**과목명: 시스템프로그래밍**

**2분반**

**<<Project #2>>**

**서강대학교 [컴퓨터공학과]**

**[20171666]**

**[이예은]**

목 차

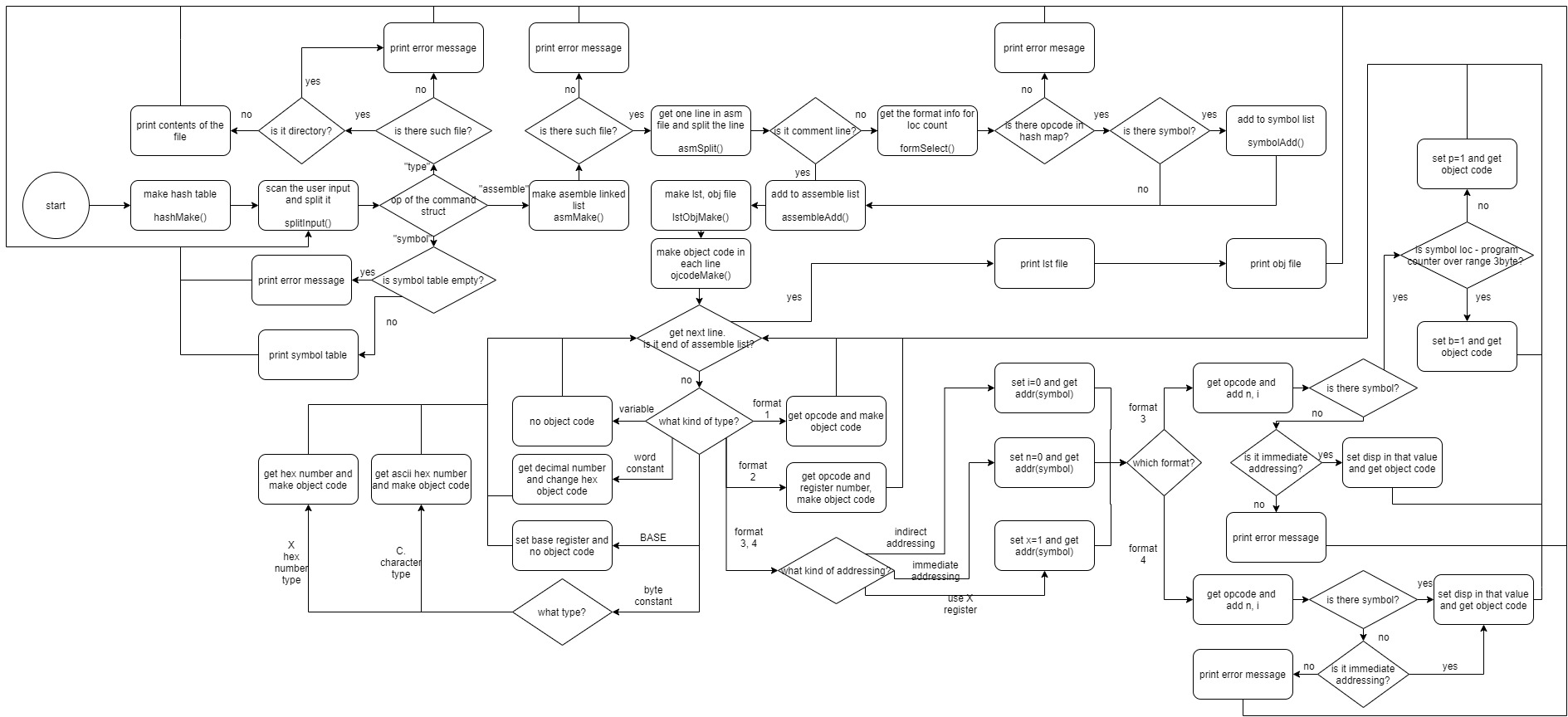
1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. 정의한 모듈에 대해 간략적인 설명
4. **전역 변수 정의**
5. **코드 설명**
6. **프로그램 개요**

이 프로그램은 지난 프로젝트 1에서 만든 SIC/XE 머신에 추가로 assembler를 구현한 프로그램입니다. 추가된 명령어는 type, assemble, symbol입니다. Asm 파일을 입력 받아 obj, lst 파일을 생성하고, assemble 과정 중에 생성된 symbol table과 obj, lst 파일을 볼 수 있는 기능을 수행합니다. 이전 프로젝트와 마찬가지로 사용자가 명령을 입력하면 그에 따른 결과를 보여주는 방식으로 구현된 프로젝트입니다.

1. **프로그램 설명**

프로그램의 전반적인 동작은 이전 프로젝트와 같으며, assemble의 경우 크게 2개의 pass로 이루어집니다. 1번째 pass에서는 symbol table과 assemble list를 만들고 2번째 pass에서 object code를 생성해 lst, obj를 만드는 과정으로 이루어집니다. 이전과 마찬가지로 사용자가 quit 명령을 입력하기 전까지 프로그램이 실행됩니다.

* 1. **프로그램 흐름도**

이전 프로젝트에서 추가한 부분만 흐름도를 만들었습니다.

