숭실대학교 Soongsil University

시스템프로그래밍 2021 보고서

보고서 제출서약서

나는 숭실대학교 컴퓨터학부의 일원으로 명예를 지키면서 생활하고 있습니다.

나는 보고서를 작성하면서 다음과 같은 사항을 준수하였음을 엄숙히 서약합니다.

1. 나는 자력으로 보고서를 작성하였습니다.
   1.1. 나는 동료의 보고서를 베끼지 않았습니다.
   1.2. 나는 비공식적으로 얻은 해답/해설을 기초로 보고서를 작성하지 않았습니다.
2. 나는 보고서에서 참조한 문헌의 출처를 밝혔으며 표절하지 않았습니다. (나는 특히 인터넷에서 다운로드한 내용을 보고서에 거의 그대로 복사하여 사용하지 않았습니다.)
3. 나는 보고서를 제출하기 전에 동료에게 보여주지 않았습니다.
4. 나는 보고서의 내용을 조작하거나 날조하지 않았습니다.

과목	시스템프로그래밍 2021
과제명	SIC/XE 어셈블러 구현 (Project #1c)
담당교수	최 재 영 교 수
제출인	컴퓨터학부 20163340 강원경 (출석번호 107번)
제출일	2021년 05월 22일

차 례

1장. 프로젝트 동기/목적

이번 프로젝트는 우리가 2021년 1학기 '시스템 프로그래밍' 시간에 배운 SIC/XE 머신의 어셈블러 프로그램 을 구현하는 것이다. 지난 프로젝트에서 절차 지향 언어인 C를 통해 어셈블러를 구현하고, 객체지향 언어인 JAVA를 통해 구현하였다면, 이번에는 가장 유행하는 언어인 파이썬을 통해 다시 한 번 그 프로그램을 작성하는 것이다. 이를 통해 다양한 언어를 다룰 수 있는 능력을 기르고, 각각의 언어에 대해 어떤 특징을 가지고 어떤 장점이 있는가에 대해 생각해보는 것을 이번 프로젝트의 주요 목표 중 하나로 정하였다.

파이썬에는 여러 가지 특징이 있다. 다양한 External Library를 통해 일반적인 프로그래밍 뿐만 아니라 머신러닝, 데이터 분석 등의 기능을 수행할 수도 있고, 다른 언어의 프로그램들과의 유기적인 결합을 통해 비교적 속도가 느린 Python의 단점을 보완함과 동시에 다른 언어간의 프로그램들과의 연동을 비교적 간단하게 수행할 수 있도록 해주기도 한다.

하지만 위의 내용들은 지금의 프로젝트와는 크게 상관이 없는 내용이고, 파이썬을 통해 프로그램을 작성할 때는 이와 다른 특징들과 마주하게 된다. 이러한 특징들을 잘 파악하고 그에 맞게 프로그래밍을 할 수 있는 실력을 기르는 것이 무엇보다 가장 필요해 보였고, 다양한 경로를 통해 정보를 습득함으로써 처음 마주한 언어를 다룰 때 어떤 프로세스로 프로그래밍을 진행해야하는가에 대한 스스로의 고민과 함께 프로그래밍 과정중에 마주하는 여러 가지 문제를 능동적으로 해결할 수 있는 능력을 기르는 것을 이번 프로젝트의 주요 요소이자 이번 과제의 또 다른 목표로 정하였다.

2장. 설계/구현 아이디어

위에서 언급한 대로, 파이썬에는 여러 가지 특징이 있다. 그 중, 이번 프로젝트를 진행하면서 가장 크게 와닿았던 특징 두 가지는 다음과 같다.

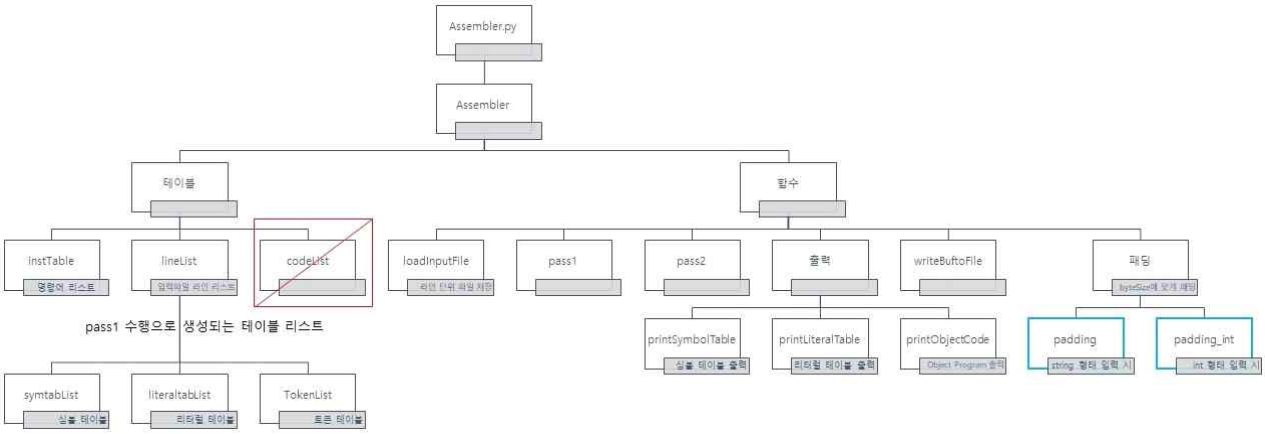
1. Python에는 형식 지정자가 없다. (즉, 모든 변수가 가변 객체이다.)
   ㄴ 이 부분은 javascript와 비슷하다. 개발자에게 더 많은 자유를 부여하기에 장점으로 여겨질 수도 있으나, 프로그래밍 과정 중 형식에 대한 체크가 진행되지 않고 빌드가 끝난 후 프로그램이 실행될 때에서야 비로소 형태에 대한 오류 체크가 가능하기 때문에 본인과 같이 이러한 개념과 자주 마주할 일이 없는 프로그래머들은 당황할 수밖에 없는 요소이다.
   하지만, 본인이 과거 3.4버전의 Python을 다뤘을 때와는 달리 3.6 버전 이후로부터는 프로그래머가 희망할 경우 변수에 대한 형식 지정이 가능해져 이러한 부분들을 프로그래밍 과정 중에 적극적으로 활용하였다.

2. Python에는 오버로드 개념이 없다.
   ㄴ 프로그래밍 과정 중 2번째로 당황한 부분이다. 본인이 JAVA로 작성한 프로그램의 경우 padding 함수와 생성자에서 오버로드의 개념을 사용한다. 이를 통해 하나의 메소드 이름만으로도 다양한 형태의 인자에 대한 처리가 가능하기 때문에 프로그래밍 과정 중 어떤 함수를 써야하는지에 대한 고민이 줄어들고 개발을 용이하게 한다. 하지만, 파이썬에서는 이러한 개념을 사용할 수 없기 때문에 각 형태에 대한 새로운 함수의 정의가 필요했고, 생성자 또한 오버로드가 불가능하기 때문에 JAVA와는 다른 방법으로 생성자에 대한 구현을 수행할 수밖에 없었다.

이번 프로젝트는 Windows 10 운영체제 상에서 IDE의 경우 ***Pycharm Community Edition 2021.1.1***을 사용하였고, Python 버전은 ***3.8***을 활용하였다.

이번 프로젝트에 작성된 Python 코드의 수행 절차는 지난번 JAVA로 작성했을 때와 크게 다르지 않다. 따라서, 이번 파트에서는 지난번 JAVA로 작성한 프로그램과 어떤 점이 다른지를 중점으로 기술하도록 한다.

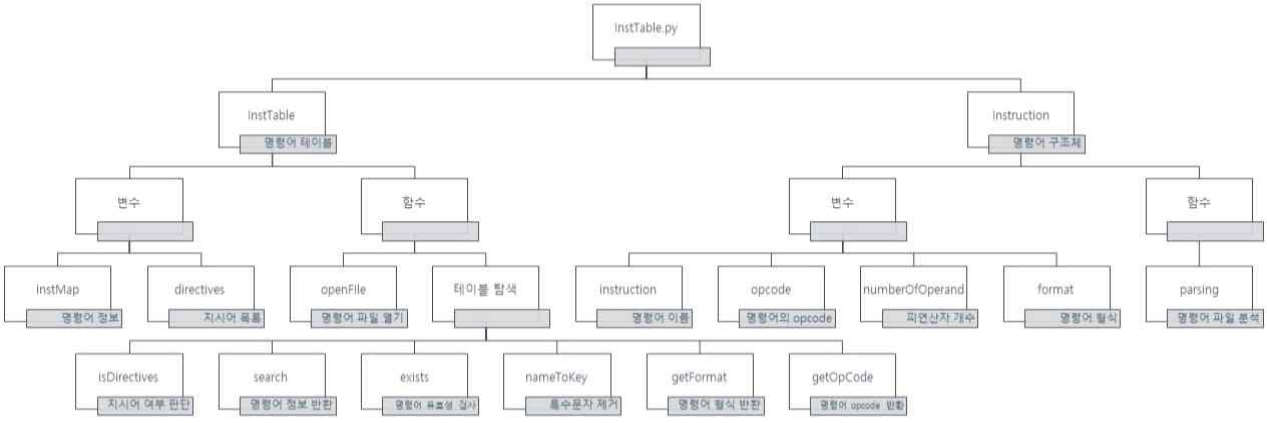
우선, 어셈블러 역할을 수행하는 Assmebler.py이다.



