

설계(프로젝트) 보고서

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

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

| | |
|--------|--|
| 교과목 | 시스템프로그래밍 2020 |
| 프로젝트 명 | PYTHON으로 파서 구현하기 |
| 교과목 교수 | 최 재 영 |
| 제출인 | 전자정보공학부 학번: 20160433 성명: 김민정 (출석번호: 108) |
| 제출일 | 2020년 5월 20 일 |

차 례

1장 프로젝트 개요

1.1 개발 배경 및 목적

2장 배경 지식

2.1 주제에 관한 배경지식

2.2 기술적 배경지식

3장 시스템 설계 내용

3.1 전체 시스템 설계 내용

3.2 모듈별 설계 내용

4장 시스템 구현 내용 (구현 화면 포함)

4.1 전체 시스템 구현 내용

4.2 모듈별 구현 내용

4.3 기존에 생성해놓은 inst.data 파일 내용

4.4 디버깅 과정

4.5 수행결과

5장 기대효과 및 결론

첨부 프로그램 소스파일

1장 프로젝트 개요

1.1 개발 배경 및 목적

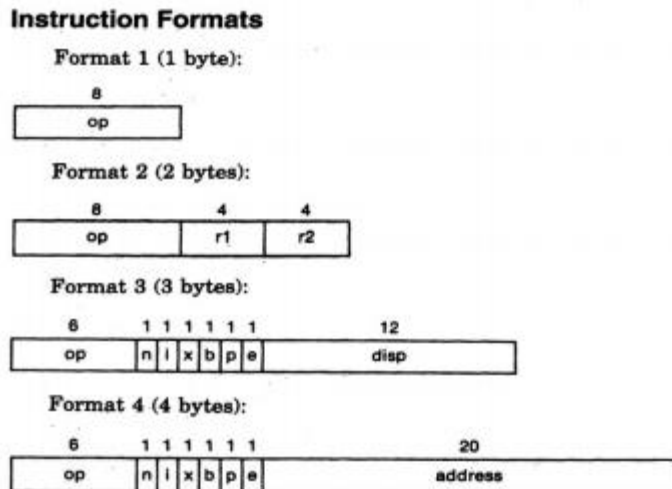
- 주어진 input파일을 이용하여 기본적인 SIC/XE 어셈블러를 구현해보는 프로젝트를 진행해보았다. 이번 프로젝트는 지난번 프로젝트처럼 input 파일을 원하는 object program으로 작성해보는 프로젝트이지만 구현언어가 c와 java에서 python으로 바뀌었다. input파일을 라인별로 읽어들이고 후 각각의 명령어들을 원하는 대로 파싱한 후 파싱한 토큰들을 각각 분석하여 어셈블러가 어떠한 규칙으로 기계어들을 변환하는지를 이해할 수 있다. 또한 object program을 작성하기 위해 input파일의 프로그램을 변환해보면서 그 과정들에 대해 좀 더 명확한 이해가 가능하다. 이를 통해 시스템프로그래밍의 2장 Assemblers를 더 확실하게 공부할 수 있다. 또한 C와 java로 설계해보았던 프로젝트를 python으로 다시 프로그래밍해보면서 프로그램에 대해 다시 복습해볼 수 있고 python 언어에 대한 이해도를 높일 수 있다.

2장 배경 지식

2.1 주제에 관한 배경지식

1) opcode와 명령어 형식

- SIC/XE의 머신은 4종류의 형식을 지원한다.



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

2) 주소지정방식

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

3) assembler program

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

4) object program format

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

다.

□ Header record

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

◆ Define record

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

◆ Refer record

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

□ Text record

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

◆ Modification record (revised)

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

□ End record

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

2.2 기술적 배경지식

1) 파일입출력

– python에서 파일을 다룰 때에는 기본적으로 내장되어 있는 함수인 `open()`를 이용한다. 파일 객체를 열어 파일을 다루는 작업을 한 다음에는 반드시 파일을 닫아야한다. 이 때 사용되는 함수는 `close()`이다. 파일에 데이터를 입력하기 위해서는 mode를 'w(쓰기모드)'로 지정한 후 파일을 열어야 한다. 그 후 `write()`를 이용하여 데이터를 입력한다. 파일에 존재하는 데이터를 읽어오기 위해서는 mode를 'r(읽기모드)'로 지정한 후에 파일을 열어야 한다. 그 후 `readline()`로 한 줄씩 데이터를 읽어온다. 아니면 `read()`를 이용하여 파일의 전체 내용을 한번에 읽어올 수도 있다.

2) 토큰분리

– `split()`를 이용하여 문자열을 분리하였다. `seq` 파라미터에 구분자를 넣어주고 나누고 싶은 개수를 정할 때에는 `maxsplit`에 값을 넣어주면 된다. `split()`는 문자열을 나누어준 후 배열로 저장하여 리턴해주는 함수이다. 기본적으로 `split()`에는 한 개의 구분자만 넣어줄 수 있지만 여러개의 구분자를 넣어주고자 한다면 `re` 모듈을 사용하여 정규표현식으로 구분자를 넣어주면 된다. `re`모듈을 사용할 때는 'import re'와 같이 모듈을 import해주어야 한다.

3) 비트연산

– 비트연산자에는 다음과 같은 연산자들이 있다.

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

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

4) 서식지정자를 활용한 입출력

- 다음과 같은 서식지정자들을 활용해서 입출력을 좀 더 편하게 할 수 있다. 파이썬에서는 format 메서드를 이용하여 서식지정자를 활용할 수 있다.

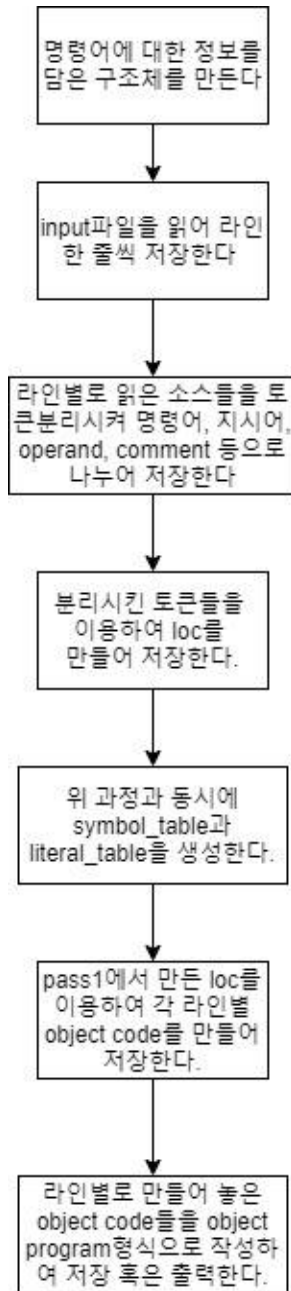
| 서식 지정자 | 설명 | 플래그 | 설명 |
|--------|-------------------|-----|-----------------------|
| d,i | 부호 있는 10진 정수 | - | 왼쪽 정렬 |
| u | 부호 없는 10진 정수 | 공백 | 양수일 때는 공백, 음수일 때는 - |
| s | 문자열 | 0 | 출력하는 폭의 남은 공간에 0으로 채움 |
| X | 부호 없는 16진 정수(대문자) | | |

5) PYTHON

- python은 동적 타이핑 범용 프로그래밍 언어로 다양한 플랫폼에서 쓸 수 있고, 라이브러리(모듈)가 풍부하여 대학을 비롯한 여러 교육 기관, 연구 기관 및 산업계에서 이용이 증가하고 있다. 또한, python은 순수한 프로그램 언어로서의 기능 외에도 다른 언어로 쓰인 모듈들을 연결하는 풀언어(glue language)로써 자주 이용된다. 실제로도 파이썬은 많은 상용 응용 프로그램에서 스크립트 언어로 채용되고 있다. python은 실행시간에 자료형을 검사하는 동적타이핑 언어이고, 속성이나 전용의 메소드 혹은 만들어 제한할 수는 있지만 객체의 멤버에 무제한으로 접근할 수 있다. 또한 모듈, 클래스, 객체와 같은 언어의 요소가 내부에서 접근할 수 있고 리플렉션을 이용한 기술을 쓸 수 있다는 특징이 있다. 현대의 python은 여전히 인터프리터 언어처럼 동작하나 사용자가 모르는 사이에 스스로 파이썬 소스 코드를 컴파일하여 바이트 코드를 만들어 냄으로써 다음에 수행할 때에는 빠른 속도를 보여준다. 또한 python의 한가지 독특한 특성은 들여쓰기를 사용해서 블록을 구분한다는 것이다.python은 python2와 python3으로 구분되는데 두 버전의 문법은 서로 호환되지 않는다. 이 프로그램에서는 python3을 이용하여 코딩하였다.

3장 시스템 설계 내용

3.1 전체 시스템 설계 내용



3.2 모듈별 설계 내용

1) InstTable.py

* 클래스 : InstTable

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

① openFile(self, fileName)

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

② getOpcode(self, instruction)

- 인자로 받은 명령어의 opcode를 찾아 리턴하는 메소드이다.

③ getNumberOfOperand(self, instruction)

- 인자로 받은 명령어의 피연산자 개수를 찾아 리턴하는 메소드이다.

④ getFormat(self, instruction)

- 인자로 받은 명령어의 형식을 찾아 리턴하는 메소드이다.

⑤ isInstruction(self, str)

- 입력문자열이 명령어인지 검사하는 메소드이다.

⑥ isDirective(self, str)

- 입력문자열이 지시어인지 검사하는 메소드이다.

* 클래스 : Instruction

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

① parsing(self, line)

- 일반 문자열을 파싱하여 instruction 정보를 파악하고 저장하는 메소드이다.

2) TokenTable.py

* 클래스 : TokenTable

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

① putToken(self, line)

- 일반 문자열을 받아서 Token단위로 분리시켜 tokenList에 추가하는 메소드이다.

② getToken(self, index)

- tokenList에서 index에 해당하는 Token을 리턴하는 메소드이다.

③ makeObjectCode(self, index)

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

④ getObjectCode(self, index)

- index번호에 해당하는 object code를 리턴하는 메소드이다.

* 클래스 : Token

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

- ① parsing(self, line)
 - line의 실질적인 분석을 수행하는 메소드로 Token의 각 변수에 분석한 결과를 저장한다.
- ② setObjectCode(self, objectcode)
 - object code를 인자로 받아와 저장하는 메소드이다.
- ③ setByteSize(self, byteSize)
 - 의미 분석이 끝난 후 pass2에서 object code로 변형시켰을 때의 byteSize를 인자로 받아와 저장하는 메소드이다.
- ④ setLocation(self, loc)
 - loc값을 인자로 받아와 저장하는 메소드이다.
- ⑤ setFlag(self, flag, value)
 - n,i,x,b,p,e flag를 설정해주는 메소드이다.
- ⑥ getFlag(self, flags)
 - 원하는 flag들의 값을 얻어오는 메소드이다. flag의 조합을 통해 동시에 여러개의 플래그를 얻을 수 있다.