1. **모듈 정의**

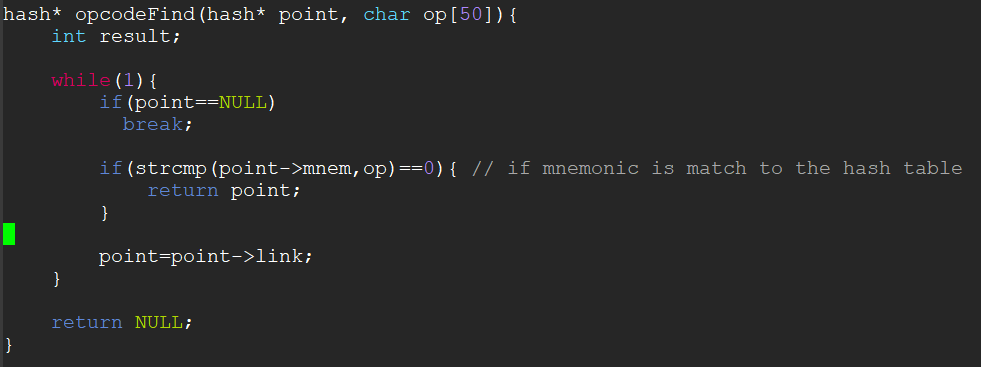
추가 및 수정된 함수만 설명하였습니다.

|  |  |
| --- | --- |
| Hash\* opcodeFind(hash\*, char[]) | Hash table의 시작점 node와 opcode 문자열을 받아 hash table 내에 opcode가 있는지 검사합니다. 있다면 그 포인터를 넘겨주고 없다면 null을 반환합니다. |
| Void orType(command, char[]) | Command 구조체와 유저입력을 인자로 받아 입력 받은 파일을 열어 출력해줍니다. 입력 받은 파일이 없거나 directory 이름이라면 에러메세지를 출력합니다. |
| Void orAssemble(command, char[]) | Command 구조체와 유저입력을 인자로 받아 입력 받은 asm 파일을 assemble 해줍니다. asmMake 함수를 불러 assemble list와 symbol table을 만들고 lstObjMake를 불러 lst 파일과 obj 파일을 만들어줍니다. 중간에 에러가 발생하면 에러메세지를 출력합니다. |
| Void orSymbol() | 저장되어 있는 symbol table을 출력해줍니다. Symbol table이 없다면 에러메세지를 출력합니다. |
| Int asmMake(char[]) | 파일 이름을 인자로 받아 해당 asm 파일을 열어 한 줄씩 입력 받고, 이를 쪼개 state, mnem, addr을 얻고 formSelect를 불러 loc을 계산합니다. 이후 이 정보를 이용해 assemble list와 symbol table을 만들어줍니다. 파일이 없거나 list에 추가 중 에러가 발생하면 발생한 line number를 반환합니다. 에러가 발생하지 않았다면 -1을 반환합니다. |
| Int formSelect(char[], char[]) | Mnemonic과 addr을 인자로 받아 loc 계산을 위해 몇 바이트가 필요한지 계산하고 이를 반환해줍니다. |
| Int asmSplit(char[], char[], char[], char[]) | 스캔한 한 줄 문자열, state, mnem, addr을 인자로 받아 한 줄 문자열을 쪼개 각각 state, mnem, addr에 맞게 넣어줍니다. 주석 문장이면 0, state(symbol)가 없으면 1, 있으면 2를 반환합니다. |
| Void assembleAdd(int, char[], char[], char[]) | Assemble list node에 필요한 loc, state, mnem, addr을 받아 새로운 노드를 만들고 이를 list에 추가해줍니다. |
| Int symbolAdd(int, char[]) | Symbol list node에 필요한 loc, state를 받아 새로운 노드를 만들고 이를 symbol list에 추가해줍니다. 이 때, insertion sort를 이용하여 symbol table이 정렬되게 합니다. 중간에 같은 symbol이 존재한다면 에러 발생으로 0을 반환하고, 추가가 잘 되었다면 1을 반환합니다. |
| Int lstObjMake(char[]) | 파일 이름을 인자로 받아 먼저 만들어진 assemble list를 이용해 ojcodeMake를 불러 object code를 계산 후 assemble list에 추가합니다. 계산 중 에러가 발생했다면 그때의 line number를 반환합니다. 이후 인자로 받은 파일 이름을 이용해 lst, obj 파일을 만들고 만들어진 object code를 이용해 형식에 맞게 파일에 출력해줍니다. 에러가 발생하지 않았다면 0을 반환합니다. |
| Int ojcodeMake(assem\*) | Assemble list의 한 노드를 받아 그 때의 object code를 계산해줍니다. Object code를 만들기 위해 symbol table을 순회할 때 없는 symbol을 이용하는 등의 에러가 발생한 경우 1을 반환하고 성공적으로 object code를 만들었다면 0을 반환합니다. |
| Int registerFind(char) | Char 타입의 문자를 받아 register에 따라 맞는 register number를 반환해줍니다. |
| Symb\* symbolFind(char[]) | State(symbol)을 인자로 받아 symbol table을 순회하며 symbol이 존재하는지 확인합니다. 있다면 그 때의 포인터를 반환하고 없다면 null을 반환합니다. |
| Void assembleDelete() | Assemble list를 free 해주고 다시 초기화해줍니다. |
| Void symbolDelete() | Symbol list를 free 해주고 다시 초기화 해줍니다. |
| Int main() | 이전 main에 더해 type, assemble, symbol 명령어를 추가해 명령어를 수행하도록 합니다. 또한 추가된 assemble list, symbol table을 초기화해줍니다. |