지난번 JAVA 과제와는 달리, 실제 objectProgram 작성에서 활용도가 없다시피한 codeList 테이블을 사용하지 않고, 대신 Token class의 내부변수 objectCode를 적극적으로 활용하였다.

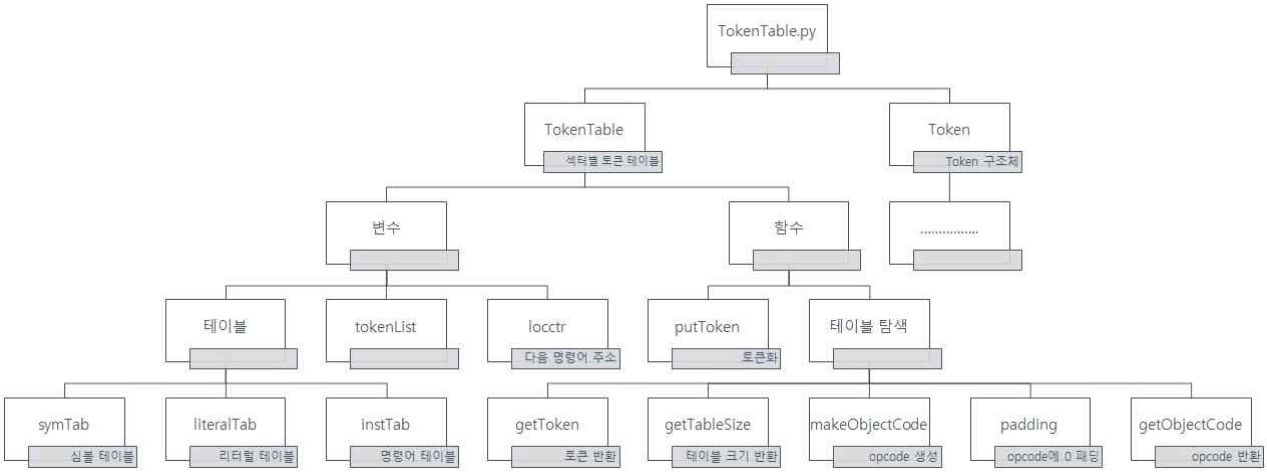
또, 오버로딩 개념을 지원하지 않는 Python 환경에 따라, (String, int), (int, int) 의 두가지 형태로 오버로딩 하였던 padding 함수를 각각의 경우에 따른 별개의 함수로 분할하여, (int, int) 형태의 padding 함수에서 1차적으로 string으로의 변환을 수행한 뒤 다시 (String, int) 형태의 함수를 호출하는 방식으로 변경하였다.

다음은 InstTable.py의 구조이다.

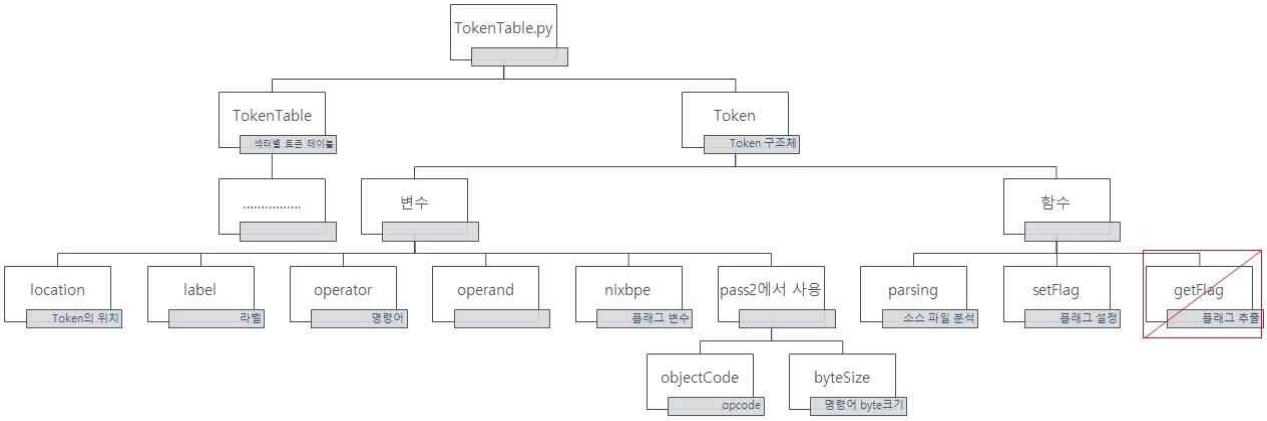


InstTable.py는 지난 프로젝트와 동일한 구성방식을 사용하였다.

TokenTable.py는 class별로 두 개의 조직도를 구성하였다.

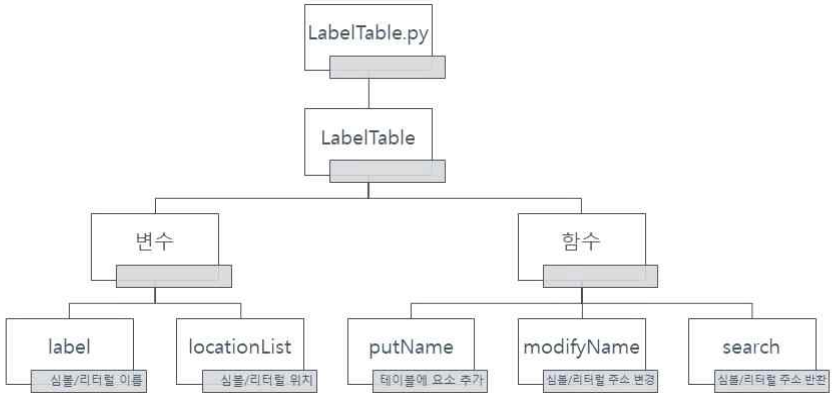


위 그림은 TokenTable.py 내의 TokenTable에 대한 구조이다. TokenTable에도 padding이라는 함수는 있지만, 이 클래스 내에는 하나의 형태만 구현되어 있기 때문에 오버로딩의 개념이 필요 없었다.



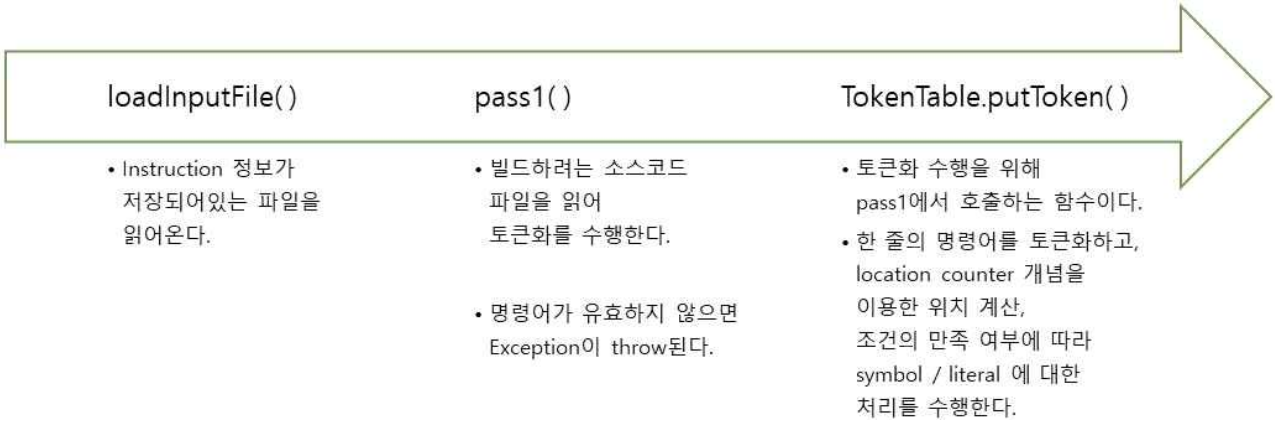
위의 그림은 Token class의 구조이다. getFlag 함수 역시 본인이 작성한 프로그램에서는 사용 빈도가 거의 없기 때문에 이를 사용하지 않고 문자 비교와 조건문등을 통한 방법으로 해당 함수의 역할을 대신하였다.

