

**else** **if** (token.getOperator().equals("CSECT") || token.getOperator().equals("END")) {

//head 출력, start end 값 설정

**if** (j == 0) {

start = token.getLocation();

end = start;

**if**(file.isFile() && file.canWrite()){

bufferedWriter.write(token.getObjectCode());

bufferedWriter.newLine();

}

}

**else** {

//text에 값이 있으면 text 라인 정보를 생성하고 text 출력

**if** (text.length() > 0) {

text = String.*format*("T%06X%02X", start, end - start - 1).concat(text);

**if**(file.isFile() && file.canWrite()){

bufferedWriter.write(text);

bufferedWriter.newLine();

}

start = end;

text = "";

}

//modify 출력

**if**(file.isFile() && file.canWrite()){

bufferedWriter.write(token.getObjectCode());

}

//End 출력

**if** (TokenList.get(i).getToken(0).getOperator().equals("START"))

str = String.*format*("E%06X", token.getLocation() - TokenList.get(i).getToken(0).getLocation());

**else**

str = "E";

**if**(file.isFile() && file.canWrite()){

bufferedWriter.write(str);

bufferedWriter.newLine();

}

}

}

//resw이나 resb 토큰이면 text에 값이 있는 경우 text 라인 정보를 생성하고 text 출력

**else** **if** (token.getOperator().equals("RESW") || token.getOperator().equals("RESB")) {

**if** (text.length() > 0) {

text = String.*format*("T%06X%02X", start, end - start).concat(text);

**if**(file.isFile() && file.canWrite()){

bufferedWriter.write(text);

bufferedWriter.newLine();

}

start = end;

text = "";

}

start = end + token.getByteSize();

}

//extref extdef 코드 출력

**else** **if** (token.getOperator().equals("EXTREF") || token.getOperator().equals("EXTDEF")) {

**if**(file.isFile() && file.canWrite()){

bufferedWriter.write(token.getObjectCode());

bufferedWriter.newLine();

}

}

//inst에 있는 operator 토큰

**else** **if** (token.getOperator().equals("EQU") == **false**){

//text가 0x1D 이상이면 text 라인 정보를 생성하고 출력

**if** (end - start >= 0x1D) {

text = String.*format*("T%06X%02X", start, end - start).concat(text);

**if**(file.isFile() && file.canWrite()){

bufferedWriter.write(text);

bufferedWriter.newLine();

}

start = end;

text = "";

}

//text에 코드를 저장

**else**

text = text.concat(token.getObjectCode());

}

//text쓴 만큼 증가

end += token.getByteSize();

}

//object 끝나면 개행

**if**(file.isFile() && file.canWrite()){

bufferedWriter.newLine();

}

}

bufferedWriter.close();

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

/\*\*

\* 작성된 SymbolTable들을 출력형태에 맞게 출력한다.<br>

\* **@param** fileName : 저장되는 파일 이름

\*/

**private** **void** printSymbolTable(String fileName) {

// **TODO** Auto-generated method stub

File file = **new** File(fileName);

BufferedWriter bufferedWriter;

**try** {

bufferedWriter = **new** BufferedWriter(**new** FileWriter(file));

**for** (**int** i = 0; i < symtabList.size(); i++) {

SymbolTable symTab = symtabList.get(i);

**for** (**int** j = 0; j < symTab.length(); j++) {

**if**(file.isFile() && file.canWrite()){

bufferedWriter.write(symTab.getSymbol(j) + "\t" + Integer.*toString*(symTab.getLocation(j), 16).toUpperCase());

bufferedWriter.newLine();

}

}

bufferedWriter.newLine();

}

bufferedWriter.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 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 선언되어야 한다.

\*/

**private** **void** pass1() {

// **TODO** Auto-generated method stub

SymbolTable symTab = **new** SymbolTable();

TokenTable tokenTab = **new** TokenTable(symTab, instTable);

**int** returnValue;

**for** (**int** i = 0; i < lineList.size(); i++) {

returnValue = tokenTab.putToken(lineList.get(i));

//return값이 1이나 2면 오브젝트 끝으로 새로운 토큰리스트를 만든다

**if** (returnValue == 1) {

**if** (symTab.length() > 1) {

symtabList.add(symTab);

TokenList.add(tokenTab);

symTab = **new** SymbolTable();

tokenTab = **new** TokenTable(symTab, instTable);

**if** (returnValue == 1)

i--;

}

}

}

}

/\*\*

\* 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 < TokenList.size(); i++) {

**for** (**int** j = 0; j < TokenList.get(i).size(); j++) {

TokenList.get(i).makeObjectCode(j);

}

}

}

/\*\*

\* inputFile을 읽어들여서 lineList에 저장한다.<br>

\* **@param** inputFile : input 파일 이름.