1. **전역 변수 정의**

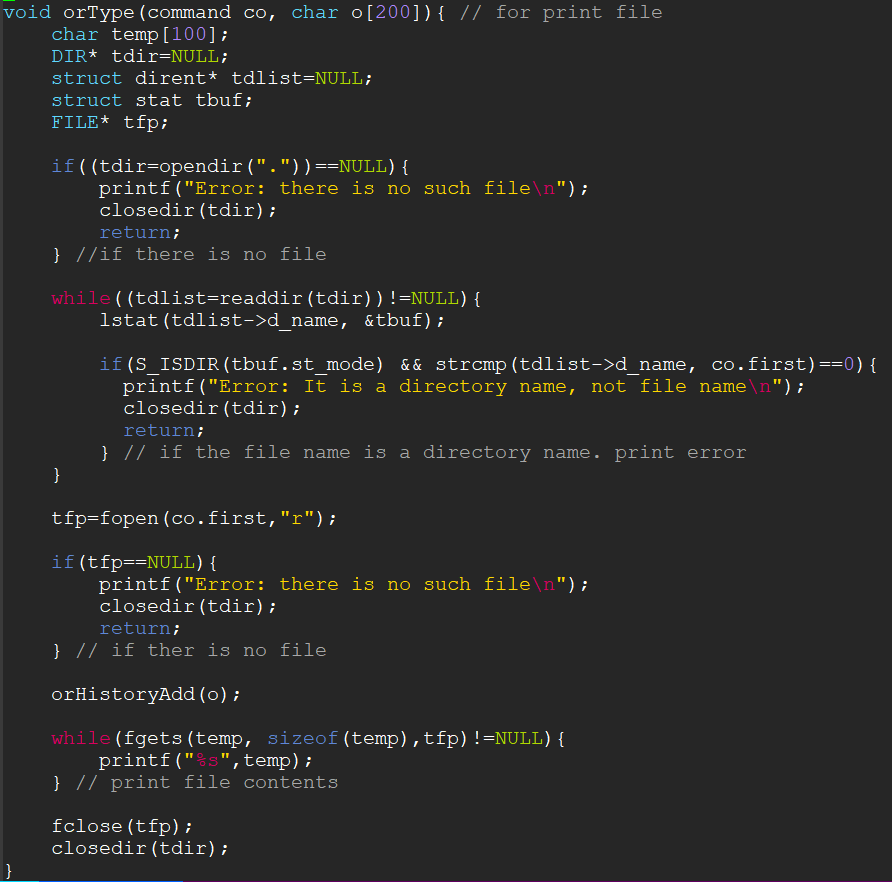
|  |  |
| --- | --- |
| Assem | Assemble list를 만들기 위한 구조체. Location counter를 저장하는 loc, object code의 길이와 object code를 저장하는 length, ojcode가 있고, state(symbol), mnem(opcode), address(oprand)를 저장하는 문자열이 있다. 또한 다음 node를 가리키는 link가 있다. |
| Symb | Symbol table을 위한 구조체. Location counter를 저장하는 loc, symbol을 저장하는 state, 다음 node를 가리키는 link가 존재한다. |
| objPrint | Obj 파일을 만들기 위한 구조체. 출력하려는 object code의 길이인 length와 object code를 저장하는 ojcode가 존재한다. |
| Int base | BASE register의 값을 저장하기 위한 전역변수이다. |
| Symb\* sPresent | 현재 assemble 중에 생성되는 symbol table이다. |
| Symb\* sSaved | Assemble이 성공적으로 되었을 때, 저장되는 symbol table이다. 때문에 assemble 중간에 에러가 발생하더라도 이전 성공한 assemble의 symbol table이 저장되어 있다. |
| Assem\* Ast | Assemble list의 시작 포인터이다. |
| Assem\* Aed | Assemble list의 끝 포인터이다. |

1. **코드 설명**
   1. **opcodeFind**



탐색하고 싶은 인덱스의 hash table 시작 포인터와 opcode를 찾고자 하는 것의 문자열을 인자로 받는다. 이 인덱스의 hash table을 돌면서 인자로 받은 op와 같은 mnem이 있는지 확인한다. 만약 mnem이 같다면 찾고자 하는 opcode이므로 그 포인터를 반환한다. 만약 hash table의 끝까지 탐색해도 찾지 못했다면 NULL을 반환한다.