마지막으로 심볼 테이블과 리터럴 테이블에 사용되는 LabelTable.py의 구조이다.

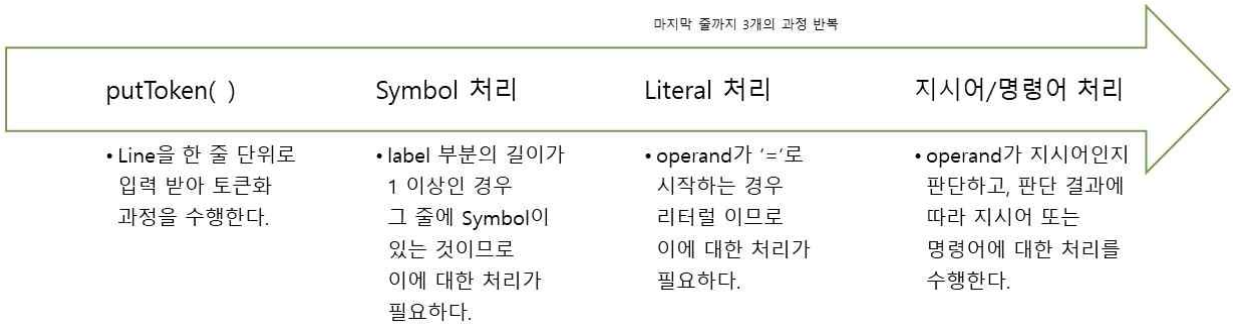


이제부터는 프로그램의 수행 과정의 측면에서 구현된 내용을 설명하려고 한다. main.py에 정의되어있는 메인 루틴의 진행에 따른 코드의 동작을 간단한 시각화와 설명을 통해 나타내도록 하고, 지난 JAVA 프로젝트와 다른점에 대해서만 추가 설명을 작성하였다.

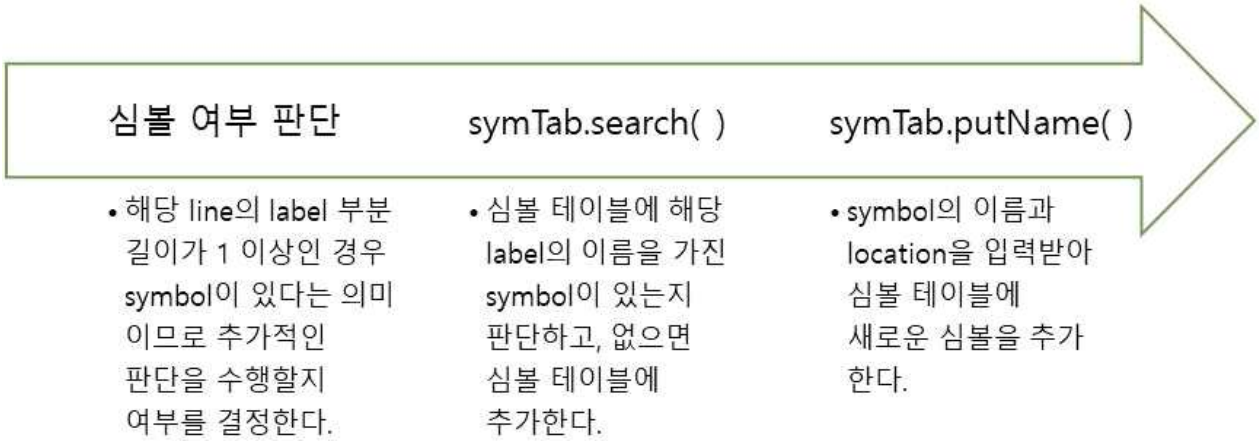
우선, 명령어가 저장된 데이터 셋과 초기 코드가 저장되어있는 파일을 읽어오는 과정이다.



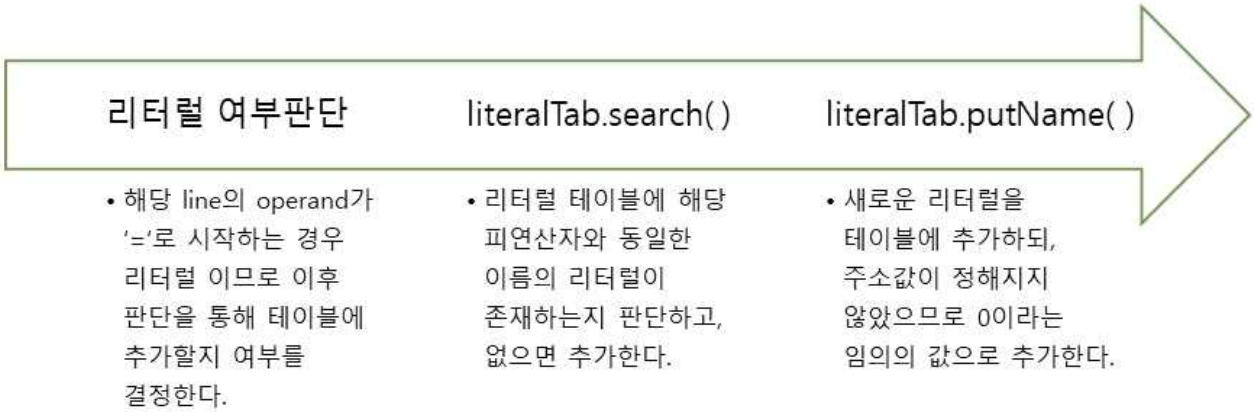
다음은 putToken( ) 함수에서 토큰화 과정 중 해당 라인에 심볼이나 리터럴이 있는지에 대한 판단 기준과 판단 순서를 나타낸 그림이다.



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



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



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

Python 에서의 append( )함수는 JAVA에서 add( )와 수행하는 내용이 동일하다. 따라서, 주요한 변경점으로 취급하지 않는다.

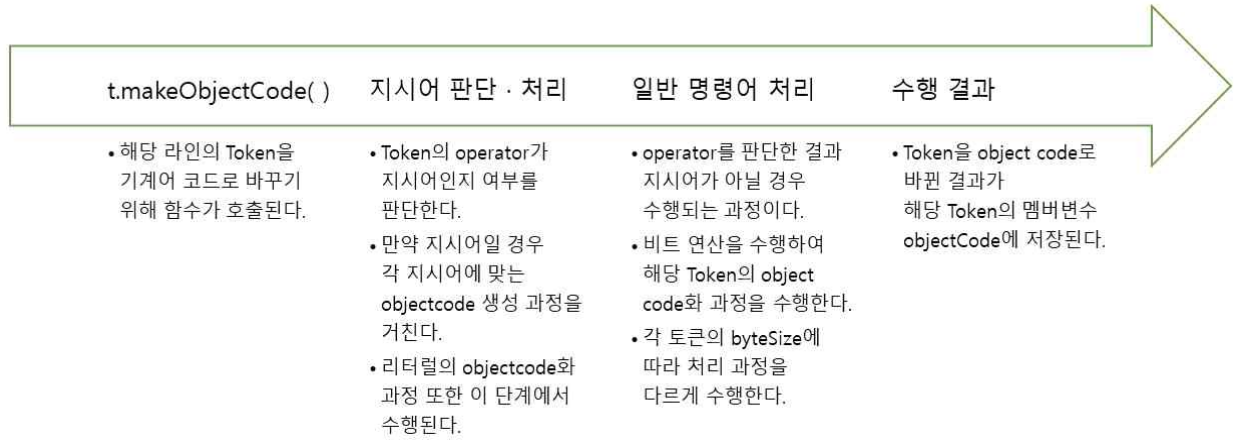
위의 과정이 모두 끝나면 pass1의 과정이 마무리되며, symTab과 literalTab에는 각각 심볼과 리터럴들이 Control Section별로 구분되어 저장된다. 이후에는 pass1에서 control section별로 분리되었던 테이블을 각각의 종류에 따라 병합하여 하나의 파일로 출력하는 과정이다.

다음으로 설명하는 내용은 pass2에서 수행되는 과정에 대한 부분이다.

