시스템프로그래밍(나) Project 1(b)

- SIC/XE Assembler in Java -

컴퓨터학부 정해성

목차

1. 동기 및 목적

본 보고서는 SIC/XE 소스를 파싱하여 오브젝트 프로그램으로 변환하는 어셈블러를 자바 언어로 재구현하는 과제의 보고서입니다. 또한 정의되지 않은 심볼을 사용하는 등 SIC/XE 소스의 문법적 오류를 감지하는 기능도 추가하였습니다. 자바라는 객체지향 언어로 구현하는 만큼, SIC/XE 문법의 공통점들을 최대한 추출하여 객체지향적으로 설계하기 위해 깊이 고민해보았습니다.

2. 설계/구현 아이디어

과제에서 주어진 템플릿을 기반으로 과제를 진행하였고, 핵심이 되는 로직은 `ControlSection` 클래스에서 확인할 수 있습니다.

- **instruction table**

  inst_table.txt는 공백 문자를 기준으로 순서대로 명령어의 이름, opcode, 피연산자의 개수, 형식 순으로 나열되어 있습니다. 여기서 형식은 해당 명령어가 차지하는 바이트 수를 의미하며 3형식, 2형식, 1형식이 존재합니다.

  따로 형식에 명시하지 않았지만 appendix를 살펴봤을 때 3형식인 명령어는 피연산자가 존재한다면 메모리 주소 symbol 또는 immediate, 2형식인 명령어는 피연산자로 레지스터 1개 또는 2개가 온다는 사실을 관찰할 수 있었습니다. 그래서 이후 명령어 라인을 `InstructionToken`으로 변환할 때 해당 관찰이 참이라고 가정하고 구현하였습니다.

- **`Numeric`**

  제가 설계한 `Numeric` 클래스는 말 그대로 실제 값을 그대로 반영하고 있는 클래스입니다. 예를 들어 리터럴 표현인 =123, =C'EOF', =X'05'나 WORD, BYTE의 피연산자 자리에 들어오는 123, C'EOF', X'05'를 raw한 값(byte 배열)으로 나타내 줍니다. 예를 들어 123은 byte 배열에 00007B이 저장되고, X'05'는 05가 저장되는 방식입니다. Numeric 객체를 초기화 할 때 인자로 들어오는 문자열을 기반으로 사이즈 필드와 배열의 크기가 결정되기 때문에 C'EOFFFFF...'와 같이 숫자로 표현하면 int 범위를 초과하는 경우들에 대해서도 대응할 수 있습니다.

  이 Numeric 클래스는 Literal 클래스와 이후에 설명할 `ValueDirectiveToken`에서 사용됩니다. Pass 2 단계에서 해당 값이 직접 오브젝트 프로그램에 포함되어야 할 때, byte 배열을 16진수 문자열로 변환해주는 `packValue()` 메서드를 활용하고 있습니다.

Numeric.packValue()

```java
// bytes를 16진수 문자열로 변환한 값을 리턴합니다.
public String packValue() {  1 usage  cometj03 *
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < size; i++) {
        builder.append(String.format("%02X", bytes[i] & 0xFF));
    }
    return builder.toString();
}
```

ControlSection.buildObjectCode() 중

```java
if (token instanceof ValueDirectiveToken) {
    List<Numeric> numerics = ((ValueDirectiveToken) token).getNumerics();
    String value = numerics.stream()  Stream<Numeric>
            .map(numeric -> numeric.packValue())  Stream<String>
            .collect(Collectors.joining());
    objCode.addText(token.getAddress(), value);
} else {
```

- **ValueDirectiveToken**

  구현할 때 까다로운 것 중 하나가 지시어를 처리해주는 것이었습니다. START, CSECT처럼 단순한 것도 있지만, WORD, BYTE, LTORG와 같이 직접 오브젝트 프로그램의 Text record에 관여하는 지시어도 있기 때문입니다. 이 문제를 객체지향적으로 해결해보고자 하여 `DirectiveToken`의 하위 클래스로 `ValueDirectiveToken`를 정의하였습니다. `ValueDirectiveToken`은 지시어 중 WORD, BYTE, LTORG(, 간혹 END)에 대응되는 토큰으로, Pass 2에서 오브젝트 프로그램을 구성할 때 Text record 내용을 추가해야 한다는 지시어 토큰을 의미합니다. 간혹 END 지시어가 포함되는 이유는 중간에 LTORG가 없을 경우 END 부분에서 아직 정의되지 않은 리터럴이 정의되기 때문입니다. 앞서 언급한 것처럼 내부적으로 `Numeric` 객체를 활용하고 있으며, LTORG의 경우 (리터럴에 대응되는) `Numeric` 객체가 1개 이상일 수 있기 때문에 리스트 형태로 구성되어 있습니다.