* 1. **orType**

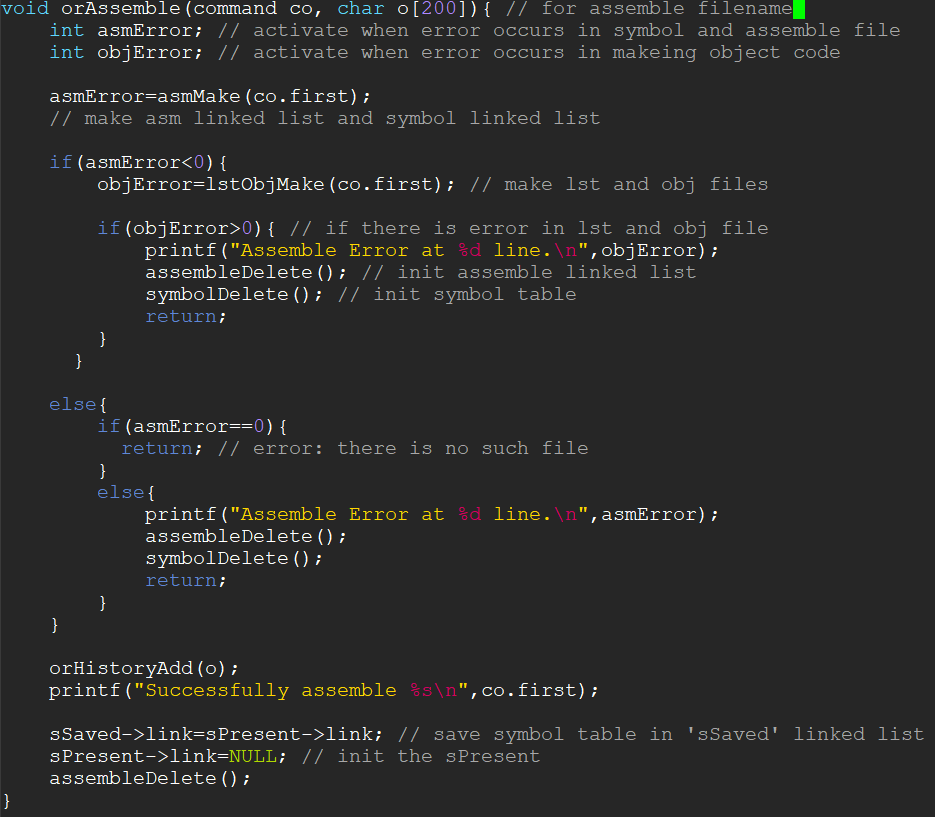


Command 구조체와 유저의 입력인 o를 인자로 받는다. 먼저 opendir로 현재 위치의 파일을 확인하고 현재 위치에 아무것도 없다면 에러 메세지와 함께 함수를 종료한다.

아니라면 먼저 입력받은 파일 이름이 directory name인지 확인한다. Readdir로 현재 위치의 파일 이름을 확인하며, 파일 이름이 유저의 입력과 같고, 그 때 파일을 S\_ISDIR로 검사했을 때 directory라면 에러 메시지와 함께 함수를 종료한다.

Directory도 아닌 것으로 확인됐으면 fopen을 이용해 파일을 read mode로 연다. 이후 EOF에 도달할 때까지 한 줄씩 파일을 읽어와 출력해준다. 이후 fclose와 closedir로 파일과 디렉토리를 닫아준다.

* 1. **orAssemble**

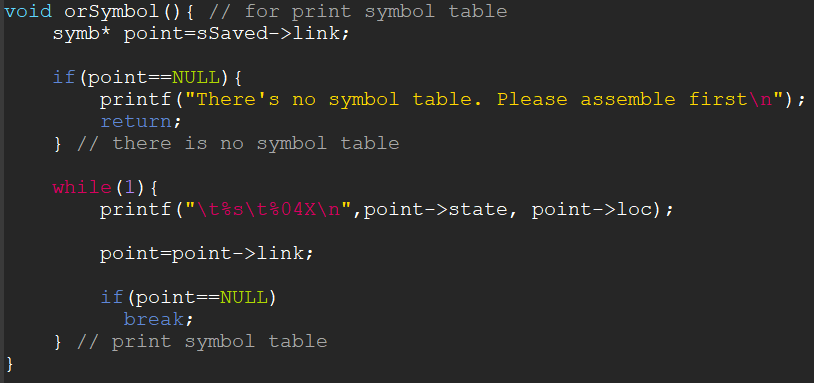


Command 구조체와 유저의 입력인 o를 인자로 받는다. 먼저 asmMake 함수를 불러 assemble list와 symbol table을 만들어준다. 이 때 asmMake의 반환값이 0이면 없는 파일을 연 것이고, 0보다 크면 오류가 있는 line의 number를 반환한 것이므로 에러 메시지와 함께 함수를 종료한다. 여기까지가 pass 1이다.

0 미만이라면 에러가 나지 않았으므로 lstObjMake 함수를 불러 lst 파일과 obj 파일을 만들어준다. 만약 lstObjMake의 반환값이 0보다 크면 또한 파일을 만들던 중 에러가 난 것이므로 에러 메시지와 함께 함수를 종료한다.

0 미만이라면 성공적으로 assemble 되었으므로 history list에 추가해주고 현재 symbol table인 sPresent를 sSaved에 연결해 저장되도록 한다. 이후 sPresent는 새로운 symbol을 만들어야 하므로 NULL로 초기화 해주고 새로운 assemble list를 받기 위해 assembleDelete를 불러 assemble list 또한 초기화 해준다. 여기까지 pass 2가 모두 수행된다.