구조 단계에서 한 차례 설명한 바 있는 내용으로, 본인이 작성한 코드에서는 codeList에 대한 활용도가 매우 떨어진다. 이는 JAVA 프로그램의 보고서에도 작성하였듯이, Control Section에 대한 구분이 어렵고, Control Section의 구분이 용이한 방법으로 변수를 재정의/추가 하기에는 이미 Token class 구조체 내에 objectCode라는 변수가 있기 때문에 이를 활용하면 추가적인 공간 낭비 없이 처리가 가능할 것으로 판단하였기 때문이다.

따라서 이번 Python 과제에서는 변수나 함수에 대한 별다른 명세가 주어지지 않았기 때문에, 이전 프로젝트에서 본인이 생각했던 개선점을 적극 반영하여 작성하였다. 이 과정에서 codeList를 사용하였던 루틴을 전부 objectCode로 연결시켰다.

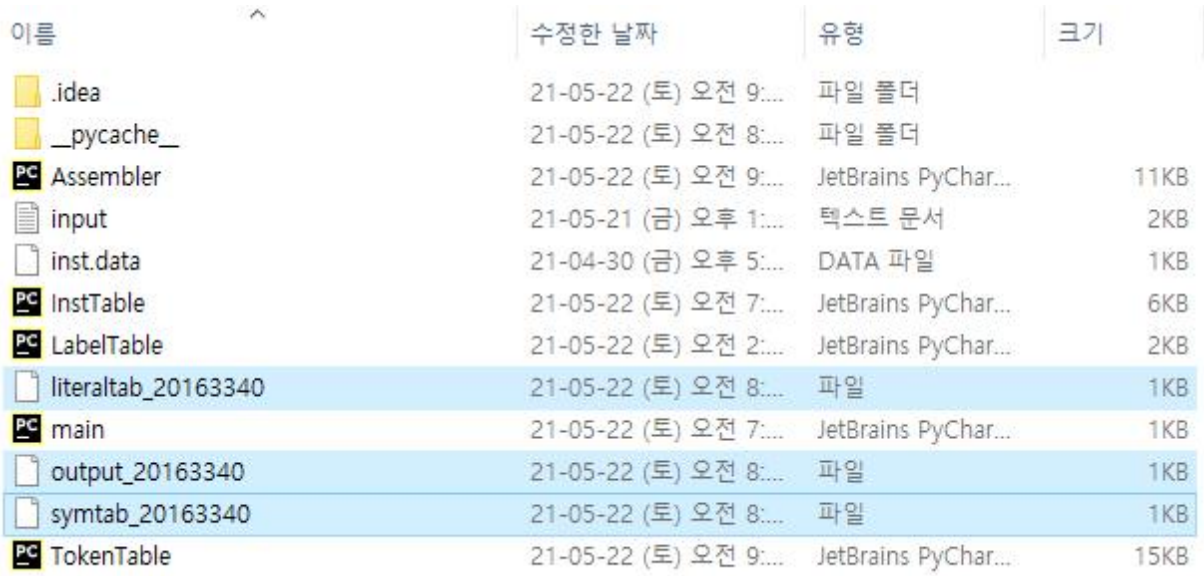
다음은 pass2에서 objectCode를 만드는 핵심 부분에 대한 설명이다.



마지막으로, pass2까지의 과정을 모두 거친 후 최종적으로 objectProgram의 양식을 가진 파일로 결과물을 출력하는 과정이다.

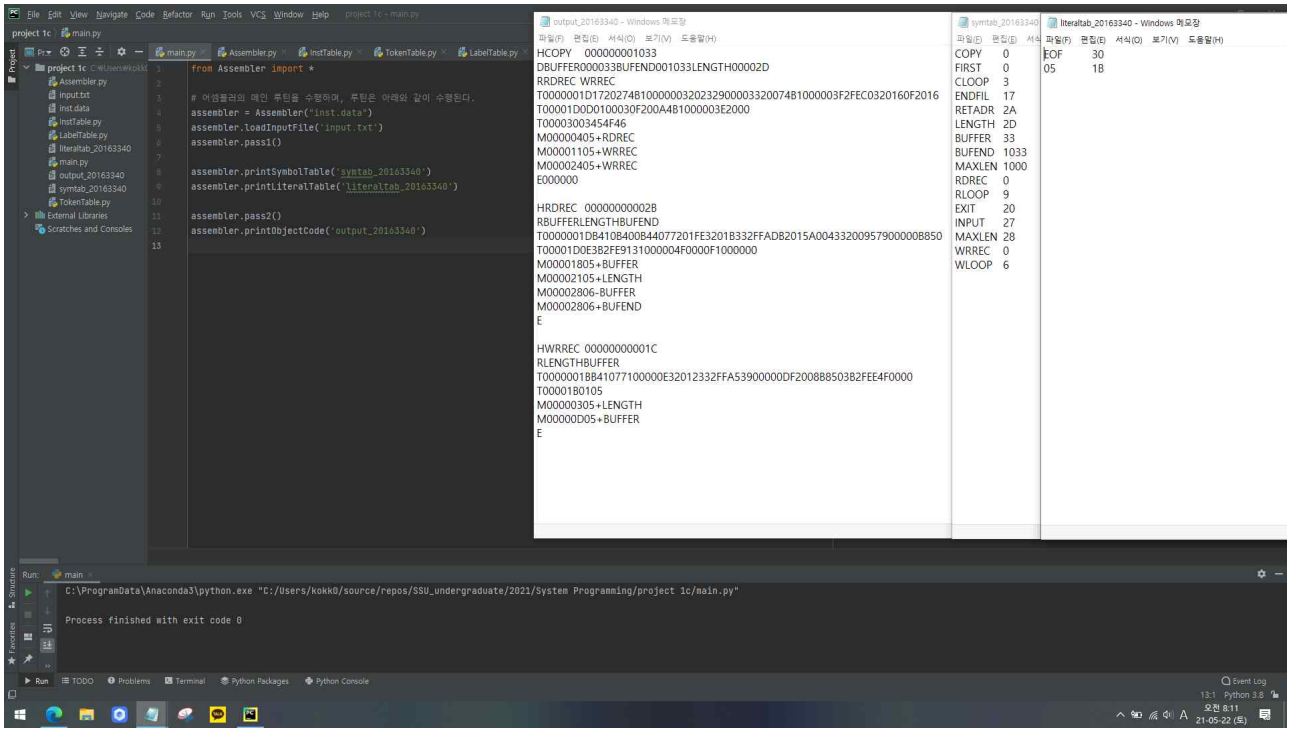
JAVA에서는 한 줄씩 코드를 저장하기 위해 파일에 내용을 직접 기록하는 BufferedWriter와 한 줄의 내용을 임시로 저장하는 String 형태의 버퍼, 이렇게 2가지의 버퍼를 사용하였으나, 파이썬에서는 BufferedWriter라는 개념이 없기 때문에 하나의 String 변수에 한줄의 내용을 저장하고 일정 길이가 넘어가게 되면 그 시점에 String의 내용을 파일에 기록하고 String의 내용을 다시 초기화시키는 방법을 선택하였다.

위의 과정을 모두 수행하게 되면 최종적으로 심볼테이블이 저장된 symtab_20163340, 리터럴 테이블이 저장된 literaltab_20163340, 그리고 ObjectProgram이 저장된 output_20163340이 저장된다.



이름	수정한 날짜	유형	크기
.idea	21-05-22 (토) 오전 9:...	파일 폴더	
__pycache__	21-05-22 (토) 오전 8:...	파일 폴더	
Assembler	21-05-22 (토) 오전 9:...	JetBrains PyChar...	11KB
input	21-05-21 (금) 오후 1:...	텍스트 문서	2KB
inst.data	21-04-30 (금) 오후 5:...	DATA 파일	1KB
InstTable	21-05-22 (토) 오전 7:...	JetBrains PyChar...	6KB
LabelTable	21-05-22 (토) 오전 2:...	JetBrains PyChar...	2KB
literaltab_20163340	21-05-22 (토) 오전 8:...	파일	1KB
main	21-05-22 (토) 오전 7:...	JetBrains PyChar...	1KB
output_20163340	21-05-22 (토) 오전 8:...	파일	1KB
symtab_20163340	21-05-22 (토) 오전 8:...	파일	1KB
TokenTable	21-05-22 (토) 오전 9:...	JetBrains PyChar...	15KB

3장. 수행 결과



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

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

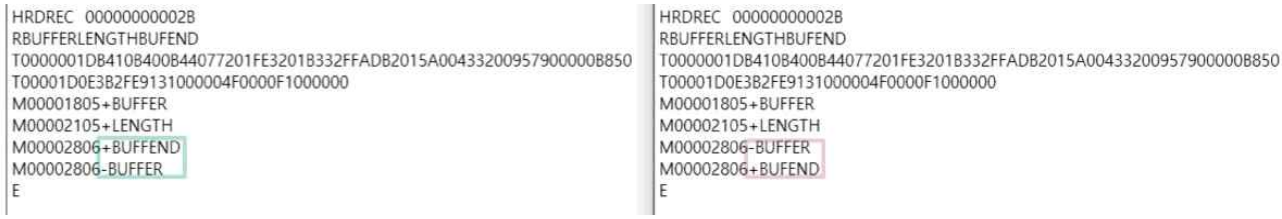
4장. 결론 및 보충할 점.