3) SymbolTable.py

* 클래스 : SymbolTable

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

- ① putSymbol(self, symbol, location)
 - 새로운 Symbol을 table에 추가하는 메소드이다. 중복된 symbol은 putSymbol()로 입력될 수 없고 symbol의 주소를 변경하려면 modifySymbol()를 이용한다.
- ② modifySymbol(self, symbol, newLocation)
 - 기존에 존재하는 symbol 값에 대해서 가리키는 주소값을 변경해주는 메소드이다.
- ③ modSymbol(self, symbol, location, size)
 - modification record에 작성할 symbol들을 table에 추가해주는 메소드이다.
- ④ setDefSymbol(self, symbol)
 - 지시어 EXTDEF가 나왔을 때 symbol을 저장하는 메소드이다.
- ⑤ setRefSymbol(self, symbol)
 - 지시어 EXTREF가 나왔을 때 symbol을 저장하는 메소드이다.
- ⑥ searchRefSymbol(self, symbol)
 - 주어진 symbol이 reference된 symbol인지 확인하는 메소드이다.
- ⑦ search(self, symbol)
 - 인자로 전달된 symbol이 어떤 주소를 지칭하는지 알려주는 메소드이다.
- ⑧ getSymbol(self, index)
 - index에 해당하는 symbol을 리턴해주는 메소드이다.
- ⑨ getAddress(self, index)
 - index를 이용하여 symbol을 찾은 후 그 symbol에 해당하는 address를 리턴한다.
- ⑩ getSize(self)

- symboltable의 크기를 리턴해주는 메소드이다.

4) LiteralTable.py

* 클래스 : LiteralTable

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

① putLiteral(self, literal, location)

- 새로운 Literal을 table에 추가해주는 메소드이다. 중복된 literal은 putLiteral()로 입력될 수 없고 literal의 주소를 변경하려면 modifyLiteral()를 이용한다.

② modifyLiteral(self, literal, newLocation)

- 기존에 존재하는 literal 값에 대해서 가리키는 주소값을 변경해주는 메소드이다.

③ search(self, literal)

- 인자로 전달된 literal이 어떤 주소를 지칭하는지 알려주는 메소드이다.

④ addrLiteralTab(self, locctr)

- locctr을 이용하여 literal들의 주소를 넣어주는 메소드이다.

⑤ getLiteral(self, index)

- index에 해당하는 literal을 리턴해주는 메소드이다.

⑥ getAddress(self, index)

- index를 이용하여 literal을 찾은 후 해당하는 literal의 address을 리턴해주는 메소드이다.

⑦ getSize(self)

- literaltable의 크기를 리턴해주는 메소드이다.

5) Assembler.py

* 클래스 : Assembler

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

① loadInputFile(self, inputFile)

- inputFile을 읽어들이어서 lineList에 저장하는 메소드이다.

② pass1(self)

- pass1 과정을 수행하는 메소드이다.

③ printSymbolTable(self, fileName)

- 작성된 SymbolTable들을 출력형태에 맞게 출력해주는 메소드이다.

④ printLiteralTable(self, fileName)

- 작성된 LiteralTable들을 출력형태에 맞게 출력해주는 메소드이다.

⑤ pass2(self)

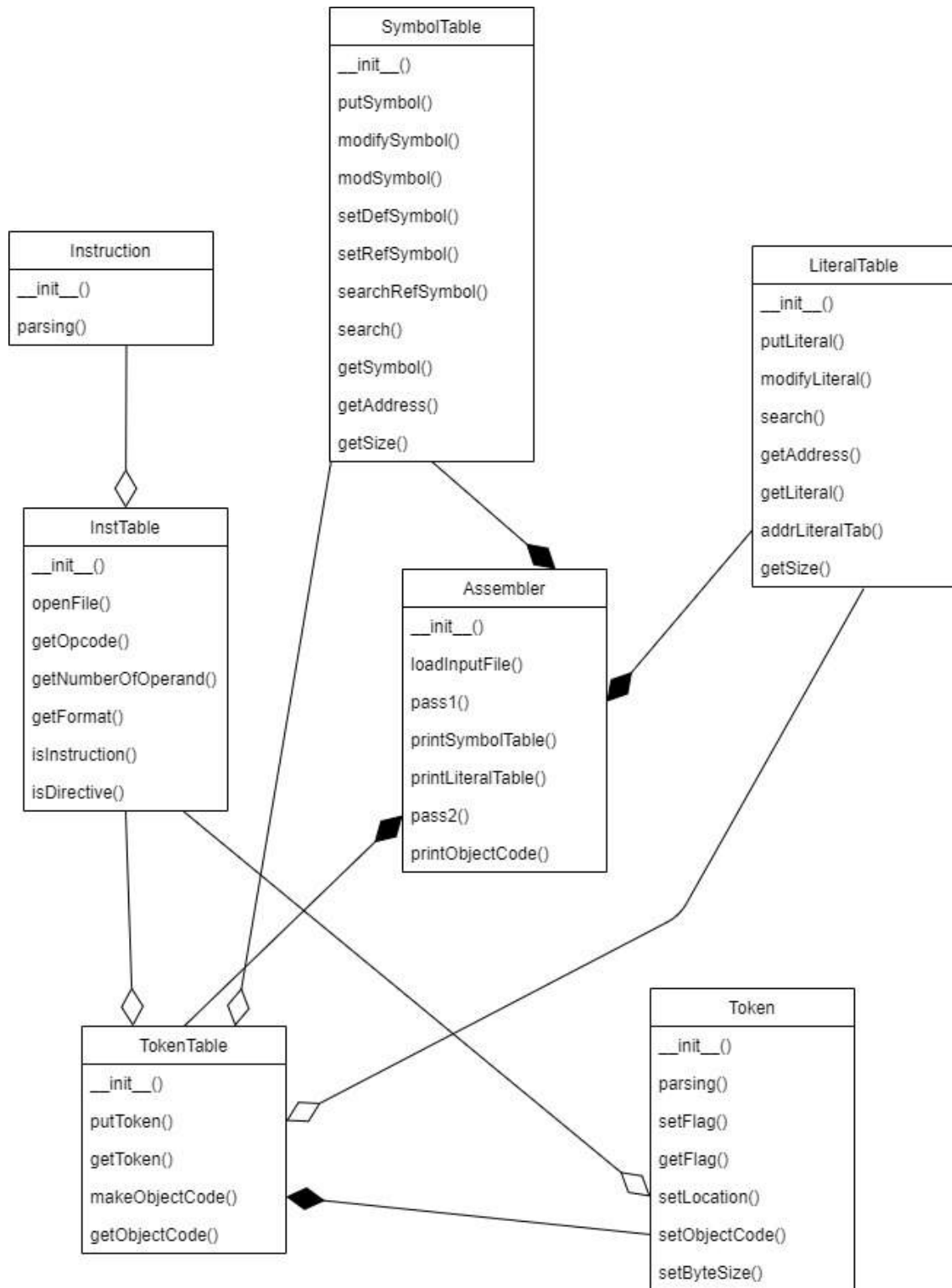
- pass2 과정을 수행하는 메소드이다.

⑥ printObjectCode(self, fileName)

- 이전 과정들에서 작성된 codeList를 출력형태에 맞게 출력해주는 메소드이다.

4장 시스템 구현 내용 (구현 화면 포함)

4.1 전체 시스템 구현 내용



4.2 모듈별 구현 내용

1) InstTable.py

- InstTable 클래스에 명령어에 대한 정보를 담은 파일을 기존에 만들어놓고 그 파일을 한 줄씩 읽어들이어 dictionary자료형의 instMap에 명령어들에 대한 정보를 저장하였다. 그러다음 Instruction 클래스에서 명령어들에 대한 정보를 각각 토큰분리하여 각 정보에 해당하는 변수들에 저장하였다. instMap의 key는 명령어이고 value는 명령어에 대한 정보를 담은 클래스이다.

```
class InstTable:
    def __init__(self, instFile):
        """
        클래스를 초기화하는 함수로 명령어에 대한 정보를 파싱하는 과정을 동시에 처리한다.
        명령어에 대한 정보는 변수 instMap에 저장한다.

        instFile : 명령어에 대한 정보를 저장하고 있는 파일의 이름
        """
        self.instMap = dict() #명령어에 대한 정보를 저장
        self.directive = {"START" : 1, "END" : 1, "BYTE" : 1, "WORD" : 1, "RESB" : 1, "RESW" : 1, "CSECT" : 0, "EXTDEF" : 1, "EXTREF" : 1, "EQU" : 1, "ORG" : 1, "LTORG" : 0} #지시어에 대한 정보를 저장
        self.openFile(instFile)
```

```
class Instruction:
    def __init__(self, line):
        """
        클래스를 선언 및 초기화하면서 동시에 명령어의 구조에 맞게 파싱하여 정보를 저장한다.

        line : 명령어의 정보가 포함된 파일로부터 한 줄씩 읽어들이는 문자열
        """
        self.instruction = "" #명령어의 이름
        self.opcode = 0 #명령어의 opcode 값
        self.numberOfOperand = 0 # 명령어가 몇개의 피연산자를 가지고 있는지
        self.format = 0 #명령어의 형식
        self.parsing(line)
```

2) TokenTable.py

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

```
class TokenTable:
    def __init__(self, symTab, literalTab, instTab):
        """
        클래스를 초기화시키는 함수로 pass2 과정에서 object code로 변환하기 위해 symbolTable과 instTable과 instTable을 링크시킨다.

        symTab : 해당 section과 연결되어있는 symbol table\n
        literalTab : 해당 section과 연결되어있는 literal table\n
        instTab : instruction 명세가 정의된 instTable\n
        """
        #bit 조작의 가독성을 위해 static 변수 선언
        TokenTable.nFlag = 32
        TokenTable.iFlag = 16
        TokenTable.xFlag = 8
        TokenTable.bFlag = 4
        TokenTable.pFlag = 2
        TokenTable.eFlag = 1

        #토큰을 다룰 때 필요한 테이블들을 링크시킨다.
        self.symTab = symTab
        self.literalTab = literalTab
        self.instTab = instTab

        #각 line을 의미별로 분할하고 분석하는 공간
        self.tokenList = list()

        #각 프로그램이 끝날때마다 마지막 주소값을 저장
        self.lastAddr = 0
```

```

class Token:
    def __init__(self, line, instTable):
        """
        클래스를 초기화하면서 바로 line의 의미분석을 수행한다.

        line = line 문장단위로 저장된 프로그램 코드\n
        instTable = 토큰을 파싱하기 위해 instruction table 링크
        """

        #의미 분석 단계에서 사용되는 변수들
        self.location = 0
        self.label = ""
        self.operator = ""
        self.operand = []
        self.comment = ""
        self.nixbpe = 0
        self.plus_check = 0 #operand에 +가 포함되어 있는지에 대한 정보를 저장
        self.directive = "" #명령어 라인 중 지시어
        self.type = "" #해당 명령어 라인이 어떤 타입인지 명시

        #object code 생성 단계에서 사용되는 변수들
        self.objectCode = ""
        self.byteSize = 0

        #instruction에 대한 정보를 참고하기 위해 링크시킨 instruction table
        self.instTable = instTable
        self.parsing(line)