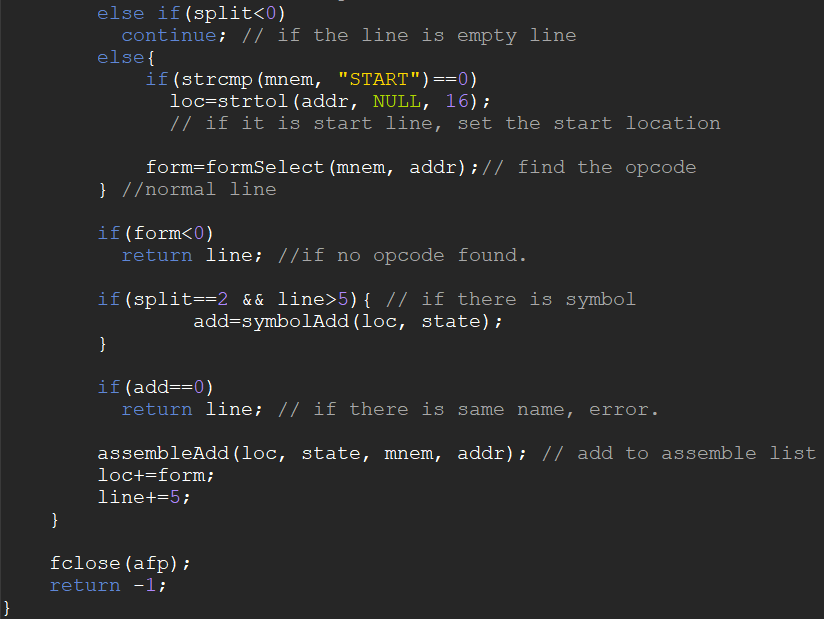
* 1. **orSymbol**



저장되어 있는 symbol table, sSaved의 처음부터 돌며 symbol table을 출력해준다. 만약 sSaved가 NULL로 초기화 되어 있다면 아직 symbol이 만들어지지 않았으므로 에러 메세지와 함께 함수를 종료한다.

* 1. **asmMake**





파일 이름을 인자로 받아 fopen으로 파일을 연다. 만약 파일이 없다면 에러 메시지와 함께 0을 반환한다.

파일의 끝인 EOF에 다다를 때까지 아래 과정을 반복한다. 먼저 파일의 한 줄을 입력받는다. 이후 state, mnem, addr을 초기화 하고 asmSplit 함수를 불러 문자열을 쪼개준다. asmSplit이 0을 반환한다면 주석 문장이므로 assembleAdd로 assemble list에 추가만 해준 후 다음 줄로 넘어간다.

0 미만의 값을 반환한다면 엔터만 있는 등의 빈 문장이므로 받지 않고 바로 다음 줄로 넘어간다.

0 초과라면 알맞게 쪼개진 것이다. 이후 현재 한 줄이 START 줄인지 확인한다. START줄이라면 loc를 START 이후 addr에 나온 시작 주소로 초기화해준다.

formSelect 함수를 불러 몇 형식인지, 즉 loc 연산 시 얼마의 바이트를 사용하는 지 구한다. formSelect의 반환값이 0보다 작으면 에러가 발생한 것이므로 현재의 line number를 반환하고 함수를 종료한다.

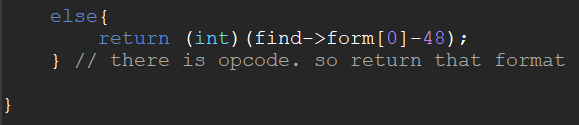
split했을 때 반환 값이 2이고 START 줄이 아니라면 symbol이 있는 줄로 symbolAdd를 불러 symbol table에 추가해준다. symbolAdd의 반환 값이 0이라면 추가 중 에러가 발생한 것이므로 현재 line number를 반환해주고 함수를 종료한다.

모든 선별 과정이 끝나면 assembleAdd를 불러 현재 줄의 정보를 assemble list에 추가해주고 loc와 line을 업데이트 해준다.

이 모든 과정이 중간에 종료되지 않고 끝났다면 에러 없이 assemble list와 symbol table이 완성된 것이므로 fclose로 파일을 닫고 -1을 반환해준다.

* 1. **formSelect**





현재 줄의 mnem과 addr을 인자로 받는다. 만약 mnem이 +로 시작한다면 4형식이므로 4를 반환한다.

먼저 opcodeFind를 불러 mnem의 opcode가 있는지 확인한다.

opcodeFind의 반환 값이 존재하면 opcode를 찾은 것이므로 그 때 저장된 format ascii 코드 값에서 48을 뺀 값, 즉 숫자 값을 반환한다.

opcodeFind의 반환 값이 NULL이라면 opcode가 없는 것이므로 어느 경우인지 이제 확인한다. BASE이면 유저에게 알려주기 위한 줄이므로 loc연산이 필요없다. 그러므로 0을 반환한다.