\*/

**private** **void** loadInputFile(String inputFile) {

// **TODO** Auto-generated method stub

String str;

**try** {

File file = **new** File(inputFile);

FileReader filereader = **new** FileReader(file);

BufferedReader bufReader = **new** BufferedReader(filereader);

**while**((str = bufReader.readLine()) != **null**){

lineList.add(str);

}

bufReader.close();

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

InstTable.java

**import** java.util.HashMap;

**import** java.io.BufferedReader;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

/\*\*

\* 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.

\*/

**private** 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) {

//...

Instruction newInstruction;

String str;

**try** {

File file = **new** File(fileName);

FileReader filereader = **new** FileReader(file);

BufferedReader bufReader = **new** BufferedReader(filereader);

**while**((str = bufReader.readLine()) != **null**){

newInstruction = **new** Instruction(str);

instMap.put(newInstruction.getInstruction(), newInstruction);

}

bufReader.close();

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**public** Instruction getInstruction(String str) {

**if** (str.charAt(0) == '+')

**return** instMap.get(str.substring(1));

**else**

**return** instMap.get(str);

}

//get, set, search 등의 함수는 자유 구현

}

/\*\*

\* 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;

\*/

/\*\* instruction이 몇 바이트 명령어인지 저장. 이후 편의성을 위함 \*/

**private** **int** format;

**private** **int** opcode;

**private** String instruction;

**private** **int** numberOfOperand;

/\*\*

\* 클래스를 선언하면서 일반문자열을 즉시 구조에 맞게 파싱한다.

\* **@param** line : instruction 명세파일로부터 한줄씩 가져온 문자열

\*/

**public** Instruction(String line) {

parsing(line);

}

/\*\*

\* 일반 문자열을 파싱하여 instruction 정보를 파악하고 저장한다.

\* **@param** line : instruction 명세파일로부터 한줄씩 가져온 문자열

\*/

**public** **void** parsing(String line) {

// **TODO** Auto-generated method stub

String[] array = line.split(" ");

**if** (array.length != 4) {

System.***err***.println("inst error " + array.length);

System.*exit*(-1);

}

instruction = array[0];

format = Integer.*parseInt*(array[1]);

opcode = Integer.*parseInt*(array[2], 16);

numberOfOperand = Integer.*parseInt*(array[3]);

}

**public** **int** getFormat() {

**return** format;

}

**public** **int** getOpcode() {

**return** opcode;

}

**public** **int** getNumberOfOperand() {

**return** numberOfOperand;

}

**public** String getInstruction() {

**return** instruction;

}

//그 외 함수 자유 구현

}

SymbolTable.java

**import** java.util.ArrayList;

/\*\*

\* symbol과 관련된 데이터와 연산을 소유한다.

\* section 별로 하나씩 인스턴스를 할당한다.

\*/

**public** **class** SymbolTable {

ArrayList<String> symbolList;

ArrayList<Integer> locationList;

// 기타 literal, external 선언 및 처리방법을 구현한다.

**public** SymbolTable() {

symbolList = **new** ArrayList<String>();

locationList = **new** ArrayList<Integer>();

}

/\*\*

\* 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 코드에 문제가 있음을 나타낸다.

\* 매칭되는 주소값의 변경은 modifySymbol()을 통해서 이루어져야 한다.

\*/

**public** **void** putSymbol(String symbol, **int** location) {

symbolList.add(symbol);

locationList.add(location);

}

**public** String getSymbol(**int** index) {

**return** symbolList.get(index);

}

**public** Integer getLocation(**int** index) {

**return** locationList.get(index);

}

/\*\*

\* 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) {

**int** index = symbolList.indexOf(symbol);

locationList.set(index, 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** search(String symbol) {

**int** address = 0;

**int** index;

//...