- **Operand**

  `Operand` 클래스는 명령어의 피연산자를 분류하기 위해 정의된 추상 클래스입니다. 하위 클래스로는 `SymbolOperand`, `ImmediateOperand`, `RegisterOperand`가 있습니다. `SymbolOperand`는 테이블에서 검색 후 값을 넣어주는 방식으로, 일반적인 심볼은 물론이고 indirect 표현이나 리터럴도 해당 operand에 해당합니다.

  `InstructionToken`에서 피연산자의 타입으로 사용되며, 명령어 토큰이 text record로 변환될 때 피연산자의 타입을 확인하여 값을 적절히 변환하는 것으로 설계해둔 상태입니다. 시간상의 이유로 Operand 부분은 아직 구현하지 못했습니다.

- **TextInfo**

  `InstructionToken`은 Pass 2 단계에서 text record의 내용으로 변환되어 추가되어야 합니다. 이때, 이 변환되는 과정을 도와주는 클래스로 `TextInfo`라는 클래스를 정의했습니다. `TextInfo`는 필드로 `opcode, nixbpe, displacement, sizeHalfBytes`를 가지고 있으며 해당 필드들로 text record의 내용을 생성합니다.

  `InstructionToken.getTextInfo()` 메서드에서 `TextInfo` 객체가 생성되는데, 이곳에서 대부분의 에러 핸들링이 이루어지기도 합니다. 앞서 Operand 클래스의 미비로 `getTextInfo()` 메서드 내에서 피연산자를 일일이 경우에 따라 나눠 처리해주고 있다는 점이 조금 아쉽습니다. 메서드의 수행이 완료되면 명령어에 해당하는 opcode, `nixbpe` 필드, `displacement`에 들어갈 `addr` 값이 할당되고, 마지막으로 형식에 따라 `sizeHalfBytes`가 결정되어 생성된 `TextInfo` 객체를 반환합니다.

  `TextInfo` 클래스는 필드들의 내용을 조합하여 대응되는 문자열을 생성하는 메서드가 정의되어 있습니다.

```
private void handlePass2Instruction(ObjectCode objCode, InstructionToken token) {  1 usage
    InstructionToken.TextInfo textInfo = token.getTextInfo(symbolTable, literalTable);
    objCode.addText(token.getAddress(), textInfo.generateText());

    // todo modification
}
```

  따라서 Pass 2 단계에서는 간단히 `TextInfo` 객체를 사용하여 텍스트를 넣어주고 있습니다.

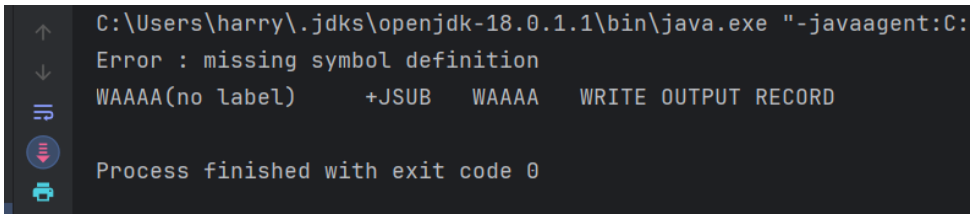
- **Error Handling**

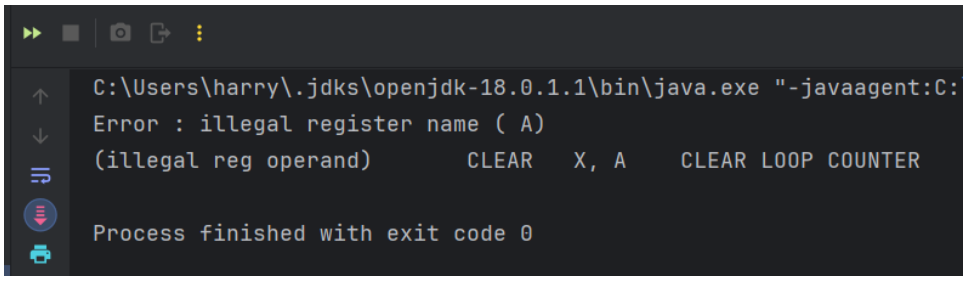
  마지막으로 에러 처리에 관한 내용입니다. 파싱 도중 에러가 발생하면 RuntimeException을 던지고, catch하는 부분은 Assembler의 main 함수에서 받아 콘솔에 출력해주고 있습니다.