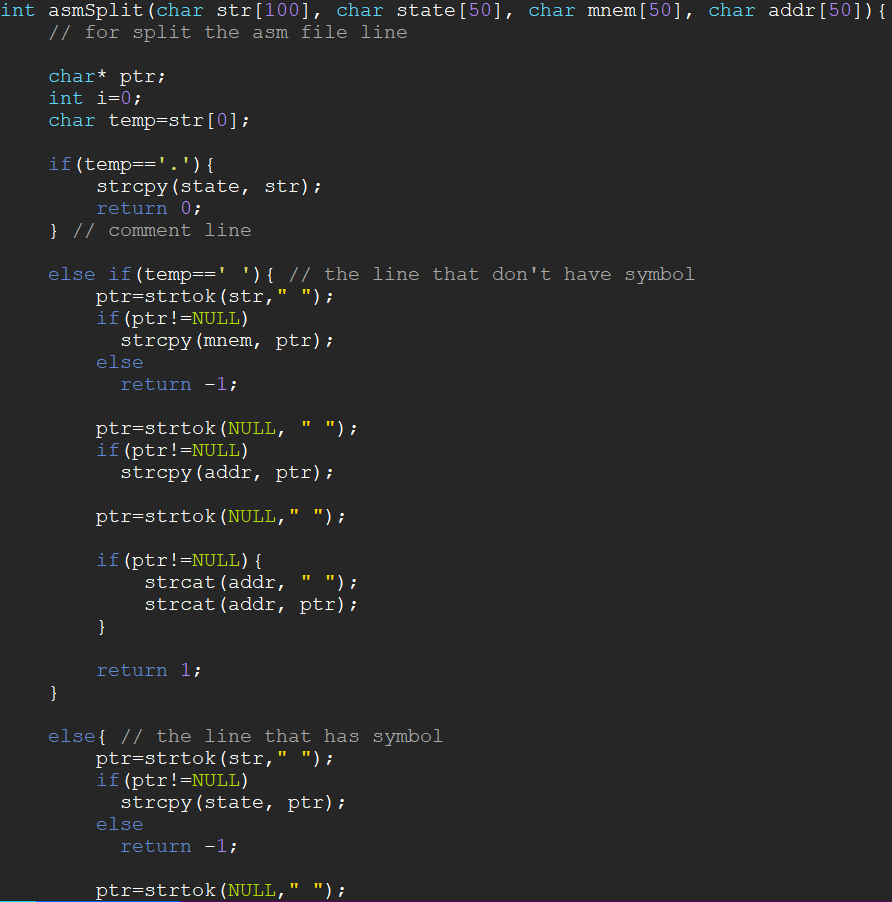
BYTE이면 byte 상수이다. C로 시작한다면 char 형 상수이므로 문자 하나 당 1바이트가 필요하다. 그러므로 C’\_\_’의 중간 문자열의 길이를 반환해준다. X로 시작한다면 hex 수이고 1~2개는 1바이트, 3~4는 2바이트 식으로 바이트가 필요하므로 X’\_\_’의 중간 문자열의 길이/2 + 중간 문자열의 길이%2를 반환해준다.

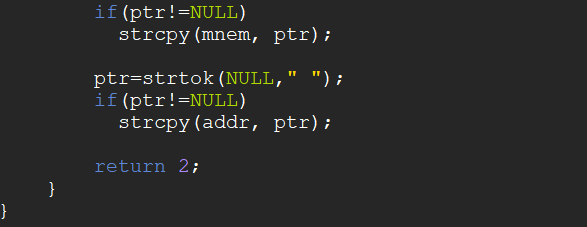
WORD면 word 상수로 3바이트가 필요하므로 3을 반환한다.

RESB는 byte형 변수로 입력 받은 숫자만큼의 byte가 필요하므로 이를 반환한다.

RESW는 word형 변수로 입력 받은 숫자\*3byte 가 필요하므로 이를 반환한다.

* 1. **asmSplit**





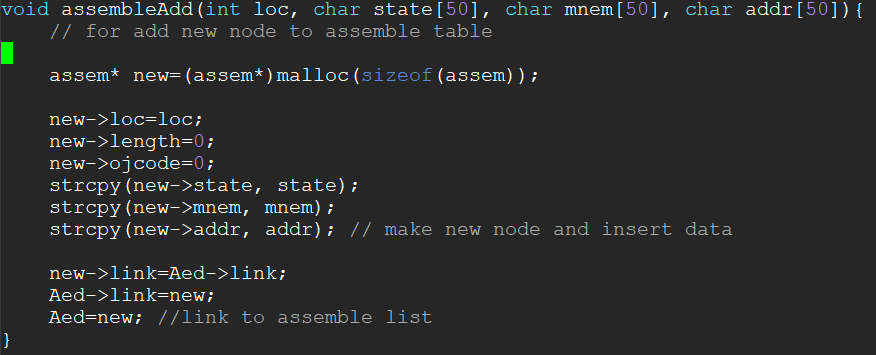
쪼갤 문자열과 쪼갠 결과를 넣을 state, mnem, addr을 인자로 받는다. Str의 가장 처음을 기준으로 경우를 나눈다.

가장 첫 문자가 ‘.’이라면 주석 문장이므로 state에 문장 전체를 넣고 0을 반환한다.

가장 첫 문자가 ‘ ‘라면 state(symbol)이 없는 줄이므로 두 번 쪼개 mnem과 addr에 넣어준다. 만약 처음 쪼갰을 때 문자열이 안 나온다면 아예 빈 문장이므로 -1을 반환한다. 두 번 쪼갠 후 한번 더 쪼갰을 때 NULL이 아니라면 ,X 등 다른 register를 쓴 문장이므로 addr의 끝에 추가해준다. 이후 symbol이 없는 문장이라는 의미로 1을 반환한다.