```

3) Assembler.py

- 메인 루틴이 진행되는 부분이다. Assembler 클래스에서 클래스를 초기화하면서 필요한 여러 인스턴스를 생성한다. 프로그램의 메인루틴은 크게 pass1과 pass2로 나눈다. 이 두 함수가 포함되어 있는 모듈이 Assembler이다. pass1은 파일을 라인별로 읽어들이어 의미에 맞게 토큰 분리를 시킨 후 loc를 할당하는 과정이다. 그렇기 때문에 pass1을 진행한 후에는 input파일이 의미에 맞게 토큰분리되어 저장이 되고 각 라인에 맞는 loc가 할당되며 symbol table과 literal table이 만들어진 다. 그 후 이 정보들을 이용하여 pass2를 진행하며 각 라인에 맞는 object code를 만든다. pass2에서는 먼저 TokenTable모듈에서 각 라인별로 object code를 만든 후에 object program의 형식으로 저장한다. 그리고나서 objectProgram 출력함수에서 결과를 출력하면 파일로 저장된 결과를 볼 수 있다.

```

class Assembler:
    def __init__(self, instFile):
        """
        클래스를 초기화하는 함수로 instruction table을 동시에 세팅한다.

        instFile : instruction에 대한 정보를 가지고 있는 파일 이름
        """

        self.instTable = InstTable.InstTable(instFile) #명령어에 대한 정보를 저장
        self.lineList = list() #읽어들인 input 파일의 내용을 한줄씩 저장
        self.symtabList = list() #프로그램의 section별로 symboltable을 저장
        self.literalList = list() #프로그램의 section별로 literaltable을 저장
        self.TokenList = list() #프로그램이 section별로 프로그램을 저장
        self.codeList = list() #object code를 object program형식에 맞춰 작성한 후 저장
        self.locctr = 0 #주소
        self.sectionNum = 0 #프로그램이 총 몇개의 section으로 구성되어 있는지 저장

```

4) SymbolTable.py

- Symbol들을 프로그램별로 관리하기 위해 만든 클래스이다. sub program이

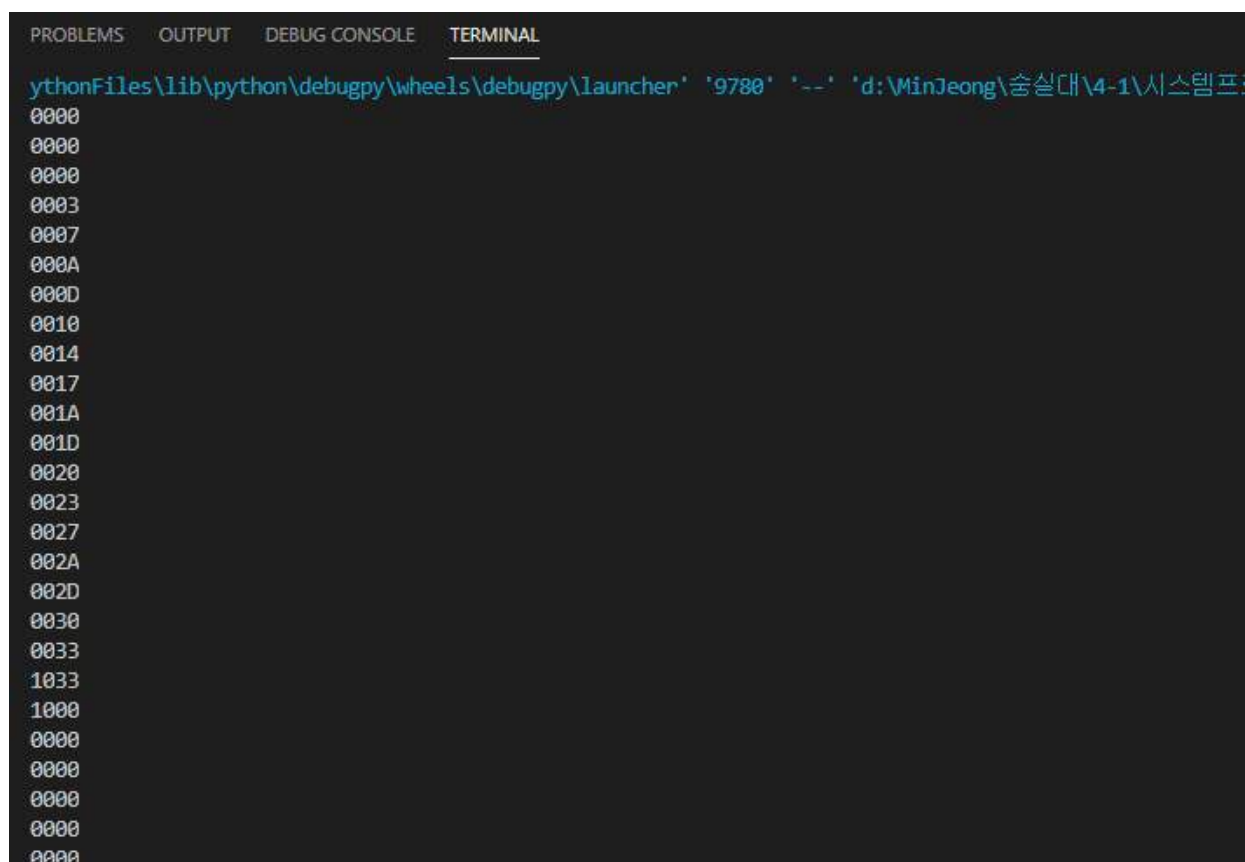
시작될 때마다 새로운 SymbolTable을 만들고 symbol과 address를 저장한다. 또한 object program을 작성하기 위해 modification되는 symbol들을 따로 관리하고 프로그램의 시작부분에 나와있는 def symbol과 ref symbol 또한 따로 관리해주는 클래스이다.

```
class SymbolTable:
    def __init__(self):
        """
        클래스를 초기화한다
        """
        self.symbolList = []
        self.locationList = []
        self.modList = []
        self.modLocationList = []
        self.modSize = []
        self.defList = []
        self.refList = []
```

5) LiteralTable.py

- Literal들을 프로그램별로 관리하기 위해 만든 클래스이다. sub program이 시작될 때마다 새로운 LiteralTable을 만들고 literal과 address를 저장한다.

```
class LiteralTable:
    def __init__(self):
        """
        클래스를 초기화한다.
        """
        self.literallist = []
        self.locationList = []
```



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
pythonFiles\lib\python\debugpy\wheel\debugpy\launcher' '9780' '--' 'd:\MinJeong\숭실대\4-1\시스템프
0000
0000
0000
0003
0007
000A
000D
0010
0014
0017
001A
001D
0020
0023
0027
002A
002D
0030
0033
1033
1000
0000
0000
0000
0000
0000
```

그림 1 pass1을 진행한 후 생성된 loc 중간출력

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

ers\MinJeong_Kim\.vscode\extensions\ms-python.python-2020.5.78807\pythonFiles\
적1c\Project1c\Assembler.py'

172027
4B100000
032023
290000
332007
4B100000
3F2FEC
032016
0F2016
010003
0F200A
4B100000
3E2000

454F46

B410
B400
```

그림 2 pass2를 진행하면서 생성된 각 라인들의 object code 중간출력

4.3 기존에 생성해놓은 inst.data 파일 내용

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

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

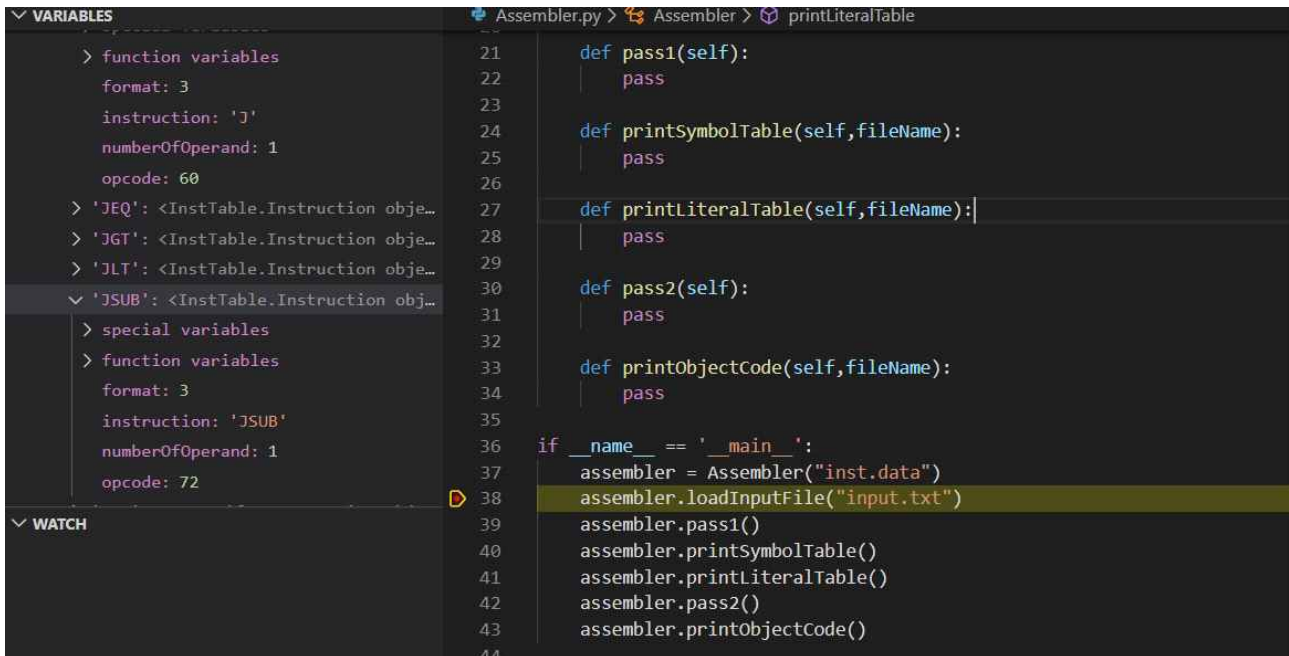
그림 3 inst.data의 내용(1)

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

그림 4 inst.data의 내용(2)

4.4 디버깅 과정

-프로그램을 구현하면서 디버깅했던 내용들 중 일부화면들을 첨부하였다.



```
def pass1(self):
    pass

def printSymbolTable(self, fileName):
    pass

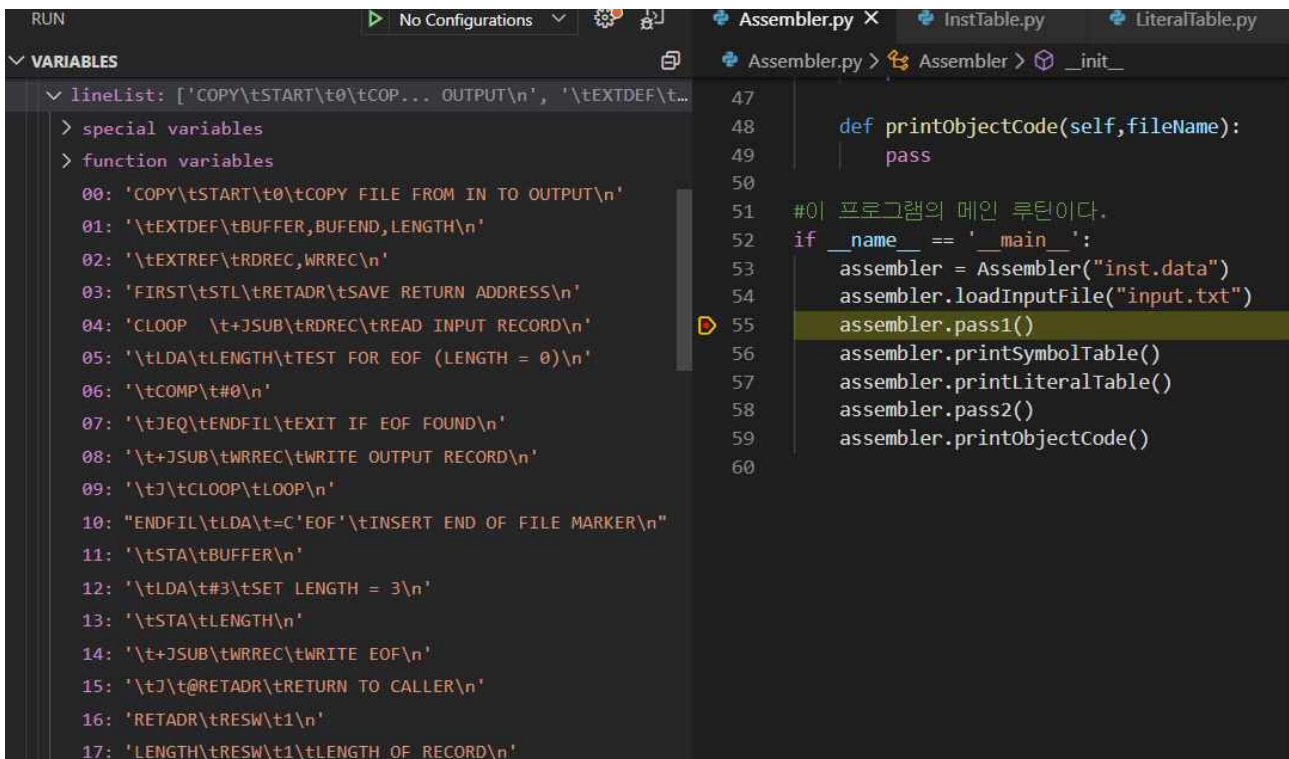
def printLiteralTable(self, fileName):
    pass

def pass2(self):
    pass

def printObjectCode(self, fileName):
    pass

if __name__ == '__main__':
    assembler = Assembler("inst.data")
    assembler.loadInputFile("input.txt")
    assembler.pass1()
    assembler.printSymbolTable()
    assembler.printLiteralTable()
    assembler.pass2()
    assembler.printObjectCode()
```