이번 과제 수행을 통해 본인은 전통적인 언어라 할 수 있는 C와 JAVA 외에 요즘 트렌드인 언어 Python을 간단히 다뤄보았다. 이를 통해 하나의 로직을 여러 개의 언어로 구현하는 능력을 길렀고, 처음 접하는 언어를 다룰 때에 어떤 프로세스로 프로그램을 작성해야 하는지에 대한 고민과 그에 대한 솔루션을 스스로 작성해본다는 목표를 달성하였다.

C에서 JAVA로 작성할 때에는 완전히 종류가 다른 언어이기 때문에 (C-절차지향, JAVA-객체지향) 로직을 새로 짜고 구현 방법을 달리 하는 등 상당히 많은 부분이 각각의 언어로 작성한 프로그램의 결과물이 달랐으나, JAVA에서 Python으로 작성하는 과정은 거의 동일한 로직으로 작성하면 되었다. 이는 Python이 확장성이 높은 언어중의 하나이고, 그들 중 하나가 바로 OOP를 지원한다는 점이 있기 때문에 가능한 일이었다. 하지만 프로그램 작성 중 Python과 JAVA간 다른점에 대해 고민하고 이를 옮길 때 어떤 방식으로 옮겨야하는가에 대한 고민이나, 함수의 이름은 같더라도 세부적인 동작 방법이 달라 이런 부분에 대한 연구와 고민이 필요했다.

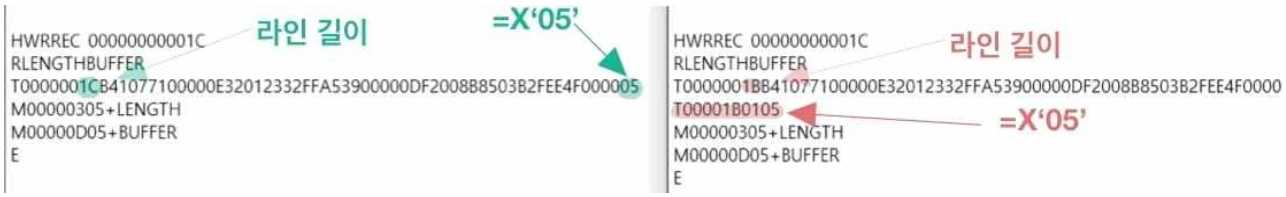
JAVA 프로그램과 로직이 거의 동일하기 때문에, 이상적인 정답과 본인의 정답이 다른 부분도 기존 프로그램과 동일하다. 실행 결과를 통해 어느 부분이 부족한 부분인지 다시 살펴보도록 한다.

첫 번째로, M 레코드에서의 기록 순서가 다르다.



왼쪽의 텍스트 파일은 이론에 가장 부합하는 Object Program의 작성 결과이고, 오른쪽의 파일은 본인이 작성한 코드의 실행 결과로 생성된 Object Program이다. 모범 답안을 살펴보면 (그림에서의 연녹색 부분) BUFEND의 주소값을 먼저 더한 후 BUFFER의 주소값을 빼지만, 본인이 작성한 프로그램(그림에서의 빨간 부분)은 BUFFER의 주소값을 먼저 빼고 BUFEND의 주소값을 더한다.

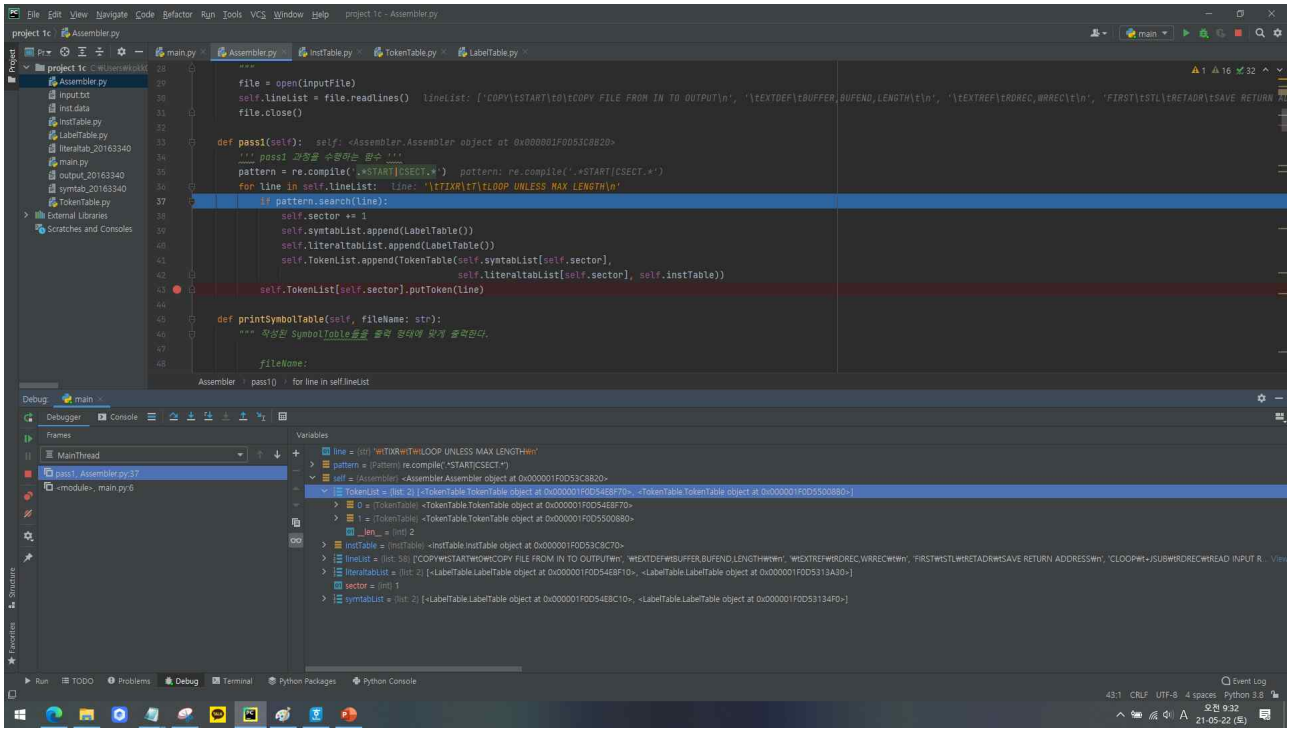
두 번째는 Literal에 대한 부분이다.



이 부분은 Control Section의 마지막에서 END를 만났을 때 Literal을 처리하는 부분이다. 모범 답안 (왼쪽)은 END를 만났을 때 리터럴을 프로그램 코드에 이어붙히지만 본인이 작성한 코드(오른쪽)는 리터럴을 처리하기 위해 새로운 한 줄을 만들고 리터럴들이 기록되기 시작하는 위치와 리터럴들의 길이, 그리고 object code화가 수행된 코드들을 뒤에 기록해 나간다. 이 과정에 따라 오른쪽의 그림에서 명령어가 담긴 토큰들이 모두 기록된 후, 줄 바꿈을 수행한 뒤 리터럴 정의 시작주소 (00001B), 리터럴들의 길이 (1byte = 01), object code화가 수행된 리터럴 (X'05' → 05)이 추가되었다.

5장. 디버깅

Pycharm Community Edition에서도 일반적인 IDE와 같이 디버거를 지원하기 때문에 코드를 작성하고 실행하면서 생각하지 못했던 부분에서 Exception이 발생하거나 의도한 코드 결과가 나오지 않았을 경우 디버거를 통한 디버깅을 수행하였다.



위 사진은 첫 번째 sector에서 일부 명령어에서 token이 이상하게 저장되어 한 단계씩 코드를 수행하면서 어느 과정에서의 로직이 문제가 있는지 확인하는 과정이다.

Pycharm은 다른 IDE와는 달리 해당 단계에서 사용되는 변수의 내용을 직접 코드 옆에 overlay 해주기 때문에 어떤 부분의 어떤 변수의 활용이 이상이 있는지 더 빠르게 파악할 수 있었다.