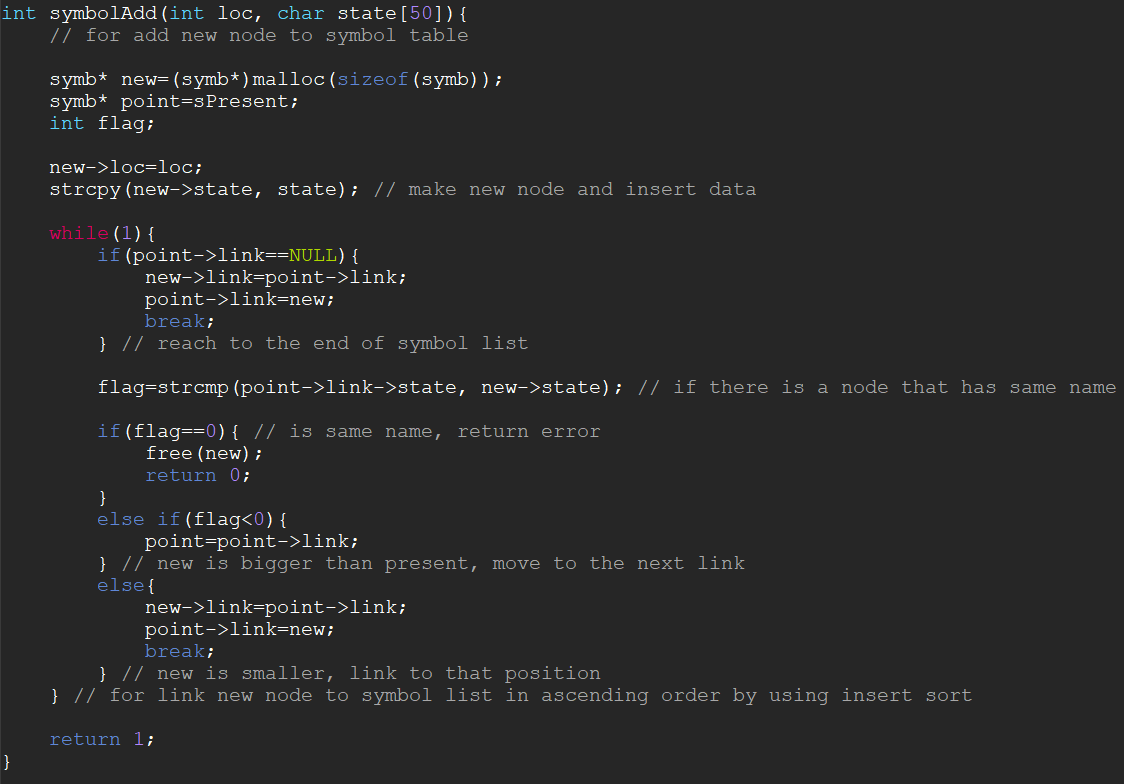
‘.’과 ‘ ‘가 아니라면 state(symbol)이 있는 줄이므로 3번 쪼갠다. 처음 쪼갰을 때 문자열이 안 나온다면 또한 잘못된 문장이므로 -1을 반환한다. 이렇게 3번 쪼개서 각각 state, mnem, addr에 넣어준다. 이후 symbol이 있는 문장이라는 의미로 2를 반환한다.

* 1. **assembleAdd**



새로운 assem 노드를 생성하고 인자로 받은 정보들을 넣어준다. 이 때 length와 ojcode는 아직 모르므로 0으로 초기화해준다. 이후 Aed, assemble list의 끝에 추가해준다.

* 1. **symbolAdd**



새로운 Symb 노드를 생성하고 인자로 받은 정보들을 넣어준다. Symbol table이 정렬되어 있어야 하므로 넣을 때 insertion sort의 방법을 활용해 넣어준다.