그림 5 기존에 만들어 놓은 inst.data 파일의 내용을 읽어와 inst_table에 저장하는 과정 디버깅



```
def printObjectCode(self, fileName):
    pass

#이 프로그램의 메인 루틴이다.
if __name__ == '__main__':
    assembler = Assembler("inst.data")
    assembler.loadInputFile("input.txt")
    assembler.pass1()
    assembler.printSymbolTable()
    assembler.printLiteralTable()
    assembler.pass2()
    assembler.printObjectCode()
```

그림 6 input file의 내용을 라인단위로 저장하는 과정 디버깅

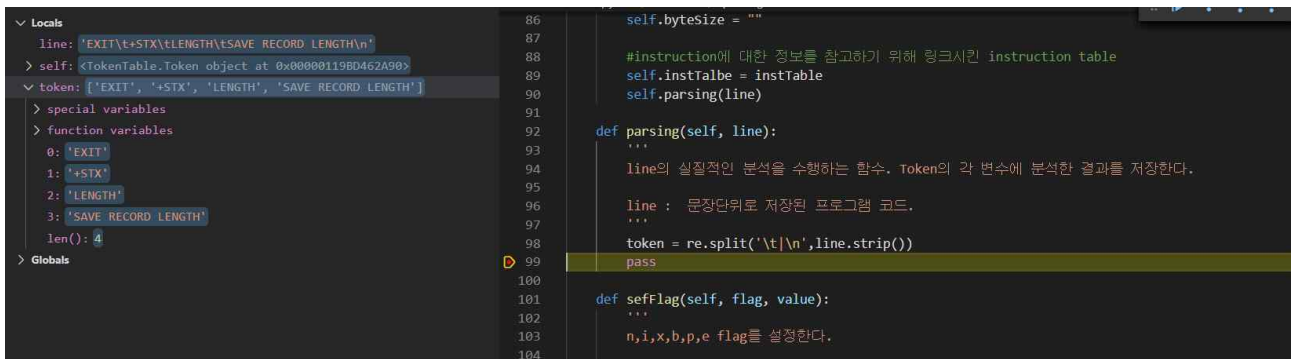


그림 7 pass1()의 parsing과정 디버깅

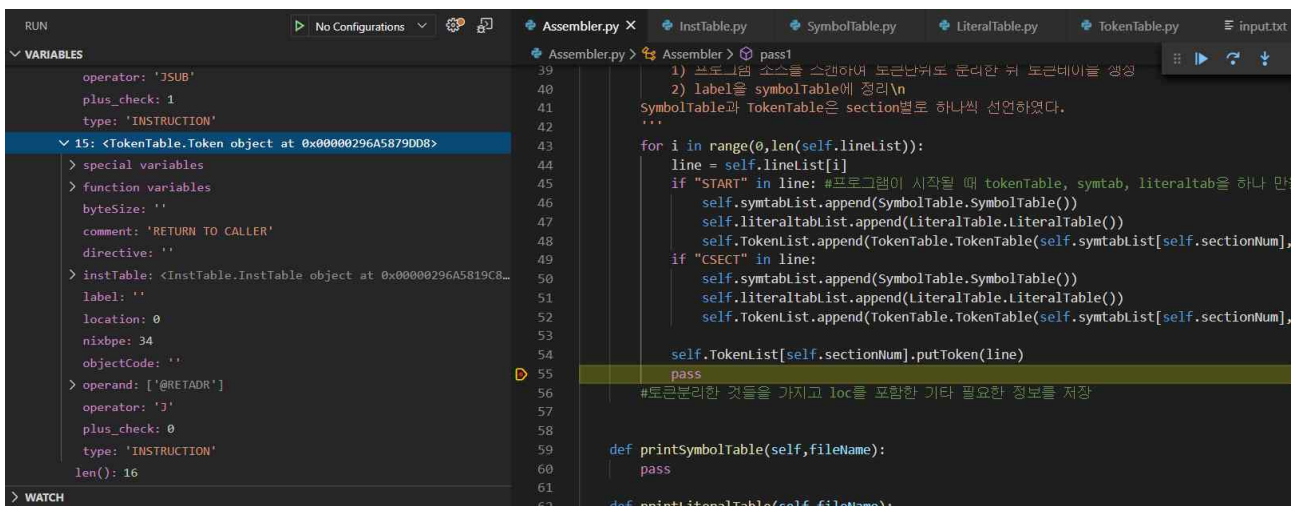


그림 8 소스 코드를 읽어와 토큰단위로 분리하여 토큰 테이블을 작성하는 과정 디버깅

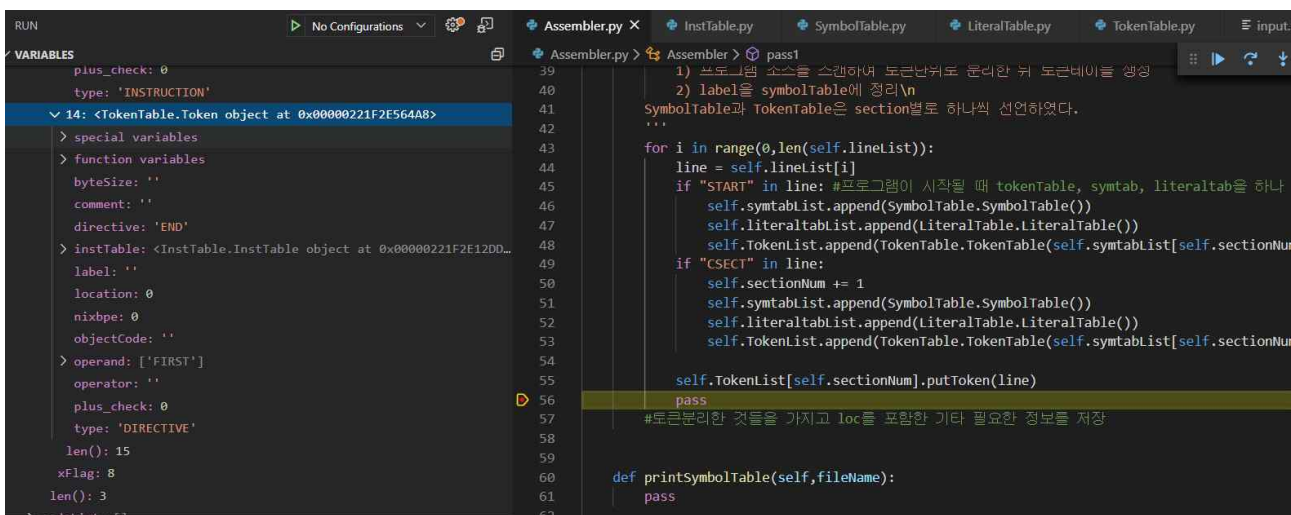


그림 9 소스 코드를 읽어와 토큰단위로 분리하여 토큰 테이블을 작성하는 과정 디버깅

The screenshot shows a debugger interface. On the left, the 'VARIABLES' pane lists several objects, including a 'TokenTable.Token object' and an 'InstTable.InstTable object'. The 'location: 10' property of the 'InstTable' object is highlighted. On the right, the 'Assembler.py' source code is visible, showing a function 'pass' at line 117.

그림 10 loc가 제대로 저장되었는지 디버깅

The screenshot shows a debugger interface. On the left, the 'VARIABLES' pane lists several objects, including a 'SymbolTable.SymbolTable object'. The 'symbolList' property of the 'SymbolTable' object is highlighted. On the right, the 'Assembler.py' source code is visible, showing a function 'pass' at line 117.

그림 11 symbol_table이 제대로 저장되었는지 디버깅

The screenshot shows a debugger interface. On the left, the 'VARIABLES' pane lists several objects, including a 'LiteralTable.LiteralTable object'. The 'literalList' property of the 'LiteralTable' object is highlighted. On the right, the 'Assembler.py' source code is visible, showing a function 'pass' at line 117.

그림 12 literal_table이 제대로 저장되었는지 디버깅

```

> 03: <TokenTable.Token object at 0x0000018905082008>
04: <TokenTable.Token object at 0x0000018905082198>
> special variables
> function variables
  bufferSize: 4
  comment: 'READ INPUT RECORD'
  directive: ''
> instTable: <InstTable.InstTable object at 0x0000018904FE...>
  label: 'CLOOP'
  location: 3
  nixbpe: 49
  objectCode: '4B100000'
> operand: ['RDREC']
  operator: 'JSUB'
  plus_check: 1
  type: 'INSTRUCTION'
05: <TokenTable.Token object at 0x0000018905082208>

118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
elif tmpToken.type == "INSTRUCTION":
    op = 252
    opcode = op & tmpToken.instTable.getOpcode(tmpToken.operator)
    opcode += int(tmpToken.getFlag(self.nFlag | self.iFlag)/16)

    if tmpToken.plus_check == 1: #명령어가 4형식일 때
        self.tokenList[index].setObjectCode(format(opcode,"02X")+format(tmpToken.get
        self.tokenList[index].setByteSize(4)
        if self.symTab.searchRefSymbol(tmpToken.operand[0]) == 1:
            self.symTab.modSymbol("+" + tmpToken.operand[0], tmpToken.location + 1,
        return

    if tmpToken.instTable.getFormat(tmpToken.operator) == 1: #1형식일 때
        self.tokenList[index].setObjectCode(format(opcode, "02X"))
        self.tokenList[index].setByteSize(1)
    elif tmpToken.instTable.getFormat(tmpToken.operator) == 2: #2형식일 때
        if tmpToken.instTable.getNumberOfOperand(tmpToken.operator) == 1: #피연산자의
            reg_num = 0
            for i in range(0,10):