**if** (symbol.charAt(0) == '#' || symbol.charAt(0) == '@')

symbol = symbol.substring(1);

index = symbolList.indexOf(symbol);

**if** (index == -1)

**return** -1;

address = locationList.get(index);

**return** address;

}

**public** **int** length() {

**return** symbolList.size();

}

}

ExtRef.java

**import** java.util.ArrayList;

**public** **class** Extref {

String ref; //extref 문자

ArrayList<Integer> locationList; //addr

ArrayList<Boolean> isSubList; //+ or -

**public** Extref(String ref) {

**this**.ref = ref;

locationList = **new** ArrayList<Integer>();

isSubList = **new** ArrayList<Boolean>();

}

**public** **void** add(**int** location, Boolean isSub) {

locationList.add(location);

isSubList.add(isSub);

}

**public** **int** length() {

**return** locationList.size();

}

**public** String getRef( ) {

**return** ref;

}

**public** **int** getLocation(**int** index) {

**return** locationList.get(index);

}

**public** **boolean** isSub(**int** index) {

**return** isSubList.get(index);

}

}

TokenTable.java

**import** java.util.ArrayList;

**import** javax.script.ScriptEngineManager;

**import** javax.script.ScriptException;

**import** javax.script.ScriptEngine;

/\*\*

\* 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;

//레지스터

**public** **static** **final** **int** ***aReg***=0;

**public** **static** **final** **int** ***xReg***=1;

**public** **static** **final** **int** ***lReg***=2;

**public** **static** **final** **int** ***bReg***=3;

**public** **static** **final** **int** ***sReg***=4;

**public** **static** **final** **int** ***tReg***=5;

**public** **static** **final** **int** ***fReg***=6;

**public** **static** **final** **int** ***pcReg***=7;

**public** **static** **final** **int** ***swReg***=8;

//문자열 수식계산

ScriptEngineManager mgr = **new** ScriptEngineManager();

ScriptEngine engine = mgr.getEngineByName("JavaScript");

/\* Token을 다룰 때 필요한 테이블들을 링크시킨다. \*/

SymbolTable symTab;

InstTable instTab;

ArrayList<String> ltorg;

//extref 배열

ArrayList<Extref> extrefList;

/\*\* 각 line을 의미별로 분할하고 분석하는 공간. \*/

ArrayList<Token> tokenList;

//pass1에 쓰일 pc addr

**private** **static** **int** *pc*;

/\*\*

\* 초기화하면서 symTable과 instTable을 링크시킨다.

\* **@param** symTab : 해당 section과 연결되어있는 symbol table

\* **@param** instTab : instruction 명세가 정의된 instTable

\*/

**public** TokenTable(SymbolTable symTab, InstTable instTab) {

//...

**this**.symTab = symTab;

**this**.instTab = instTab;

ltorg = **new** ArrayList<String>();

extrefList = **new** ArrayList<Extref>();

tokenList = **new** ArrayList<Token>();

}

/\*\*

\* 일반 문자열을 받아서 Token단위로 분리시켜 tokenList에 추가한다.

\* **@param** line : 분리되지 않은 일반 문자열

\*/

**public** **int** putToken(String line) {

//주석무시

**if** (line.charAt(0) == '.')

**return** -1;

**int** returnValue = 0; //리턴값 1일때 csect end 토큰, 그 외 토큰 0

**int** format = 0;

Token newToken = **new** Token(line);

//intruction

Instruction instruction = instTab.getInstruction(newToken.getOperator());

//라벨 있으면 심볼에 추가

**if** (newToken.getLabel().length() != 0) {

**if** (newToken.getOperator().equals("CSECT") == **false** || symTab.length() == 0) {

**if** (symTab.search(newToken.getLabel()) == -1)

symTab.putSymbol(newToken.getLabel(), *pc*);

**else** {

System.***err***.println("label error");

System.*exit*(-1);

}

}

}

//operator가 inst에 있는 경우

**if** (instruction == **null**) {

**if** (newToken.getOperator().equals("END") || newToken.getOperator().equals("CSECT")

|| newToken.getOperator().equals("LTORG")) {

**if** (newToken.getOperator().equals("END") || newToken.getOperator().equals("CSECT"))

returnValue = 1;