6장. 소스코드(+주석)

!! Pycharm은 소스코드 복사 시 indent를 무시하고 복사하기 때문에, 이 문서에 작성된 코드는 구조가 망가져있으므로 (Python에서는 indent로 코드 범위 구분)
자세한 코드의 내용은 소스코드 파일을 직접 확인해주시기 바랍니다.

main.py

```
from Assembler import *

# 어셈블러의 메인 루틴을 수행하며, 루틴은 아래와 같이 수행된다.
assembler = Assembler("inst.data")
assembler.loadInputFile('input.txt')
assembler.pass1()

assembler.printSymbolTable('symtab_20163340')
assembler.printLiteralTable('literaltab_20163340')

assembler.pass2()
assembler.printObjectCode('output_20163340')
```

Assembler.py

```
from LabelTable import *
from TokenTable import *
from InstTable import *
import re


class Assembler:
''' 어셈블러의 역할을 수행하는class이다. '''
sector = -1

def __init__(self, instFile: str = None):
if instFile:
self.lineList = list()
self.symtabList = list()
self.literaltabList = list()
self.TokenList = []
self.instTable = InstTable(instFile)
else:
pass

def loadInputFile(self, inputFile: str = None):
""" inputFile을 읽어 한 줄씩lineList에 저장한다.

inputFile:
소스코드가 저장된 입력 파일 이름.
"""
file = open(inputFile)
self.lineList = file.readlines()
file.close()

def pass1(self):
''' pass1 과정을 수행하는 함수'''
pattern = re.compile('.*START|CSECT.*')
for line in self.lineList:
if pattern.search(line):
self.sector += 1
self.symtabList.append(LabelTable())
self.literaltabList.append(LabelTable())
self.TokenList.append(TokenTable(self.symtabList[self.sector],
self.literaltabList[self.sector], self.instTable))
self.TokenList[self.sector].putToken(line)

def printSymbolTable(self, fileName: str):
""" 작성된SymbolTable들을 출력 형태에 맞게 출력한다.

fileName:
```

```
SymbolTable을 저장하려는 파일 이름
"""
file = open(fileName, 'w', encoding='utf-8')
for i in range(self.sector + 1):
size = len(self.TokenList[i].symTab.label)
for j in range(size):
s: str = ''
s += self.TokenList[i].symTab.label[j]
s += '₩t'
s += (str(hex(self.TokenList[i].symTab.locationList[j])).replace('0x', '', 1)).upper()
file.write(s + '₩n')
file.close()

def printLiteralTable(self, fileName: str):
""" 작성된LiteralTable들을 출력 형태에 맞게 출력한다.

fileName:
LiteralTable을 저장하려는 파일 이름
"""
file = open(fileName, 'w', encoding='utf-8')
for i in range(self.sector + 1):
size = len(self.TokenList[i].literalTab.label)
for j in range(size):
s: str = ''
s += self.TokenList[i].literalTab.label[j][3:-1]
s += '₩t'
s += (str(hex(self.TokenList[i].literalTab.locationList[j])).replace('0x', '', 1)).upper()
file.write(s + '₩n')
file.close()

def pass2(self):
''' pass2 과정을 수행하는 함수'''
for t in self.TokenList:
for i in range(t.getTableSize()):
t.makeObjectCode(i)

def padding(self, objectCode: str, byteSize: int) -> str:
""" objectCode의 앞에0으로 패딩을 넣어준다.

objectCode:
패딩을 넣으려고 하는string형태의objectCode
byteSize:
objectCode가 가져야 하는 바이트 크기
Return:
byteSize에 맞게0으로 패딩을 넣은string 형태의objectCode
"""
length = len(objectCode)
ans = ''
while length < byteSize * 2:
ans += '0'
length += 1
ans += objectCode
return ans.upper()

def padding_int(self, objectCode_int: int, byteSize: int) -> str:
""" objectCode의 앞에0으로 패딩을 넣어준다.

objectCode:
패딩을 넣으려고 하는int 형태의objectCode
byteSize:
objectCode가 가져야 하는 바이트 크기
Return:
byteSize에 맞게0으로 패딩을 넣은string 형태의objectCode
"""
temp = str(hex(objectCode_int)).replace('0x', '', 1)
return self.padding(temp, byteSize)

def writeBuftoFile(self, bufWriter: str, file):
""" 한 줄의object code가 저장된string을 파일에 기록하는 함수.

bufWriter:
한 줄의objectCode가 저장된string 형태의 버퍼
file:
내용을 기록하려는 파일
"""
line_len = (len(bufWriter) - 9) // 2
file.write(bufWriter[:7])
file.write(self.padding_int(line_len, 1))
file.write(bufWriter[9:])
file.write('₩n')

def printObjectCode(self, fileName: str):
""" 작성된objectCode들을 모아 하나의objectProgram을 만든다.

file:
내용을 기록하려는 파일
"""
file = open(fileName, 'w', encoding='utf-8')
cur_buf_length = 0
cur_sector_num = 0

bufWriter = ''

# Control Section별 토큰테이블 호출
for table in self.TokenList:
# H record 작성
bufWriter += 'H'
bufWriter += table.symTab.label[0] + '₩t'
bufWriter += self.padding_int(table.symTab.locationList[0], 3)
bufWriter += self.padding_int(table.locctr, 3) + '₩n'
file.write(bufWriter)
bufWriter = ''

# EXTREF의operand를 저장하는String 배열
ref_ops = list()

# EXTDEF, EXTREF 처리
for t in table.tokenList:
if t.byteSize == 0:
# EXTDEF 처리
if t.operator == 'EXTDEF':
bufWriter = 'D'
```

```
cur_buf_length = 1
for s in t.operand:
if cur_buf_length >= 57:
bufWriter += '₩n'
file.write(bufWriter)
bufWriter = 'D'
cur_buf_length = 1
bufWriter += s
for i in range(len(s), 6):
bufWriter += ' '
addr: int = table.symTab.search(s)
bufWriter += self.padding_int(addr, 3)
cur_buf_length += len(s) + 6
bufWriter += '₩n'
file.write(bufWriter)
bufWriter = ''
cur_buf_length = 0
# EXTREF 처리
elif t.operator == 'EXTREF':
ref_ops = t.operand
bufWriter = 'R'
cur_buf_length = 1
for s in t.operand:
if cur_buf_length >= 57:
bufWriter += '₩n'
file.write(bufWriter)
bufWriter = 'R'
cur_buf_length = 1
bufWriter += s
for i in range(len(s), 6):
bufWriter += ' '
cur_buf_length += 6
bufWriter += '₩n'
file.write(bufWriter)
bufWriter = ''
cur_buf_length = 0
break

is_newline = True

# T record 작성
for t in table.tokenList:
if is_newline:
bufWriter = 'T'
bufWriter += self.padding_int(t.location, 3)
bufWriter += '00'
is_newline = False

if not self.instTable.isDirectives(t.operator):
if len(bufWriter) + t.byteSize * 2 > 69:
self.writeBuftoFile(bufWriter, file)
bufWriter = 'T'
bufWriter += self.padding_int(t.location, 3)
bufWriter += '00'
bufWriter += self.padding(t.objectCode, t.byteSize)

# 리터럴 정의 부분에 대한 처리
else:
if t.operator == '*':
self.writeBuftoFile(bufWriter, file)
bufWriter = 'T'
bufWriter += self.padding_int(t.location, 3)
bufWriter += '03'
bufWriter += self.padding(t.objectCode, t.byteSize)

elif t.operator == 'WORD' or t.operator == 'BYTE':
bufWriter += t.objectCode

self.writeBuftoFile(bufWriter, file)

# M record 작성
for t in table.tokenList:
if t.operand is not None:
for line_op in t.operand:
for ext_op in ref_ops:
i = line_op.find(ext_op)
if i != -1 and t.location != -1:
if self.instTable.exists(t.operator):
bufWriter = ('M' + self.padding_int(t.location + 1, 3))
bufWriter += '05+'
else:
bufWriter = ('M' + self.padding_int(t.location, 3))
if i > 0 and line_op[i - 1] == '-':
bufWriter += '06-'
else:
bufWriter += '06+'
bufWriter += (ext_op + '₩n')
file.write(bufWriter)
bufWriter = ''

# E record 작성
if cur_sector_num == 0:
bufWriter = 'E'
bufWriter += self.padding_int(table.symTab.locationList[0], 3)
bufWriter += '₩n₩n'
else:
bufWriter = 'E₩n₩n'
file.write(bufWriter)
bufWriter = ''
cur_sector_num += 1
file.close()
```