```

그림 13 object code 만드는 과정 디버깅

```

154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
pass2 과정을 수행한다.\n
1) 분석된 내용을 바탕으로 object code를 생성하여 co
...
#토른분리한 것들을 가지고 각 라인별 object code 작성
for i in range(0, len(self.TokenList)):
    for j in range(0, len(self.TokenList[i].tokenList)):
        self.TokenList[i].makeObjectCode(j)

check = 1
line_max = 0
line_check = 0
line = 0
textTmp = ""
code = ""
num = 0
#프로그램의 object program을 작성하여 codeList에 저장
for i in range(0, len(self.TokenList)):
    notFinish = -1
    for j in range(0, len(self.TokenList[i].tokenList)):
        tmpToken = self.TokenList[i].getToken(j)

#Header record 작성

```

그림 14 object code 만드는 과정 디버깅

```

> VARIABLES
> Locals
  check: 0
  code: '454F46'
  i: 0
  j: 21
  k: 1
  lastAddr: 4147
  line: 39
  line_check: 1
  line_max: 0
  name: 'COPY'
  notFinish: 1
  num: 3
> self: <__main__.Assembler object at 0x0000026C4EFA0B38>
  startAddr: 0
  textTmp: 'T00003003454F46'
  tmp: 'RDREC WRREC '
> tmpToken: <TokenTable.Token object at 0x0000026C4F061AC8>
> Globals
> WATCH

Assembler.py > Assembler > pass2
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
notFinish = 1
textTmp += format(num, "02X") + code
self.codeList.append(textTmp)
line += num
textTmp = ""
code = ""
num = 0
line = 0
check = 1

if line_check == 1:
    textTmp += format(num, "02X") + code
    self.codeList.append(textTmp)
    textTmp = ""
    code = ""
    num = 0
    line = 0
    check = 1

#Modification record 작성
if len(self.symtabList[i].refList) != 0:
    for k in range(0, len(self.symtabList[i].modList)):
        tmp = "M" + format(self.symtabList[i].modLocationList[k],
        self.codeList.append(tmp)

#End record 작성

```

그림 15 object program 만드는 과정 디버깅

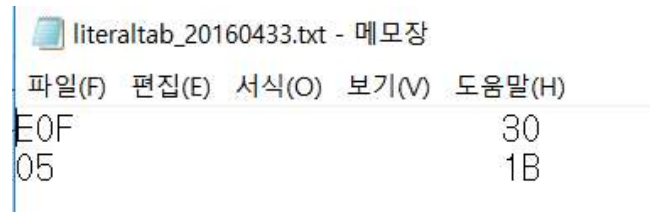
4.5 수행 결과

1) symtab_20160433.txt의 출력결과



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

2) literal_20160433.txt의 출력결과



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

3) output_20160433.txt의 출력결과



```
HCOPY 000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
T0000001D1720274B1000000320232900003320074B1000003F2FEC0320160F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000

HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER
E

HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB41077100000E32012332FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E
```

5장 기대효과 및 결론

-이번 프로젝트는 PYTHON를 이용하여 주어진 input파일을 파싱하여 토큰화하고 기본적인 SIC/XE 머신의 어셈블러를 구현해보는 프로젝트였다. 이전 프로젝트의 언어였던 JAVA와 마찬가지로 PYTHON 또한 익숙치 않은 언어였기 때문에 몇몇 문법과 관련하여 어려운 점들도 있었지만 C와 JAVA로 한번씩 구현해 본 프로젝트였기 때문에 좀 더 수월하게 프로젝트를 진행할 수 있었다. 지난번 프로젝트와 마찬가지로 이번 프로젝트에서도 input파일을 어떻게 토큰화할지, 토큰분리한 토큰들을 어떻게 활용하여 object code를 작성할지가 중요했다. 다행히 JAVA로 한번 구현했었던 로직이라 JAVA 프로젝트를 참고하니 PYTHON으로 구현하는 데에는 큰 어려움이 없었다.

이번 프로젝트에서도 COPY프로그램을 변환하는 것이 목표였기 때문에 모든 문법들이나 모든 기능들이 포함되어 있는 어셈블러를 구현하지는 못했다는 것이 조금 아쉽기는 하지만 이번에 해보았던 프로젝트들 중 하나의 프로젝트만 보완하여도 나머지 프로젝트를 완성하는데에는 큰 어려움이 없을 것 같다. 추후에 여러가지를 보완하여 더 나은 어셈블러를 구현해볼 것이다. 마찬가지로 이번 프로젝트 또한 assembler에 대한 이해를 높이고 객체지향 프로그래밍과 PYTHON언어에 대해서도 좀 더 익숙해질 수 있었던 프로젝트였다.

첨부 프로그램 소스파일

1) InstTable.py

```
class InstTable:
    def __init__(self, instFile):
        '''
        클래스를 초기화하는 함수로 명령어에 대한 정보를 파싱하는 과정을 동시에 처리한다.
        명령어에 대한 정보는 변수 instMap에 저장한다.
        instFile : 명령어에 대한 정보를 저장하고 있는 파일의 이름
        '''
        self.instMap = dict() #명령어에 대한 정보를 저장
        self.directive = {"START" : 1,"END" : 1,"BYTE" : 1,"WORD" : 1,"RESB" : 1,"RESW" : 1,"CSECT" : 0,"EXTDEF" : 1,"EXTREF" : 1,"EQU" : 1,"ORG" : 1,"LTORG" : 0} #지시어에 대한 정보를 저장
        self.openFile(instFile)

    def openFile(self,fileName):
        '''
        전달받은 이름의 파일을 열고 파일 안에 저장되어 있던 내용을 파싱하여 instMap에 저장한다.
        instFile : 명령어에 대한 정보를 저장하고 있는 파일의 이름
        '''
        file = open(fileName,'r') #파일 열기
        while True:
            line = file.readline() #파일을 한줄씩 읽어들이
            if not line: #더 이상 읽어들이 라인이 존재하지 않으면
```

while문 탈출

break

token = Instruction(line) #읽어들인 라인의 정보를 토큰분리한 값을 저장

self.instMap[token.instruction] = token #instruction 이름을 key로 저장하고 해당하는 instruction에 대한 정보를 value로 저장

file.close() #파일 닫기

def getOpcode(self,instruction):

'''

인자로 받은 명령어의 opcode를 찾아서 리턴하는 함수이다.

instruction : opcode를 알고 싶은 명령어

return : 해당 명령어의 opcode

'''

info = self.instMap.get(instruction) #주어진 문자열이 key값으로 존재한다면 해당하는 키값의 value를 리턴

if info == 'None': # 주어진 문자열이 key값으로 존재하지 않을 때 -1리턴

return -1

else : # 주어진 문자열이 key값으로 존재할 때 해당하는 키값의 value의 opcode를 리턴

return info.opcode

def getNumberOfOperand(self,instruction):

'''

인자로 받은 명령어의 피연산자 개수를 찾아서 리턴하는 함수이다.

instruction : 피연산자 개수를 알고 싶은 명령어

return : 해당 명령어의 피연산자 개수

```

'''
    info = self.instMap.get(instruction) #주어진 문자열이 key값으로 존재한다면 해당하는 키값의 value를 리턴
    if info == 'None': # 주어진 문자열이 key값으로 존재하지 않을 때 -1리턴
        return -1
    else : # 주어진 문자열이 key값으로 존재 할 때 해당하는 키값의 value의 opcode를 리턴
        return info.numberOfOperand
def getFormat(self,instruction):
'''
    인자로 받은 명령어의 형식을 찾아서 리턴하는 함수이다.
    instruction : 형식을 알고 싶은 명령어
    return : 해당 명령어의 형식
'''

    info = self.instMap.get(instruction) #주어진 문자열이 key값으로 존재한다면 해당하는 키값의 value를 리턴
    if info == 'None': # 주어진 문자열이 key값으로 존재하지 않을 때 -1리턴
        return -1
    else : # 주어진 문자열이 key값으로 존재 할 때 해당하는 키값의 value의 opcode를 리턴
        return info.format
def isInstruction(self,str):
'''
    입력문자열이 명령어인지 검사하는 함수이다.
    str : 명령어인지 검사하고 싶은 문자열
    return : 명령어가 맞다면 1, 아니면 -1을 리턴
'''

```

```

'''
    if str in self.instMap: #주어진 문자열이 key값으로 존재한다면 1 리턴
        return 1
    else : #주어진 문자열이 key값으로 존재하지 않는다면 -1 리턴
        return -1
def isDirective(self,str):
'''
    입력문자열이 지시어인지 검사하는 함수이다.
    str : 지시어인지 검사하고 싶은 문자열
    return : 지시어가 맞다면 지시어 뒤에 몇 개의 정보가 포함되어 있는지(0,1)를 리턴하고 아니라면 -1을 리턴
'''

    if str in self.directive: #주어진 문자열이 key값으로 존재한다면 1 리턴
        return self.directive[str]
    else : #주어진 문자열이 key값으로 존재하지 않는다면 -1 리턴
        return -1

class Instruction:
    def __init__(self, line):
        '''
            클래스를 선언 및 초기화하면서 동시에 명령어의 구조에 맞게 파싱하여 정보를 저장한다.
            line : 명령어의 정보가 포함된 파일로부터 한 줄씩 읽어들이는 문자열
        '''
        self.instruction = "" #명령어의 이름

```

```

        self.opcode = 0 #명령어의 opcode 값
        self.numberOfOperand = 0 # 명령어가 몇개의 피연산자를 가
지고 있는지
        self.format = 0 #명령어의 형식
        self.parsing(line)
def parsing(self,line):
    """
    전달된 문자열을 파싱하여 instruction에 대한 정보를 저장한다.
    line : 명령어의 정보가 포함된 파일로부터 한 줄씩 읽어들이는 문
자열
    """
    token = line.split()
    self.instruction = token[0]
    self.numberOfOperand = int(token[1])
    self.format = int(token[2])
    self.opcode = int(token[3],16)

```

```

2) SymbolTable.py
class SymbolTable:
    def __init__(self):
        """
        클래스를 초기화한다
        """
        self.symbolList = []
        self.locationList = []
        self.modList = []
        self.modLocationList = []
        self.modSize = []
        self.defList = []
        self.refList = []
    def putSymbol(self, symbol, location):
        """
        새로운 symbol을 table에 추가하는 함수이다. 중복된 symbol은
putSymbol로 입력될 수 없다. symbol의 주소를 변경하려면
modifySymbol()를 이용한다.
        symbol : 새로 추가되는 symbol의 label\n
        location : 해당 symbol의 주소값
        """
        if not symbol in self.symbolList:
            self.symbolList.append(symbol)
            self.locationList.append(location)
    def modifySymbol(self, symbol, newLocation):
        """
        기존에 존재하는 symbol값에 대하여 가리키는 주소값을 변경한
다.

```



```

symbol : 변경을 원하는 symbol의 label\n
newLocation : 새로 바꾸고자 하는 주소값
'''

if symbol in self.symbolList:
    index = self.symbolList.index(symbol)
    self.locationList[index] = newLocation
def modSymbol(self, symbol, location, size):
    '''
    modification record에 작성할 symbol들을 table에 추가한다.

    symbol : modify될 symbol의 label\n
    location : 해당 symbol의 location\n
    size : modify될 바이트의 크기
    '''
    self.modList.append(symbol)
    self.modLocationList.append(location)
    self.modSize.append(size)
def setDefSymbol(self, symbol):
    '''
    지시어 EXTDEF가 나왔을 때 symbol을 저장

    symbol : 새로 추가되는 symbol의 label
    '''
    self.defList.append(symbol)
def setRefSymbol(self, symbol):
    '''
    지시어 EXTREF가 나왔을 때 symbol을 저장

```

```

symbol : 새로 추가되는 symbol의 label
'''

self.refList.append(symbol)
def searchRefSymbol(self, symbol):
    '''
    주어진 symbol이 reference된 symbol인지 확인
    symbol : 확인하고자 하는 symbol\n
    return : reference symbol이면 1, 아니면 -1 반환
    '''

    if symbol in self.refList:
        return 1
    else:
        return -1
def search(self, symbol):
    '''
    인자로 전달된 symbol이 어떤 주소를 가리키는지 알려준다.
    symbol : 검색을 원하는 symbol의 label\n
    symbol이 가지고 있는 주소값. 해당 symbol이 없을 경우 -1 리
    턴

    '''
    if symbol in self.symbolList:
        index = self.symbolList.index(symbol)
        return self.locationList[index]
    else:
        return -1
def getSymbol(self, index):
    '''
    주어진 index에 해당하는 symbol을 리턴한다.