//ltorg를 만났거나 object가 끝난경우 ltorg 배열을 토큰으로 생성

**if** (ltorg.size() > 0) {

Token[] ltToken = **new** Token[ltorg.size()];

String str;

**for** (**int** i = 0; i < ltorg.size(); i++) {

ltToken[i] = **new** Token();

str = ltorg.get(i);

symTab.putSymbol(str, *pc*);

ltToken[i].setOperator("=");

ltToken[i].setOperand(str.substring(1), 0);

**if** (str.charAt(1) == 'X') {

*pc* += 1;

}

**else** **if** (str.charAt(1) == 'C') {

*pc* += str.length() - 4;

}

ltToken[i].setLocation(*pc*);

tokenList.add(ltToken[i]);

}

ltorg = **new** ArrayList<String>();

}

}

//start와 csect 로 시작하는 오브젝트의 pc값

**if** (newToken.getOperator().equals("START"))

*pc* = Integer.*valueOf*(newToken.getOperand(0));

**if** (newToken.getOperator().equals("CSECT"))

*pc* = 0;

//EQU

**if** (newToken.getOperator().equals("EQU")) {

String str = newToken.getOperand(0);

//\*을 현재 주소값으로 대체

str = str.replaceAll("\\\*", Integer.*toString*(*pc*));

//심볼을 주소값으로 대체

**for** (**int** i = 0; i < symTab.length(); i++) {

str = str.replaceAll(symTab.getSymbol(i), symTab.getLocation(i).toString());

}

**try** {

//문자열 수식을 계산해 수정

symTab.modifySymbol(newToken.getLabel(), (**int**) engine.eval(str));

} **catch** (ScriptException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

//resw, resb, word, byte

**if** (newToken.getOperator().equals("RESW")) {

*pc* += Integer.*valueOf*(newToken.getOperand(0)) \* 3;

}

**if** (newToken.getOperator().equals("RESB")) {

*pc* += Integer.*valueOf*(newToken.getOperand(0));

}

**if** (newToken.getOperator().equals("WORD")) {

*pc* += 3;

}

**if** (newToken.getOperator().equals("BYTE")) {

**if** (newToken.getOperand(0).charAt(0) == 'X')

*pc* += 1;

**else** **if** (newToken.getOperand(0).charAt(0) == 'C') {

*pc* += newToken.getOperand(0).length() - 3;

}

}

}

// operator가 inst에 있는 경우

**else** {

//format을 계산

**if** (newToken.getOperator().charAt(0) == '+') {

**if** (instruction.getFormat() == 34) {

format = 4;

newToken.setFlag(***eFlag***, 1);

}

**else** {

System.***err***.println("format error");

System.*exit*(-1);

}

}

**else** {

**if** (instruction.getFormat() == 34)

format = 3;

**else**

format = instruction.getFormat();

}

//format이 3이거나 4인경우

**if** (format == 3 || format == 4) {

**if** (newToken.getOperand(0) != **null**) {

//xflag

**if** (newToken.getOperand()[newToken.getOperand().length - 1].equals("X")) {

newToken.setFlag(***xFlag***, 1);

**if** (instruction.getNumberOfOperand() != newToken.getOperand().length - 1) {

System.***err***.println("operand error " + newToken.getOperand(0));

System.*exit*(-1);

}

}

**else** **if** (instruction.getNumberOfOperand() != newToken.getOperand().length) {

System.***err***.println("operand error");

System.*exit*(-1);

}

//iflag, nflag

**if** (newToken.getOperand(0).charAt(0) == '#')

newToken.setFlag(***iFlag***, 1);

**else** **if** (newToken.getOperand(0).charAt(0) == '@')

newToken.setFlag(***nFlag***, 1);

**else** {

newToken.setFlag(***iFlag***, 1);

newToken.setFlag(***nFlag***, 1);

}

**if** (newToken.getOperand(0).charAt(0) == '=') {

**if** (ltorg.contains(newToken.getOperand(0)) == **false**) {

ltorg.add(newToken.getOperand(0));

}

}

}

**else** {

newToken.setFlag(***iFlag***, 1);

newToken.setFlag(***nFlag***, 1);

}

}

//format만큼 pc증가

*pc* += format;

}

//토큰추가

newToken.setLocation(*pc*);

newToken.setFormat(format);

tokenList.add(newToken);

**return** returnValue;

}

/\*\*

\* tokenList에서 index에 해당하는 Token을 리턴한다.