InstTable.py

```
class Instruction:
''' 각각의 명령어에 대한 정보가 담기게 되는class 구조체.
명령어에 대한 정보를 저장하고, 기초적인 연산을 제공한다.'''

def __init__(self, line: str = None):
self.instruction: str = ''
self.opcode: int = 0
self.numberOfOperand: int = 0
self.format: int = 0
if line:
self.parsing(line)
else:
pass

def parsing(self, line: str):
""" 문자열을 파싱하여instruction 정보를 파악하고 저장한다.

line:
instruction 명세 파일로부터 한 줄씩 가져온 문자열
"""
line_separate = line.split('₩t')
self.instruction = line_separate[0]
self.format = int(line_separate[1])
self.opcode = int(line_separate[2], 16)
self.numberOfOperand = int(line_separate[3])


class InstTable:
'''
모든instruction의 정보를 관리하는 클래스. instruction data들을 저장한다.
또한instruction 관련 연산, 예를 들면 목록을 구축하는 함수, 관련 정보를 제공하는 함수 등을
제공 한다.
'''
MAX_INST = 256
directives = ["START", "END", "BYTE", "WORD", "RESB", "RESW",
"EXTDEF", "EXTREF", "LTORG", "*", "EQU", "CSECT"]
instMap = {}  # 파이썬에서는Map 형식이 아닌Dictionary 방식으로 저장한다. 기능은 비슷함.

def __init__(self, instFile=None):
if instFile:
self.openFile(instFile)
else:
pass

def openFile(self, fileName: str):
""" 입력받은 이름의 파일을 열고 그 내용을parsing하여instMap에 저장한다.

fileName:
명령어 정보가 담긴 파일 이름
"""
file = open(fileName)
line = file.readlines()
for s in line:
inst = Instruction(s)
self.instMap[inst.instruction] = inst
file.close()

def isDirectives(self, name: str) -> bool:
""" 입력받은 명령어가 지시어 또는 예약어(ex. 선언자'*') 인지 확인하는 함수.

name:
판단하려는 명령어의 이름
Return:
지시어/예약어인 경우true, 아닌 경우false
"""
tempName = self.removeNewLine(name)
for s in self.directives:
if tempName == s:
return True
return False

def nameToKey(self, name: str) -> str:
""" 입력받은string에서'+' 문자를 제거하는 함수

name:
검색중 오류를 발생시킬 수 있는 특수문자가 포함된string
Return:
특수문자가 제거된string
"""
return name.replace('+', '')

def removeNewLine(self, name: str) -> str:
""" 입력받은string에서 개행 문자를 제거하는 함수

name:
검색중 오류를 발생시킬 수 있는 개행문자가 포함된string
Return:
개행문자가 제거된string
"""
return name.replace('₩n', '')

def getFormat(self, name: str) -> int:
""" 입력받은 이름을 가진 명령어가 몇 형식인지 반환하는 함수.

name:
검색하려는 명령어의 이름
Return:
명령어의format
Exception:
해당 명령어가 유효하지 않은 명령어인 경우Exception 발생
"""
if name[0] == '+':
return 4
if not self.exists(name):
raise Exception("없는 명령어입니다.")
return self.search(name).format
```

```
def getOpCode(self, name: str) -> int:
""" 입력받은 이름을 가진 명령어의opcode를 반환하는 함수.

name:
검색하려는 명령어의 이름
Return:
명령어의opcode
Exception:
해당 명령어가 유효하지 않은 명령어인 경우Exception 발생
"""
if not self.exists(name):
raise Exception("없는 명령어입니다.")
return self.search(name).opcode

def exists(self, name: str):
""" 입력받은 이름을 가진 명령어가instTable에 정의되어있는지 확인하는 함수.

name:
검색하려는 명령어의 이름
Return:
명령어 테이블에 있는 경우true, 없는 경우false
"""
return self.nameToKey(self.removeNewLine(name)) in self.instMap

def search(self, name: str):
""" 입력받은 이름을 가진 명령어에 대한 정보를 반환하는 함수.

name:
검색하려는 명령어의 이름
Return:
그 명령어의 정보가 담긴Instruction 구조체
"""
return self.instMap.get(self.nameToKey(name))
```