```

```

index : symbol을 검색할 인덱스\n
return : index에 해당하는 symbol
'''

return self.symbolList[index]
def getAddress(self, index):
'''
    index를 이용하여 symbol을 찾은 후 그 symbol에 해당하는
address를 리턴한다.
    index : address을 검색할 인덱스\n
    return : address
'''

return self.locationList[index]
def getSize(self):
'''
    symboltable의 크기를 리턴한다.
    return : symboltable의 크기
'''

return len(self.symbolList)

```

3) LiteralTable.py

```

class LiteralTable:
    def __init__(self):
        '''
        클래스를 초기화한다.
        '''

        self.literalList = []
        self.locationList = []

    def putLiteral(self, literal, location):
        '''
        새로운 Literal을 table에 추가하는 함수이다. 중복된 literal은
putLiteral로 입력될 수 없다. literal의 주소를 변경하려면 modifyLital()
를 이용한다.
        literal : 새로 추가되는 literal의 label\n
        location : 해당 literal의 주소값
        '''

        tmpliteral = literal[1:]
        if not tmpliteral in self.literalList:
            self.literalList.append(tmpliteral)
            self.locationList.append(location)

    def modifyLiteral(self, literal, newLocation):
        '''
        기존에 존재하는 literal 값에 대해서 해당 literal의 주소값을 변
경한다.
        literal : 주소값의 변경을 원하는 literal의 label\n
        newLocation : 새로 바꾸고자 하는 주소값
        '''

```

```

        if literal in self.literalList:
            index = self.literalList.index(literal)
            self.locationList[index] = newLocation
def search(self, literal):
    """
    인자로 전달된 literal이 어떤 주소를 지칭하는지 알려준다.
    literal : 검색을 원하는 literal의 label\n
    return : literal이 가지고 있는 주소값. 해당 literal이 없을 경우
-1 리턴
    """
    if literal in self.literalList:
        index = self.literalList.index(literal)
        return self.locationList[index]
    else:
        return -1
def getAddress(self, index):
    """
    주어진 index에 해당하는 address을 리턴한다.

    index : address을 검색할 인덱스\n
    return : index에 해당하는 address
    """
    return self.locationList[index]
def getLiteral(self, index):
    """
    주어진 index에 해당하는 literal을 리턴한다.

    index : literal을 검색할 인덱스\n

```

```

        return : index에 해당하는 literal
    """
    return self.literalList[index]

def addrLiteralTab(self, locctr):
    """
    literal들의 주소를 넣어주는 함수이다.
    locctr : 현재 프로그램의 주소\n
    return : literal들의 주소를 넣어주고 증가한 주소를 리턴
    """
    for i in range(0, len(self.literalList)):
        if -2 in self.locationList:
            self.modifyLiteral(self.literalList[i], locctr)
            if self.literalList[i][0] == 'C' or self.literalList[i][0] =
= 'c': # literal이 'C'인지 확인
                tmpStr = self.literalList[i]
                tmpStr = tmpStr[2:len(tmpStr)-1]
                locctr += len(tmpStr) #char개수만큼 locctr 증가
            elif self.literalList[i][0] == 'X' or self.literalList[i][0]
== 'x': #literal이 'X'인지 확인
                tmpStr = self.literalList[i]
                tmpStr = tmpStr[2:len(tmpStr)-1]
                locctr += int(len(tmpStr)/2) #char개수/2만큼
locctr 증가
            return locctr
def getSize(self):
    """

```

```

literalTable의 크기를 알려주는 함수이다.
return : literal table의 크기
'''
return len(self.literalList)

```

```

4) TokenTable.py
import re
class TokenTable:
    def __init__(self, symTab, literalTab, instTab):
        '''
        클래스를 초기화시키는 함수로 pass2 과정에서 object code로
        변환하기 위해 symbolTable과 instTable과 instTable을 링크시킨다.
        symTab : 해당 section과 연결되어있는 symbol table\n
        literalTab : 해당 section과 연결되어있는 literal table\n
        instTab : instruction 명세가 정의된 instTable\n
        '''

        #bit 조작의 가독성을 위해 static 변수 선언
        TokenTable.nFlag = 32
        TokenTable.iFlag = 16
        TokenTable.xFlag = 8
        TokenTable.bFlag = 4
        TokenTable.pFlag = 2
        TokenTable.eFlag = 1

        #토큰을 다룰 때 필요한 테이블들을 링크시킨다.
        self.symTab = symTab
        self.literalTab = literalTab
        self.instTab = instTab

        #각 line을 의미별로 분할하고 분석하는 공간
        self.tokenList = list()

        #각 프로그램이 끝날때마다 마지막 주소값을 저장
        self.lastAddr = 0

    def putToken(self, line):
        '''

```

일반 문자열을 받아서 Token단위로 분리시켜 tokenList에 추가하는 함수이다.

```
line : 분리되지 않은 일반 문자열
'''

self.tokenList.append(Token(line, self.instTab))
def getToken(self, index):
'''
tokenList에서 index에 해당하는 Token을 리턴한다.
index : token을 검색할 index\n
return : index번호에 해당하는 코드를 분석한 Token 클래스
'''
return self.tokenList[index]
def makeObjectCode(self, index):
'''
```

P a s s 2 과정에서 사용하는 함수로 instruction table, symbol table literal table 등을 참조하여 objectcode를 생성하고, 이를 저장한다.

```
index : object 코드를 생성하고자하는 토큰 인스턴스의 인덱스
'''
register = ["A","X","L","B","S","T","F","","PC","SW"]
tmpToken = self.tokenList[index]
if tmpToken.type == "DIRECTIVE":
    if tmpToken.directive == "LTOrg" : #프로그램에 나왔던
literal값들을 메모리에 저장
        for i in range(0,self.literalTab.getSize()):
            if self.literalTab.getLiteral(i)[0] == 'X': #literal이
'X'로 시작하면 literal값을 그대로 메모리에 할당
```

```
tmp = self.literalTab.getLiteral(i)
tmp = tmp[2:len(tmp)-1]
self.tokenList[index].setObjectCode(tmp)
self.tokenList[index].setByteSize(1)
else : #literal이 'C'로 시작하면 각 char 값의
ASCII code를 메모리에 할당
    tmp = self.literalTab.getLiteral(i)
    tmp = tmp[2:len(tmp)-1]
    tmpStr = ""
    for j in range(0,len(tmp)):
        tmpStr += format(ord(tmp[j]),"02X")
    self.tokenList[index].setObjectCode(tmpStr)
    self.tokenList[index].setByteSize(len(tmp))

elif tmpToken.directive == "END": #프로그램에 나왔던
literal값들을 메모리에 저장
    for i in range(0,self.literalTab.getSize()):
        if self.literalTab.getLiteral(i)[0] == 'X': #literal이
'X'로 시작하면 literal값을 그대로 메모리에 할당
            tmp = self.literalTab.getLiteral(i)
            tmp = tmp[2:len(tmp)-1]
            self.tokenList[index].setObjectCode(tmp)
            self.tokenList[index].setByteSize(1)
        else : #literal이 'C'로 시작하면 각 char 값의
ASCII code를 메모리에 할당
            tmp = self.literalTab.getLiteral(i)
            tmp = tmp[2:len(tmp)-1]
            tmpStr = ""
```

```

        for j in range(0, len(tmp)):
            tmpStr += format(ord(tmp[j]), "02X")
        self.tokenList[index].setObjectCode(tmpStr)
        self.tokenList[index].setByteSize(len(tmp))

    elif tmpToken.directive == "BYTE":
        tmp = tmpToken.operand[0]
        tmp = tmp[2:len(tmp)-1]
        if tmpToken.operand[0][0] == 'X' or tmpToken.operand[0][0] == 'x':
            self.tokenList[index].setObjectCode(tmp)
            self.tokenList[index].setByteSize(1)
        elif tmpToken.operand[0][0] == 'C' or tmpToken.operand[0][0] == 'c':
            tmpStr = ""
            for j in range(0, len(tmp)):
                tmpStr += format(ord(tmp[j]), "02X")
            self.tokenList[index].setObjectCode(tmpStr)
            self.tokenList[index].setByteSize(len(tmp))

    elif tmpToken.directive == "WORD":
        if tmpToken.operand[0][0].isdigit(): #숫자라면
            self.tokenList[index].setObjectCode(format(int(tmpToken.operand[0], 16), "s"))
            self.tokenList[index].setByteSize(1)
        else : #문자열일때의 처리
            tmpToken.operand = tmpToken.operand[0].split('
-')

```

```

        tmp1 = tmpToken.operand[0]
        tmp2 = tmpToken.operand[1]
        if self.symTab.searchRefSymbol(tmp1) == 1:
            self.symTab.modSymbol("+ "+ tmp1, tmpToken.location, 6)

        if self.symTab.searchRefSymbol(tmp2) == 1:
            self.symTab.modSymbol("- "+tmp2, tmpToken.location, 6)

        self.tokenList[index].setObjectCode(format(0, "06X"))

        self.tokenList[index].setByteSize(3)

    elif tmpToken.type == "INSTRUCTION":
        op = 252
        opcode = op & tmpToken.instTable.getOpcode(tmpToken.operator)
        opcode += int(tmpToken.getFlag(self.nFlag | self.iFlag)/16)

        if tmpToken.plus_check == 1: #명령어가 4형식일 때
            self.tokenList[index].setObjectCode(format(opcode, "02X")+format(tmpToken.getFlag(15), "01X")+format(0, "05X"))
            self.tokenList[index].setByteSize(4)
            if self.symTab.searchRefSymbol(tmpToken.operand[0]) == 1 :
                self.symTab.modSymbol("+ " + tmpToken.operand[0], tmpToken.location + 1, 5)

        return

```

```

        if tmpToken.instTable.getFormat(tmpToken.operator) == 1
: #1형식일 때
            self.tokenList[index].setObjectCode(format(opcode, "
02X"))

            self.tokenList[index].setByteSize(1)
        elif tmpToken.instTable.getFormat(tmpToken.operator) ==
2 : #2형식일 때
            if tmpToken.instTable.getNumberOfOperand(tmpToken
.operator) == 1: #피연산자의 개수가 1개일 때
                reg_num = 0
                for i in range(0,10):
                    if tmpToken.operand[0] == register[i]:
                        reg_num = i
                        break;
                self.tokenList[index].setObjectCode(format(opcode
,"02X")+format(reg_num,"01X")+format(0,"01X"))
                self.tokenList[index].setByteSize(2)
            elif tmpToken.instTable.getNumberOfOperand(tmpToken
n.operator) == 2: #피연산자의 개수가 2개일 때
                reg_num1 = 0
                reg_num2 = 0
                for i in range(0,10):
                    if tmpToken.operand[0] == register[i]:
                        reg_num1 = i
                    if tmpToken.operand[1] == register[i]:
                        reg_num2 = i
                self.tokenList[index].setObjectCode(format(opcode
,"02X")+format(reg_num1,"01X")+format(reg_num2,"01X"))