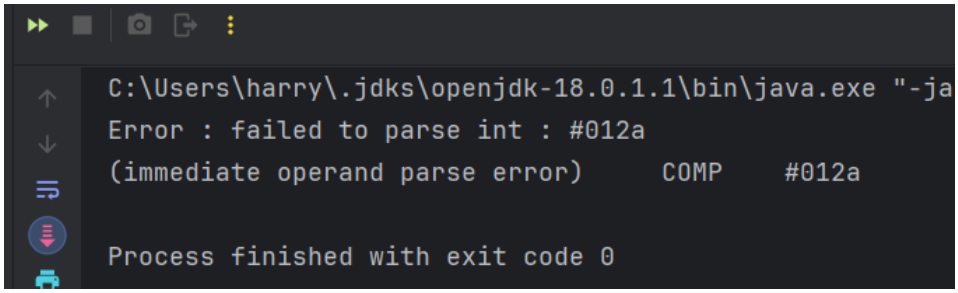
  과제에서 요구하지 않았지만 추가적으로 에러 처리한 경우들은 다음과 같습니다.

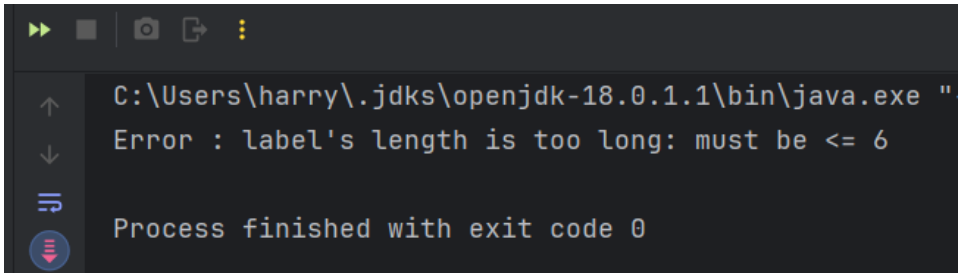
  - immediate 피연산자 파싱 에러
  - 리터럴의 주소가 아직 정의되지 않은 경우 에러
  - 리터럴 테이블, 심볼 테이블 검색 에러
  - 2형식 명령어일 때 피연산자의 개수가 맞지 않는 에러
  - EQU에 수식이 포함되어 있지 않은 경우
  - 레이블의 길이가 6 초과인 경우

- 심볼이 중복된 경우
- 레이블 없이 START, CSECT, BYTE, WORD, RESB, RESW, EQU 지시어가 사용된 경우









3. **수행 결과**

output_symtab.txt

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

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

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

심볼 테이블이 HashMap 자료구조로 구현되어 있어서 정렬이 제각각인 것을 볼 수 있습니다.

output_littab.txt

```
=C'EOF' 0x0030



=X'05'  0x001C
```

output_objectcode.txt

```
HCOPY  000000001033
RRDREC WRREC
DBUFFER000033BUFEND001033LENGTH00002D
T00000000003C5000272010000000023A10000C000072010000F03FFFEC002016300016
T00001D00001A01000330000A20100000F20000
T000030000006454F46
T000033000000
E000000

HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T00000000003C340434003400D0001F80001BC03FFFFA6000152000C00009509000003804
T00001D00001EE03FFFE940100000300000F1000000
```

```
E

HWRREC 00000000001D
RLENGTHBUFFER
T00000000003A3404D0100000802013C03FFFFA40900000702009380 4E03FFFEE300000
T00001D0000040505
E
```

instruction의 operand 처리를 아직 구현하지 못하여 text record의 결과가 제대로 나오지 않고 있습니다. 또한 modification record가 미비한 상태입니다.

4. 결론 및 보충할 점

- 결론

  문제를 카테고리화 하여 객체지향적으로 접근하고 보니 C언어로 구현했을 때보다 훨씬 이해하기 쉽고 디버깅하기 용이한 코드를 구성할 수 있었습니다. 형식적인 입력만 들어오는데도 처리해주어야 하는 것이 꽤 많은데, 좀 더 유연한 형식의 입력을 지원하기 위해서는 처리해주어야 할 것이 훨씬 많을 것 같다는 생각이 들었습니다.

- 보충할 점

  앞서 언급한 것처럼 Operand 클래스를 활용하여 명령어의 피연산자를 범주화하고 싶습니다. 또한 modification record를 이번에 구현하지 못했는데, 이 부분도 처리를 마저 진행하고 싶습니다.