TokenTable.py

```
from InstTable import *
from LabelTable import *
import re

MAX_OPERAND = 3
nFlag = 32
iFlag = 16
xFlag = 8
bFlag = 4
pFlag = 2
eFlag = 1


class Token:
    """ 각 라인별로 저장된 코드를 단어 단위로 분할한 후 의미를 해석하는 데에 사용되는 변수와
    연산을 정의한다.
    의미 해석이 끝나면pass2에서object code로 변형되었을 때의 바이트 코드 역시 저장한다.
    """

    def __init__(self, line: str = None):
        self.location: int = 0
        self.label: str = None
        self.operator: str = None
        self.operand: str = None
        self.comment: str = None
        self.nixbpe = 0

        self.objectCode: str = None
        self.byteSize: int = 0
        if line is not None:
            self.parsing(line)

    def parsing(self, line):
        """ line을 분석하여 토큰화를 수행하는 함수.

        line:
        분석을 수행할 한 줄의 내용이 담긴string
        """
        line_temp = line.split('₩t')
        if len(line_temp) > 0:
            self.label = line_temp[0]
        if len(line_temp) > 1:
            self.operator = line_temp[1]
        if len(line_temp) > 2:
            self.operand = line_temp[2].split(',')
        if len(line_temp) > 3:
            self.comment = line_temp[3]

    def setFlag(self, flag, value=1):
        """ n,i,x,b,p,e의flag를 설정하는 함수.

        flag:
        설정하려는flag의 이름
        value:
        플래그를 어떤 값으로 세팅할 것인지 지정. 값을 지정하지 않으면1로 세팅한다.
        """
        self.nixbpe |= flag


class TokenTable:
    def __init__(self, symTab: LabelTable, literalTab: LabelTable, instTab: InstTable):
        self.symTab = symTab
        self.literalTab = literalTab
        self.instTab = instTab
        self.tokenList = []
        self.locctr: int = 0

    def getToken(self, index: int) -> Token:
        """ tokenTable의index 번째에 위치에 있는Token을 리턴

        index:
        찾으려는 토큰의 테이블에서의 인덱스 번호
        Return:
        index 번째에 위치한 토큰 정보 구조체
        """
        return self.tokenList[index]

    def getTableSize(self) -> int:
        """ tokenList의 크기 반환

        Return:
        tokenList의 크기
        """
        return len(self.tokenList)

    def getObjectCode(self, index: int) -> str:
        """ tokenTable의index 번째에 위치한Token의ObjectCode를 반환

        index:
        찾으려는 토큰의 테이블에서의 인덱스 번호
        Return:
        토큰의Object Code
        """
        return self.tokenList[index].objectCode

    def padding(self, objectCode: str, byteSize: int) -> str:
        """ objectCode의 앞에0으로 패딩을 넣어준다.

        objectCode:
        패딩을 넣으려고 하는string형태의objectCode
        byteSize:
        objectCode가 가져야 하는 바이트 크기
        Return:
        byteSize에 맞게0으로 패딩을 넣은string 형태의objectCode
        """
        length: int = len(objectCode)
```

```
ans: str = ''
while length < byteSize * 2:
ans += '0'
length += 1
ans += objectCode
return ans.upper()

def putToken(self, line: str):
""" 일반 문자열을 입력받아Token 단위로 분리시켜tokenList에 추가

line:
토큰화가 수행되기 전 문자열
"""
if line[0] == '.':
return
token: Token = Token(line)
# 심볼 테이블 넣는 과정. <- label 처리
if len(token.label) > 0:
if self.symTab.search(token.label) == -1:
self.symTab.putName(token.label, self.locctr)
# 리터럴 테이블 넣는 과정<- =C'..', =X'..' 처리
if token.operand != ['*'] and token.operand[0] != '' and token.operand[0][0] == '=':
if self.literalTab.search(token.operand[0]) == -1:
self.literalTab.putName(token.operand[0], 0)  # 리터럴의 주소값은 나중에 수정된다.
# operator가 지시어일 경우 각 경우에 맞게 처리
if self.instTab.isDirectives(token.operator):
token.byteSize = 0
if token.operator == 'RESW':
token.location = self.locctr
nums: int = int(token.operand[0])
self.locctr += nums * 3
elif token.operator == 'RESB':
token.location = self.locctr
nums: int = int(token.operand[0])
self.locctr += nums
elif token.operator == 'EXTDEF' or token.operator == 'EXTREF':
token.location = -1
elif token.operator == 'LTORG' or token.operator == 'END':
token.location = -1
for s in self.literalTab.label:
literal_temp = '₩t*₩t'
literal_temp += s
self.putToken(literal_temp)
elif token.operator == '*':
length = len(token.operand[0]) - 4
if token.operand[0][1] == 'X':
length //= 2
token.byteSize = length
token.location = self.locctr
self.literalTab.modifyName(token.operand[0], self.locctr)
self.locctr += length
elif token.operator == 'CSECT' or token.operator == 'START':
token.location = self.locctr
elif token.operator == 'EQU':
templocctr = self.locctr
if token.operand[0][0] != '*':
op = None
for i in range(0, len(token.operand[0])):
if token.operand[0][i] == '+' or token.operand[0][i] == '-':
op = token.operand[0][i]
if op is not None:
operation = re.split(r'[+-]', token.operand[0])
v1 = self.symTab.search(operation[0])
v2 = self.symTab.search(operation[1])
if op == '+':
templocctr = v1 + v2
else:
templocctr = v1 - v2
token.location = templocctr
self.symTab.modifyName(token.label, templocctr)
elif token.operator == 'BYTE':
token.location = self.locctr
if token.operand[0][0] == 'C':
self.locctr += len(token.operand[0]) - 3
else:
self.locctr += (len(token.operand[0]) - 3) // 2
elif token.operator == 'WORD':
token.location = self.locctr
self.locctr += 3
# operator가 일반적인 명령어일 경우 처리
else:
token.byteSize = self.instTab.getFormat(token.operator)
token.location = self.locctr
self.locctr += token.byteSize

self.tokenList.append(token)

""" pass2에서objectCode를 만들 때 사용하는 함수.
명령어 테이블, 심볼 테이블, 리터럴 테이블을 참조하여objectCode를 만들고
Token class 객체 내의objectCode 변수에 저장.

index:
objectCode를 만드려는 토큰의 인덱스 번호
"""

def makeObjectCode(self, index: int):
t = self.getToken(index)
obj: int = 0
if t.operator is None and (t.operator == '' or t.operator is None):
return
# 지시어에 대한 처리 부분
if self.instTab.isDirectives(t.operator):
if t.operator == 'BYTE':
temp = t.operand[0][2:-1]
t.objectCode = temp
t.byteSize = len(temp) // 2
# print(t.objectCode + '₩n') <-디버깅용
elif t.operator == 'WORD':
t.byteSize = 3
# operand가 숫자라면 그대로objectCode에 넣음
if t.operand[0][0].isdigit():
t.objectCode = t.operand[0]
```

```
# 문자일 경우 처리
else:
op = None
for i in (0, len(t.operand[0]) - 1):
if t.operand[0][i] == '+' or t.operand[0][i] == '-':
op = t.operand[0][i]
operation = re.split(r'[+-]', t.operand[0])
v1: int = self.symTab.search(operation[0])
if v1 == -1:
v1 = 0
v2: int = self.symTab.search(operation[1])
if v2 == -1:
v2 = 0
if op == '+':
obj = v1 + v2
elif op == '-':
obj = v1 - v2
temp = hex(obj)
t.objectCode = self.padding(temp.replace('0x', '', 1), t.byteSize)
# print(t.objectCode + '₩n')
# 리터럴을 정의한 토큰일 경우
elif t.operator == '*':
if t.operand[0][1] == 'C':
for j in range(3, len(t.operand[0]) - 1):
obj <<= 8
obj |= ord(t.operand[0][j])
t.objectCode = (hex(obj).replace('0x', '', 1)).upper()
elif t.operand[0][1] == 'X':
temp = t.operand[0][3:-1]
t.objectCode = temp
# print(t.objectCode + '₩n')
# 일반적인 명령어일 경우의 처리 부분
else:
obj = self.instTab.getOpCode(t.operator)

if t.byteSize == 2:
for i in range(len(t.operand)):
obj <<= 4
if i == 1 and (t.operand[i] == '' or t.operand[i] is None):
obj |= 0
elif t.operand[i] == 'SW':
obj |= 9
elif t.operand[i] == 'PC':
obj |= 8
elif t.operand[i] == 'F':
obj |= 6
elif t.operand[i] == 'T':
obj |= 5
elif t.operand[i] == 'S':
obj |= 4
elif t.operand[i] == 'B':
obj |= 3
elif t.operand[i] == 'L':
obj |= 2
elif t.operand[i] == 'X':
obj |= 1
elif t.operand[i] == 'A':
obj |= 0
if len(t.operand) == 1:
obj <<= 4
elif t.byteSize == 3:
obj <<= 4
# n, i 플래그 세팅
if t.operand[0] != '':
if t.operand[0][0] == '@':
t.setFlag(nFlag, 1)
elif t.operand[0][0] == '#':
t.setFlag(iFlag, 1)
else:
t.setFlag(nFlag, 1)
t.setFlag(iFlag, 1)
else:
t.setFlag(nFlag, 1)
t.setFlag(iFlag, 1)

# x 플래그 세팅
for op_iter in t.operand:
if op_iter == 'X':
t.setFlag(xFlag, 1)
break

# operand가 있는 경우
if t.operand != [''] and t.operand[0] is not None:
# immediate addressing 일 때
if t.operand[0][0] == '#':
obj |= t.nixbpe
obj <<= 12
tmp: str = t.operand[0][1:]
obj |= int(tmp)
else:
t.setFlag(pFlag, 1)
obj |= t.nixbpe
obj <<= 12
pc: int = t.location + t.byteSize
target_addr: int

# operand가literal일 때 리터럴 테이블을 검색한다.
if t.operand[0][0] == '=':
target_addr = self.literalTab.search(t.operand[0])
# 간접 주소 방식인 경우'@'를 제외하고 심볼 테이블 검색
elif t.operand[0][0] == '@':
target_addr = self.symTab.search(t.operand[0][1:])
# simple addressing 인 경우 심볼 테이블을 찾기만 하면 됨.
else:
target_addr = self.symTab.search(t.operand[0])

if target_addr == -1:
target_addr = 0
disp: int = target_addr - pc
disp &= 0xFFF
obj |= disp
# operand가 없는 경우b,p,e 플래그 모두0이고disp도 없으므로 그냥 더한다.
```

```
else:
obj |= t.nixbpe
obj <<= 12
elif t.byteSize == 4:
obj <<= 4
'''
4형식 명령어(e=1) 특성상simple addressing이고(n,i=1), base나pc를 참조할 수
없기때문에(→주소 모름|| b,p=0)
n,i,e 플래그를 우선적으로1로 세팅하고, X는 사용 여부를 판단하여 세팅한다.
'''
t.setFlag(nFlag)
t.setFlag(iFlag)
t.setFlag(eFlag)

for op_iter in t.operand:
if op_iter == 'X':
t.setFlag(xFlag)
break

obj |= t.nixbpe
obj <<= 20
target_addr: int = self.symTab.search(t.operand[0])
if target_addr == -1:
target_addr = 0
target_addr &= 0xFFFFF
obj |= target_addr

t.objectCode = (hex(obj).replace('0x', '', 1)).upper()
t.objectCode = self.padding(t.objectCode, t.byteSize)
# print(t.objectCode + '₩n')
```

LabelTable.py

```
class LabelTable:
"""symbol, literal과 관련된 데이터와 연산을 소유한다.
Control Section별로 별도의 테이블을 가질 수 있도록 설계한다."""

def __init__(self):
self.label = list()
self.locationList = list()

def search(self, label_name: str) -> int:
"""인자로 전달된symbol, literal이 어떤 주소를 가리키고 있는지 반환

label_name:
검색하려는symbol/literal의 이름
returns:
테이블에 있는symbol/literal일 경우 주소값을, 아닐 경우-1을 반환.
"""
address: int = -1
try:
idx = self.label.index(label_name)
address = self.locationList[idx]
except ValueError:
pass
return address

def putName(self, new_name: str, location: int):
""" 새로운symbol과literal을 테이블에 추가한다.
새로운symbol/literal을table에 추가한다.

new_name:
새로 추가되는symbol/literal의 이름
location:
해당symbol/literal이 테이블에 저장될 때 가지는 최초 주소값. 만약 값의 변경이 필요할 경우
별도의 함수에서 주소값의 변경을 처리한다.
"""
if self.search(new_name) == -1:
self.label.append(new_name)
self.locationList.append(location)

def modifyName(self, label_name: str, new_location: int):
"""기존에 존재하는symbol/literal 의 주소값을 변경한다.
label_name:
변경하려는symbol/literal의 이름
new_location:
바꾸려고 하는 주소값
"""
try:
idx: int = self.label.index(label_name)
self.locationList[idx] = new_location
except ValueError:
pass
```