```

```

            self.tokenList[index].setByteSize(2)
        elif tmpToken.instTable.getFormat(tmpToken.operator) ==
3 : #3형식일 때
            if tmpToken.instTable.getNumberOfOperand(tmpToken
.operator) == 0 : #피연산자가 존재하지 않을 때
                self.tokenList[index].setObjectCode(format(opcode
,"02X")+format(0,"04X"))
                self.tokenList[index].setByteSize(3)
                return
            else : #피연산자가 존재할 때
                if tmpToken.operand[0][0] == '#' : #immediate a
ddressing일 때
                    tmpToken.operand[0] = tmpToken.operand[0
][1:]
                    self.tokenList[index].setObjectCode(format(op
code, "02X")+format(int(tmpToken.operand[0]),"04X"))
                    self.tokenList[index].setByteSize(3)
                    return
                if tmpToken.operand[0][0] == '@' : #indirection
addressing일 때
                    tmpToken.operand[0] = tmpToken.operand[0
][1:]
                    target = 0
                    if tmpToken.operand[0][0] == '=' : #피연산자가
literal일 때
                        tmpToken.operand[0] = tmpToken.operand[0
][1:]
                        target = self.literalTab.search(tmpToken.oper

```

```

and[0])
        else :
            target = self.symTab.search(tmpToken.opera
nd[0])

            pc = self.tokenList[index+1].location
            addr = target - pc
            self.tokenList[index].setObjectCode(format(opcod
e, "02X") + format(tmpToken.getFlag(15),"01X") + format(addr & 0xFF, "03X"))
            self.tokenList[index].setByteSize(3)

def getObjectCode(self, index):
    """
    index번호에 해당하는 object code를 리턴한다.

    index : objectCode를 검색할 index\n
    return : object code
    """
    return self.tokenList[index].objectCode
class Token:
    def __init__(self, line, instTable):
        """
        클래스를 초기화하면서 바로 line의 의미분석을 수행한다.
        line = line 문장단위로 저장된 프로그램 코드\n
        instTable = 토큰을 파싱하기 위해 instruction table 링크
        """
        #의미 분석 단계에서 사용되는 변수들

```

```

        self.location = 0
        self.label = ""
        self.operator = ""
        self.operand = []
        self.comment = ""
        self.nixbpe = 0
        self.plus_check = 0 #operand에 +가 포함되어 있는지에 대한
정보를 저장
        self.directive = "" #명령어 라인 중 지시어
        self.type = "" #해당 명령어 라인이 어떤 타입인지 명시

        #object code 생성 단계에서 사용되는 변수들
        self.objectCode = ""
        self.byteSize = 0

        #instruction에 대한 정보를 참고하기 위해 링크시킨
instruction table
        self.instTable = instTable
        self.parsing(line)
    def parsing(self, line):
        """
        line의 실질적인 분석을 수행하는 함수. Token의 각 변수에 분석
한 결과를 저장한다.
        line : 문장단위로 저장된 프로그램 코드.
        """
        token = re.split('\t|\n',line.strip())

        if token[0] == '.':

```



```

        self.comment = line
        self.type = "COMMENT"
        return

#명령어에 +가 추가되어 있으면 +를 제거한 후 다시 저장
if '+' in token[0]:
    self.plus_check = 1
    tmpInst = token[0][1:]
    token[0] = tmpInst
elif len(token) >= 2:
    if '+' in token[1]:
        self.plus_check = 1
        tmpInst = token[1][1:]
        token[1] = tmpInst
    if self.instTable.isInstruction(token[0]) == 1: #첫번째로 잘린
        문자열이 명령어라면
            self.operator = token[0]
            if self.instTable.getFormat(self.operator) == 3:
                if self.plus_check == 1:
                    self.setFlag(TokenTable.eFlag, 1) #4형식 명령어
                    #4형식 명령어
                    #명령어에 +가 추가되어 있지 않다면
                    pc relative를 사용할 것이므로 p를 표시
                    self.setFlag(TokenTable.pFlag, 1)

            numOfOperand = self.instTable.getNumberOfOperand(self.
operator) #operand가 있는지 확인
            if numOfOperand != 0: #operand가 있다면

```

```

        if numOfOperand == 1: #operand의 개수가 1개라면
            self.operand.append(token[1])

            if ',' in token[1]: #BUFFER,x 같이 배열을 쓰는
                경우에 토큰분리
                    self.operand = token[1].split(',')
                    if self.operand[1] == "X":
                        self.setFlag(TokenTable.xFlag, 1)

        elif numOfOperand == 2: #operand의 개수가 2개라면
            self.operand = token[1].split(',')
            #addressing방식이
            if self.operand[0][0] == '@': #indirection addressing
                일 때
                    self.setFlag(TokenTable.nFlag, 1)
                elif self.operand[0][0] == '#': #immediate addressin
                    g일 때
                        self.setFlag(TokenTable.iFlag, 1)

            else: #둘 다 아니면
                if self.instTable.getFormat(token[0]) == 3: #그런
                    데 명령어의 형식이 3형식/4형식이라면
                        self.setFlag(TokenTable.nFlag, 1)
                        self.setFlag(TokenTable.iFlag, 1)

            else: #operand가 없다면
                if self.instTable.getFormat(token[0]) == 3: #그런데
                    명령어의 형식이 3형식이라면

```

```

        self.setFlag(TokenTable.nFlag, 1)
        self.setFlag(TokenTable.iFlag, 1)
        if len(token) == 2: #코멘트가 존재한다면 저장
            self.comment = token[1]

        if len(token) == 3: #코멘트가 존재한다면 저장
            self.comment = token[2]

        self.type = "INSTRUCTION"
        elif self.instTable.isDirective(token[0]) >= 0: #첫번째로 잘린
문자열이 지시어라면
            self.directive = token[0]
            numOfOperand = self.instTable.isDirective(token[0])
            if numOfOperand == 1:
                if ',' in token[1]:
                    self.operand = token[1].split(',')
                else:
                    self.operand.append(token[1])
            if len(token) == 3:
                self.comment = token[2]
            else :
                if len(token) == 2:
                    self.comment = token[1]
            self.type = "DIRECTIVE"
        else : #첫번째로 잘린 문자열이 label이라면
            self.label = token[0]
            if self.instTable.isInstruction(token[1]) == 1: #두번째로
잘린 문자열이 명령어라면

```

```

        self.operator = token[1]
        if self.instTable.getFormat(self.operator) == 3:
            if self.plus_check == 1:
                self.setFlag(TokenTable.eFlag, 1) #4형식 명
령어이므로 e를 표시
            else : #명령어에 +가 추가되어 있지 않다면
pc relative를 사용할 것이므로 p를 표시
                self.setFlag(TokenTable.pFlag, 1)

            numOfOperand = self.instTable.getNumberOfOperand(
self.operator) #operand가 있는지 확인
            if numOfOperand != 0: #operand가 있다면
                if numOfOperand == 1: #operand의 개수가 1개라
면
                    self.operand.append(token[2])

                if ',' in token[2]: #BUFFER,x 같이 배열을 쓰
는 경우에 토큰분리
                    self.operand = token[2].split(',')
                    if self.operand[1] == "X":
                        self.setFlag(TokenTable.xFlag, 1)

            elif numOfOperand == 2: #operand의 개수가 2개
라면
                self.operand = token[2].split(',')
            #addressing방식이
            if self.operand[0][0] == '@': #indirection address
sing일 때

```

```

        self.setFlag(TokenTable.nFlag,1)
        elif self.operand[0][0] == '#': #immediate addre
            self.setFlag(TokenTable.iFlag,1)

        else: #둘 다 아니라면
            if self.instTable.getFormat(token[1]) == 3: #
                self.setFlag(TokenTable.nFlag,1)
                self.setFlag(TokenTable.iFlag,1)

            else: #operand가 없다면
                if self.instTable.getFormat(token[1]) == 3: #그런데 명령어의 형식이 3형식/4형식이라면
                    self.setFlag(TokenTable.nFlag,1)
                    self.setFlag(TokenTable.iFlag,1)

                if len(token) == 3: #코멘트가 존재한다면 저장
                    self.commnet = token[2]

                if len(token) == 4: #코멘트가 존재한다면 저장
                    self.comment = token[3]

        self.type = "INSTRUCTION"
        elif self.instTable.isDirective(token[1]) >= 0: #첫번째로 잘린 문자열이 지시어라면
            self.directive = token[1]
            numOfOperand = self.instTable.isDirective(token[1])

```

```

        if numOfOperand == 1:
            if ',' in token[2]:
                self.operand = token[1].split(',')
            else:
                self.operand.append(token[2])
            if len(token) == 4:
                self.comment = token[3]
        else :
            if len(token) == 3:
                self.comment = token[2]
            self.type = "DIRECTIVE"

def setFlag(self, flag, value):
    '''
    n,i,x,b,p,e flag를 설정한다.

    flag : 원하는 비트 위치
    value : 집어넣고자 하는 값. 1을 넣으면 추가, 0을 넣으면 삭제
    '''
    if value == 1 :
        self.nixbpe |= flag
    else :
        self.nixbpe ^= flag

def getFlag(self, flags):
    '''
    원하는 flag들의 값을 얻어오고자 하는 함수이다. flag의 조합을 통해 동시에 여러개의 플래그를 얻는 것 역시 가능하다

```

```

        flags : 값을 확인하고자 하는 비트 위치\n
        return : 비트위치에 들어가 있는 값. 플래그별로 각각
32, 16, 8, 4, 2, 1의 값을 리턴한다
'''

return self.nixbpe & flags
def setLocation(self, loc):
'''
loc 값을 인자로 받아와 저장하는 함수이다.
loc : 저장하고자 하는 location
'''
self.location = loc
def setObjectCode(self, objectCode):
'''
object code을 인자로 받아와 저장한다.
objcode : 저장하고자하는 object code
'''
self.objectCode = objectCode
def setByteSize(self, byteSize):
'''
의미분석이 끝난 후 pass2에서 object code로 변형시켰을 때의
ByteSize를 인자로 받아와 저장시키기 위한 함수이다.
byteSize : 저장하고자하는 byte size
'''
self.byteSize = byteSize

```

5) Assembler.py

```

import InstTable
import LiteralTable
import SymbolTable
import TokenTable
class Assembler:
    def __init__(self,instFile):
        '''
클래스를 초기화하는 함수로 instruction table을 동시에 세팅한
다.
instFile : instruction에 대한 정보를 가지고 있는 파일 이름
'''
self.instTable = InstTable.InstTable(instFile) #명령어에 대한 정
보를 저장
self.lineList = list() #읽어들인 input 파일의 내용을 한줄씩 저장
self.symtabList = list() #프로그램의 section별로 symboltable을
저장
self.literalList = list() #프로그램의 section별로 literaltable을
저장
self.TokenList = list() #프로그램이 section별로 프로그램을 저
장
self.codeList = list() #object code를 object program형식에 맞
춰 작성한 후 저장
self.locctr = 0 #주소
self.sectionNum = 0 #프로그램이 총 몇개의 section으로 구성
되어 있는지 저장
def loadInputFile(self,inputFile):
'''