\* **@param** index

\* **@return** : index번호에 해당하는 코드를 분석한 Token 클래스

\*/

**public** Token getToken(**int** index) {

**return** tokenList.get(index);

}

/\*\*

\* Pass2 과정에서 사용한다.

\* instruction table, symbol table 등을 참조하여 objectcode를 생성하고, 이를 저장한다.

\* **@param** index

\*/

**public** **void** makeObjectCode(**int** index){

//...

Token token = tokenList.get(index); //토큰 버퍼

String str = ""; //문자열 버퍼

**if** (token.getFormat() == 0) {

//오브젝트 시작토큰

**if** (token.getOperator().equals("START") || token.getOperator().equals("CSECT")) {

**if** (index == 0) {

Token lastToken = tokenList.get(tokenList.size() - 2);

str = "H";

str = str.concat(token.getLabel());

str = String.*format*("%s\t%06X%06X", str, token.getLocation(), lastToken.getLocation());

}

}

//오브젝트 마지막토큰 extrefList를 읽어 modify 코드를 생성

**if** (token.getOperator().equals("CSECT") || token.getOperator().equals("END")) {

**if** (index != 0) {

Extref extref = **null**;

**for** (**int** i = 0; i < extrefList.size(); i++) {

extref = extrefList.get(i);

**for** (**int** j = 0; j < extref.length(); j++) {

**if** (extref.isSub(j))

str = str.concat(String.*format*("M%08X-%s\n", extref.getLocation(j), extref.getRef()));

**else**

str = str.concat(String.*format*("M%08X+%s\n", extref.getLocation(j), extref.getRef()));

}

}

}

}

//extdef 코드 생성

**if** (token.getOperator().equals("EXTDEF")) {

str = "";

**for** (**int** i = 0; i < token.getOperand().length; i++) {

str = str.concat(token.getOperand(i));

str = str.concat(String.*format*("%06X", symTab.search(token.getOperand(i))));

}

}

//extref 코드 생성

**if** (token.getOperator().equals("EXTREF")) {

str = "";

**for** (**int** i = 0; i < token.getOperand().length; i++) {

//extrefList에 추가

extrefList.add(**new** Extref(token.getOperand(i)));

str = str.concat(token.getOperand(i) + "\t");

}

}

//resw, rewb

**if** (token.getOperator().equals("RESW") || token.getOperator().equals("RESB")) {

str = token.getOperator();

**if** (token.getOperator().equals("RESB"))

token.setByteSize(Integer.*valueOf*(token.getOperand(0)));

**else**

token.setByteSize(Integer.*valueOf*(token.getOperand(0)) \* 3);

}

//word

**if** (token.getOperator().equals("WORD")) {

str = token.getOperand(0);

str = str.replaceAll("\\\*", Integer.*toString*(*pc*));

//심볼을 주소값으로 대체

**for** (**int** i = 0; i < symTab.length(); i++) {

str = str.replaceAll(symTab.getSymbol(i), symTab.getLocation(i).toString());

}

//extref를 0으로 대체하고 extrefList에 주소값과 +- 정보를 추가

**for** (**int** i = 0; i < extrefList.size(); i++) {

//extrefList에 주소값과 +- 정보를 추가

**for** (**int** j = str.indexOf(extrefList.get(i).getRef()); j >= 0; j = str.indexOf(extrefList.get(i).getRef(), j+1)) {

**if** (j == 0)

extrefList.get(i).add((token.getLocation() - 3) \* 16 \* 16 + 6, **false**);

**else** **if** (str.charAt(j-1) == '+')

extrefList.get(i).add((token.getLocation() - 3) \* 16 \* 16 + 6, **false**);

**else** **if** (str.charAt(j-1) == '-')

extrefList.get(i).add((token.getLocation() - 3) \* 16 \* 16 + 6, **true**);