```

```

input파일을 읽어들이 lineList에 저장한다.
inputFile : input파일의 이름
'''
file = open(inputFile,'r') #파일 열기
while True:
    line = file.readline() #파일을 한줄씩 읽어들이
    if not line: #더 이상 읽어들이 라인이 존재하지 않으면
while문 탈출
        break
    self.lineList.append(line) #읽어들인 라인을 lineList에 저장
file.close() #파일 닫기
def pass1(self):
    '''
    pass1 과정을 수행하는 함수이다.\n
    1) 프로그램 소스를 스캔하여 토큰단위로 분리한 뒤 토큰테
이블 생성
    2) label을 symbolTable에 정리\n
    SymbolTable과 TokenTable은 section별로 하나씩 선언하였다.
    '''
    for i in range(0,len(self.lineList)):
        line = self.lineList[i]
        if "START" in line: #프로그램이 시작될 때
tokenTable, symtab, literalTab을 하나 만들어준다.
            self.symtabList.append(SymbolTable.SymbolTable())
            self.literalTabList.append(LiteralTable.LiteralTable())
            self.TokenList.append(TokenTable.TokenTable(self.sy
mtabList[self.sectionNum], self.literalTabList[self.sectionNum], self.inst
Table))

```

```

if "CSECT" in line:
    self.sectionNum += 1
    self.symtabList.append(SymbolTable.SymbolTable())
    self.literalTabList.append(LiteralTable.LiteralTable())
    self.TokenList.append(TokenTable.TokenTable(self.sy
mtabList[self.sectionNum], self.literalTabList[self.sectionNum], self.inst
Table))

self.TokenList[self.sectionNum].putToken(line)
#토큰분리한 것들을 가지고 loc를 포함한 기타 필요한 정보를
저장
for i in range(0,self.sectionNum + 1):
    for j in range(0,len(self.TokenList[i].tokenList)):
        tmpToken = self.TokenList[i].getToken(j)
        if tmpToken.type == "DIRECTIVE":
            if tmpToken.directive == "START": #프로그램이
시작될 때
                self.locctr = int(tmpToken.operand[0])
                self.TokenList[i].tokenList[j].setLocation(self.
locctr)

                self.symtabList[i].putSymbol(tmpToken.label,
self.locctr)

                continue
            elif tmpToken.directive == "CSECT": #sub프로그
램이 시작할 때
                self.TokenList[i-1].lastAddr = self.locctr
                self.locctr = 0
                self.TokenList[i].tokenList[j].setLocation(self.
locctr)

```

```

        self.symtabList[i].putSymbol(tmpToken.label,
self.locctr)
        continue
    elif tmpToken.directive == "EXTDEF": #EXTDEF
뒤에 나오는 값들 저장
        for k in range(0, len(tmpToken.operand)):
            self.symtabList[i].setDefSymbol(tmpToken.operand[k])
    elif tmpToken.directive == "EXTREF": #EXTREF
뒤에 나오는 값들 저장
        for k in range(0, len(tmpToken.operand)):
            self.symtabList[i].setRefSymbol(tmpToken.operand[k])

    if tmpToken.label != "":
        self.symtabList[i].putSymbol(tmpToken.label, self.locctr)

    self.TokenList[i].tokenList[j].setLocation(self.locctr)
    if tmpToken.type == "INSTRUCTION":
        if tmpToken.plus_check == 1 : #명령어의 형식에
            맞춰 locctr값 증가
            self.locctr += 4
        else :
            self.locctr += tmpToken.instTable.getFormat(tmpToken.operator)

    if len(tmpToken.operand) != 0 :
        if '=' in tmpToken.operand[0] : #indirection
            addressing일 때

```

```

        self.literalList[i].putLiteral(tmpToken.operand[0], -2)
    elif tmpToken.type == "DIRECTIVE":
        if tmpToken.directive == "WORD": #locctr을 3 증
            가시킴
            self.locctr += 3
        elif tmpToken.directive == "RESW": #locctr을
            (3 * 피연산자값)만큼 증가시킴
            self.locctr += 3 * int(tmpToken.operand[0])
        elif tmpToken.directive == "RESB": #locctr을 피연
            산자값만큼 증가시킴
            self.locctr += int(tmpToken.operand[0])
        elif tmpToken.directive == "BYTE": #locctr을 1 증
            가시킴
            self.locctr += 1
        elif tmpToken.directive == "EQU":
            if tmpToken.operand[0] == "*": #피연산자가
                *인 경우
                self.symtabList[i].modifySymbol(tmpToken.label, self.locctr) #현재 locctr의 값을 주소로 저장
            else :# 그렇지 않은 경우 피연산자를 통해 주소값을 계산하여 저장

            if '-' in tmpToken.operand[0]:
                tmpToken.operand = tmpToken.operand[0].split('-')

                self.symtabList[i].modifySymbol(tmpToken.operand[0], self.symtabList[i].search(tmpToken.operand[0]) - self.symtabList[i].search(tmpToken.operand[1]))

```

```

                self.TokenList[i].tokenList[j].setLocation(
self.symtabList[i].search(tmpToken.operand[0]) - self.symtabList
[i].search(tmpToken.operand[1]))
            elif tmpToken.directive == "LORG": #프로그램에
나왔던 literal을 저장
                self.locctr = self.literaltabList[i].addrLiteralTab(
self.locctr)
            elif tmpToken.directive == "END" : #프로그램에
나왔던 literal을 저장하고 마지막 주소값을 넣어줌
                self.locctr = self.literaltabList[i].addrLiteralTab(
self.locctr)
            self.TokenList[i].lastAddr = self.locctr
        pass

```

```

def printSymbolTable(self,fileName):
'''
작성된 SymbolTable들을 출력형태에 맞게 출력한다.
fileName : 저장되는 파일 이름
'''
file = open(fileName, 'w') #파일 열기
for i in range(0, len(self.symtabList)):
    for j in range(0, self.symtabList[i].getSize()):
        data = self.symtabList[i].getSymbol(j) + '\t\t\t' + format(
self.symtabList[i].getAddress(j),"X") + '\n'
        file.write(data)
    file.write('\n')
file.close() #파일 닫기

```

```

def printLiteralTable(self,fileName):
'''
작성된 LiteralTable들을 출력형태에 맞게 출력한다.
fileName : 저장되는 파일 이름
'''
file = open(fileName, 'w') #파일 열기
for i in range(0, len(self.literaltabList)):
    for j in range(0, self.literaltabList[i].getSize()):
        if self.literaltabList[i].getSize() == 0 :
            continue
        tmpStr = self.literaltabList[i].getLiteral(j)
        tmpStr = tmpStr[2:len(tmpStr)-1]
        data = tmpStr + '\t\t\t' + format(self.literaltabList[i].getAddress(j),"X") + '\n'
        file.write(data)
    file.close() #파일 닫기
def pass2(self):
'''
pass2 과정을 수행한다.\n
1) 분석된 내용을 바탕으로 object code를 생성하여
codeList에 저장.
'''
#토큰분리한 것들을 가지고 각 라인별 object code 작성
for i in range(0, len(self.TokenList)):
    for j in range(0,len(self.TokenList[i].tokenList)):
        self.TokenList[i].makeObjectCode(j)

check = 1

```

```

line_max = 0
line_check = 0
line = 0
textTmp = ""
code = ""
num = 0
#프로그램의 object program을 작성하여 codeList에 저장
for i in range(0,len(self.TokenList)):
    notFinish = -1
    for j in range(0,len(self.TokenList[i].tokenList)):
        tmpToken = self.TokenList[i].getToken(j)
        #Header record 작성
        if tmpToken.type == "DIRECTIVE":
            if tmpToken.directive == "START" or tmpToken.directive == "CSECT":
                startAddr = self.TokenList[i].getToken(0).location
                name = self.TokenList[i].getToken(0).label
                lastAddr = self.TokenList[i].lastAddr
                self.codeList.append( "H"+ ("%6s" %name)
                + format(startAddr,"06X") + format(lastAddr, "06X"))
                continue
            #Define record 작성
            if tmpToken.directive == "EXTDEF":
                tmp = "D"
                for k in range(0,len(self.symtabList[i].defList)):
                    tmp += ("%6s" %self.symtabList[i].defL

```

```

ist[k]) + format(self.symtabList[i].search(self.symtabList[i].defList[k]),
"06X")
                    self.codeList.append(tmp)
                    continue
            #Refer record 작성
            if tmpToken.directive == "EXTREF":
                tmp = "R"
                for k in range(0,len(self.symtabList[i].refList)):
                    tmp += ("%6s" %self.symtabList[i].refList[k])
                self.codeList.append(tmp)
                continue
            #Text record 작성
            if tmpToken.objectCode != " ":
                line_check = 0
                if check == 1:
                    textTmp = "T" + format(self.TokenList[i].getToken(0).location + line , "06X")
                    check = 0
                    notFinish = 2
                num += tmpToken.byteSize
                if num > 30: #현재까지의 byte의 개수가 한줄에
                    쓸수 있는 분량보다 많다면
                        num -= tmpToken.byteSize
                        check = 1

```



```

        textTmp += format(num, "02X") + code
        self.codeList.append(textTmp)
        line += num
        textTmp = ""
        code = ""
        line_max = 1
        num = 0
        if line_max == 1:
            code = self.TokenList[i].getToken(j).obje
ctCode.format("s")
            num += tmpToken.byteSize
            continue
        else :
            if notFinish == 1:
                textTmp = "T" + format(self.TokenList[i].
getToken(j).location, "06X")

                if line_max == 1:
                    textTmp = "T" + format(self.TokenList[i].
getToken(0).location + line, "06X")
                    line_max = 0

                if code != '':
                    code += self.TokenList[i].getToken(j).obje
ctCode.format("s")
                else :
                    code = self.TokenList[i].getToken(j).obje

```

```

ctCode.format("s")
            line_check = 1
            continue
        else :
            if line_check == 1:
                if notFinish == 2:
                    if textTmp == '':
                        continue
                    notFinish = 1
                    textTmp += format(num,"02X") + code
                    self.codeList.append(textTmp)
                    line += num
                    textTmp = ""
                    code = ""
                    num = 0
            if line_check == 1:
                textTmp += format(num, "02X") + code
                self.codeList.append(textTmp)
                textTmp = ""
                code = ""
                num = 0
                line = 0
                check = 1

#Modification record 작성
if len(self.symtabList[i].refList) != 0:
    for k in range(0,len(self.symtabList[i].modList)):
        tmp = "M" + format(self.symtabList[i].modLocati

```

```

onList[k], "06X") + format(self.symtabList[i].modSize[k], "02X") + format(self.symtabList[i].modList[k])
        self.codeList.append(tmp)
        #End record 작성
        if i == 0 :
            tmp = "E" + format(self.TokenList[i].getToken(0).location, "06X")
            self.codeList.append(tmp)
        else :
            self.codeList.append("E")
    pass
def printObjectCode(self, fileName):
    """
    작성된 codeList를 출력형태에 맞게 출력한다.

    fileName : 저장되는 파일 이름
    """
    file = open(fileName, 'w') #파일 열기
    for i in range(0, len(self.codeList)):
        file.write(self.codeList[i]+"\n")
        if self.codeList[i][0] == 'E':
            file.write("\n")
    file.close() #파일 닫기
#이 프로그램의 메인 루틴이다.
if __name__ == '__main__':
    assembler = Assembler("inst.data")
    assembler.loadInputFile("input.txt")
    assembler.pass1()

```

```

assembler.printSymbolTable("symtab_20160433.txt")
assembler.printLiteralTable("literal_20160433.txt")
assembler.pass2()
assembler.printObjectCode("output_20160433.txt")

```