}

//extref를 0으로 대체

**if** (str.equals(str.replaceAll(extrefList.get(i).getRef(), "000000")) == **false**) {

str = str.replaceAll(extrefList.get(i).getRef(), "000000");

}

str = str.replaceAll(extrefList.get(i).getRef(), "000000");

}

**try** {

//문자열 수식을 계산

str = String.*format*("%06X", (**int**) engine.eval(str));

} **catch** (ScriptException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

token.setByteSize(3);

}

//byte이거나 ltorg 토큰인 경우

**if** (token.getOperator().equals("BYTE") || token.getOperator().equals("=")) {

str = "";

**if** (token.getOperand(0).charAt(0) == 'X') {

str = str.concat(token.getOperand(0).substring(2, token.getOperand(0).length() - 1));

}

**else** **if** (token.getOperand(0).charAt(0) == 'C') {

**int** n;

**for** (**int** i = 2; i < token.getOperand(0).length() - 1; i++) {

n = token.getOperand(0).charAt(i);

str = str.concat(String.*format*("%02X", n));

}

}

token.setByteSize(token.getOperand(0).length() - 3);

}

}

**else** {

**int** code = 0;

**int** addr = 0;

**int** mask = 0;

Extref extref = **null**;

Instruction inst = instTab.getInstruction(token.getOperator());

//코드에 oparator 추가

code |= inst.getOpcode() << (token.getFormat() - 1) \* 8;

//format 1

**if** (token.getFormat() == 1) {

str = String.*format*("%02X", inst.getOpcode());

}

//format 2

**else** **if** (token.getFormat() == 2) {

**for** (**int** i = 0; i < token.getOperand().length; i++) {

code |= **this**.getRegister(token.getOperand(i)) << (4 - i \* 4);

}

str = String.*format*("%04X", code);

}

//format 3, 4

**else** {

//xflag, bflag, pflag

**if** (inst.getNumberOfOperand() != 0) {

**if** (token.getOperand(token.getOperand().length - 1).equals("X"))

token.setFlag(***xFlag***, 1);

**if** (token.getFormat() == 3 && token.getOperand().length > 0) {

**if** (Integer.*signum*(token.getLocation() - symTab.search(token.getOperand(0))) >= (1 << 9))

token.setFlag(***bFlag***, 1);

**else**

token.setFlag(***pFlag***, 1);

}

}

//코드에 nixbpe 추가

code |= token.getNixbpe() << ((token.getFormat() -3) \* 8 + 12);

//operand 있는 경우

**if** (inst.getNumberOfOperand() > 0) {

//operand가 symbol table에 없는 경우

**if** ((addr = symTab.search(token.getOperand(0))) == -1) {

//extref 서치

**for** (**int** i = 0; i < extrefList.size(); i++) {

**if** (extrefList.get(i).getRef().equals(token.getOperand(0))) {

extref = extrefList.get(i);

**break**;

}

**else** **if** (extrefList.get(i).getRef().equals(token.getOperand(0).substring(1))) {

extref = extrefList.get(i);

**break**;

}

}

//operand가 extref에 없는경우 addr값 계산

**if** (extref == **null**) {

**if** (token.getOperand(0).charAt(0) == '#')

addr = Integer.*valueOf*(token.getOperand(0).substring(1));

**else** {

addr = Integer.*valueOf*(token.getOperand(0));

addr -= token.getLocation();

}

}

//operand가 extref에 있는 경우 extrefList에 주소값추가하고 addr을 0

**else** {

**int** extrefAddr;

extrefAddr = (token.getLocation() - (token.getFormat() - 1)) \* 16 \* 16;

extrefAddr += token.getFormat() \* 2 - 3;

extref.add(extrefAddr, **false**);

addr = 0;

}

}

//operand가 symtable에 있는 경우 addr 계산

**else** {

addr = symTab.search(token.getOperand(0));

addr -= token.getLocation();

}

}

//operand가 없는 경우 addr 0

**else**

addr = 0;

//addr 마스크

**for** (**int** i = 0; i < token.getFormat() \* 4; i++) {

mask |= 1 << i;

}

addr &= mask;

code |= addr;

}

//code를 문자열로 생성

**if** (token.getFormat() == 4) {

str = String.*format*("%08X", code);

token.setByteSize(4);

}

**else** **if** (token.getFormat() == 3) {

str = String.*format*("%06X", code);

token.setByteSize(3);

}

**else** **if** (token.getFormat() == 2) {

str = String.*format*("%04X", code);

token.setByteSize(2);

}

**else** **if** (token.getFormat() == 1) {

str = String.*format*("%02X", code);

token.setByteSize(1);

}

}

//objectcode 추가

token.setObjectCode(str);

}

/\*\*

\* index번호에 해당하는 object code를 리턴한다.

\* **@param** index

\* **@return** : object code

\*/

**public** String getObjectCode(**int** index) {

**return** tokenList.get(index).getObjectCode();

}

//레지스터값 리턴

**public** **int** getRegister(String str) {

**int** returnVal = -1;

**if** (str.equals("A"))

returnVal = TokenTable.***aReg***;

**else** **if** (str.equals("X"))

returnVal = TokenTable.***xReg***;

**else** **if** (str.equals("L"))

returnVal = TokenTable.***lReg***;

**else** **if** (str.equals("B"))

returnVal = TokenTable.***bReg***;

**else** **if** (str.equals("S"))

returnVal = TokenTable.***sReg***;

**else** **if** (str.equals("T"))

returnVal = TokenTable.***tReg***;

**else** **if** (str.equals("F"))

returnVal = TokenTable.***fReg***;

**else** **if** (str.equals("PC"))

returnVal = TokenTable.***pcReg***;

**else** **if** (str.equals("SW"))

returnVal = TokenTable.***swReg***;

**return** returnVal;

}

//tokentable 사이즈

**public** **int** size() {

**return** **this**.tokenList.size();

}

}

/\*\*

\* 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

**private** **int** location;

**private** String label;

**private** String operator;

**private** String[] operand;

**private** String comment;

**private** **int** nixbpe;

**private** **int** format;

// object code 생성 단계에서 사용되는 변수들

**private** String objectCode;

**private** **int** byteSize;

/\*\*

\* 클래스를 초기화 하면서 바로 line의 의미 분석을 수행한다.

\* **@param** line 문장단위로 저장된 프로그램 코드

\*/

**public** Token(String line) {

//initialize 추가

parsing(line);

}

**public** Token() {

operand = **new** String[3];

}

/\*\*

\* line의 실질적인 분석을 수행하는 함수. Token의 각 변수에 분석한 결과를 저장한다.

\* **@param** line 문장단위로 저장된 프로그램 코드.

\*/

**public** **void** parsing(String line) {

String[] str = line.split("\t");

label = str[0];

operator = str[1];

**if** (str.length > 2 && str[2].length() > 0) {

operand = str[2].split(",");

}

**else**

operand = **new** String[3];

**if** (str.length > 3)

comment = str[3];

nixbpe = 0;

}

/\*\*

\* 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) {

//...

**int** n;

**if** (value == 0) {

n = TokenTable.***nFlag*** + TokenTable.***iFlag*** + TokenTable.***xFlag***

+ TokenTable.***bFlag*** + TokenTable.***pFlag*** + TokenTable.***eFlag***;

n = n - flag;

nixbpe &= n;

}

**else** {

n = flag;

nixbpe |= n;

}

}

/\*\*

\* 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** getNixbpe() {

**return** nixbpe;

}

**public** **int** getLocation() {

**return** location;

}

**public** **void** setLocation(**int** location) {

**this**.location = location;

}

**public** String getOperator() {

**return** operator;

}

**public** **void** setOperator(String str) {

operator = str;

}

**public** String getOperand(**int** index) {

**return** operand[index];

}

**public** String[] getOperand() {

**return** operand;

}

**public** **void** setOperand(String str, **int** index) {

operand[index] = **new** String(str);

}

**public** String getLabel() {

**return** label;

}

**public** **void** setLabel(String str) {

label = str;

}

**public** **void** setFormat(**int** format) {

**this**.format = format;

}

**public** **int** getFormat() {

**return** format;

}

**public** **void** setObjectCode(String str) {

objectCode = str;

}

**public** String getObjectCode() {

**return** objectCode;

}

**public** **void** setByteSize(**int** byteSize) {

**this**.byteSize = byteSize;

}

**public** **int** getByteSize() {

**return** byteSize;

}

}