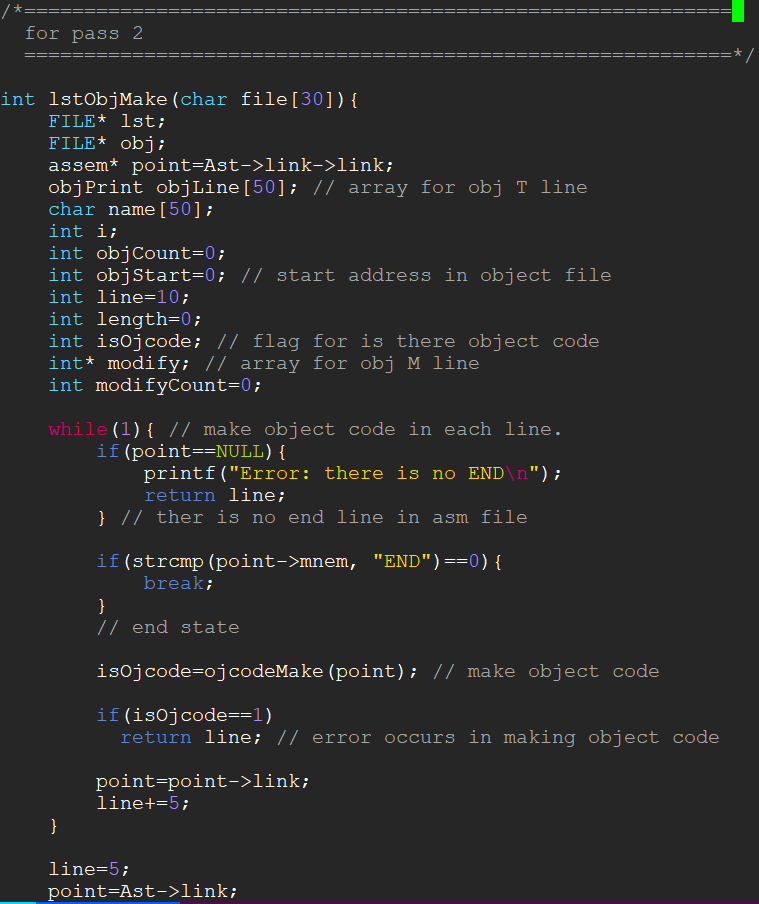
Point는 현재 만들고 있는 symbol table, sPresent를 가리키고 만약 point->link가 NULL이라면 symbol table의 마지막에 다다른 것이므로 끝에 새로운 노드를 넣어준다.

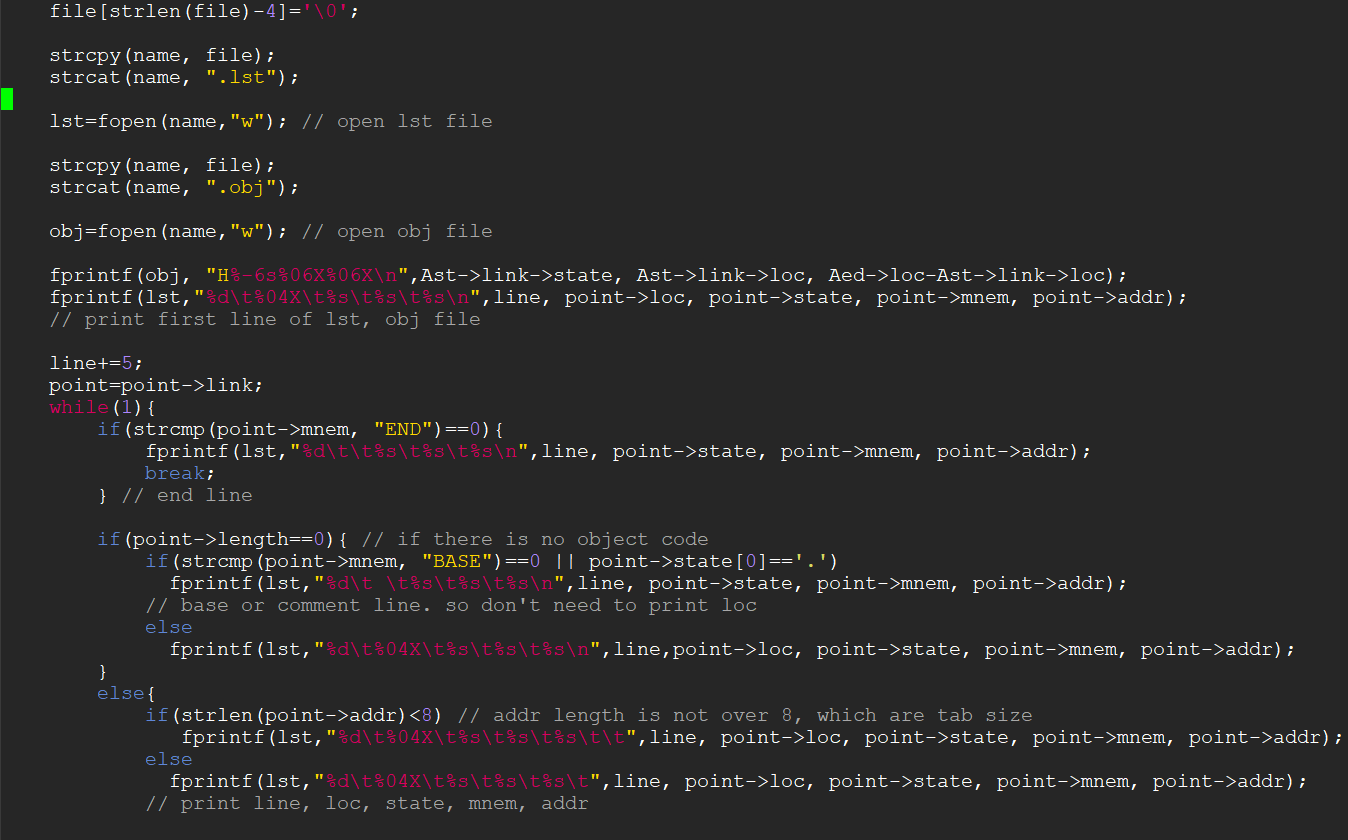
아니라면 현재 point가 가리키는 것의 state와 새로운 노드의 state를 비교한다. 둘이 같은 문자열이라면 같은 이름의 label이 사용된 것이므로 에러. 0을 반환해준다.

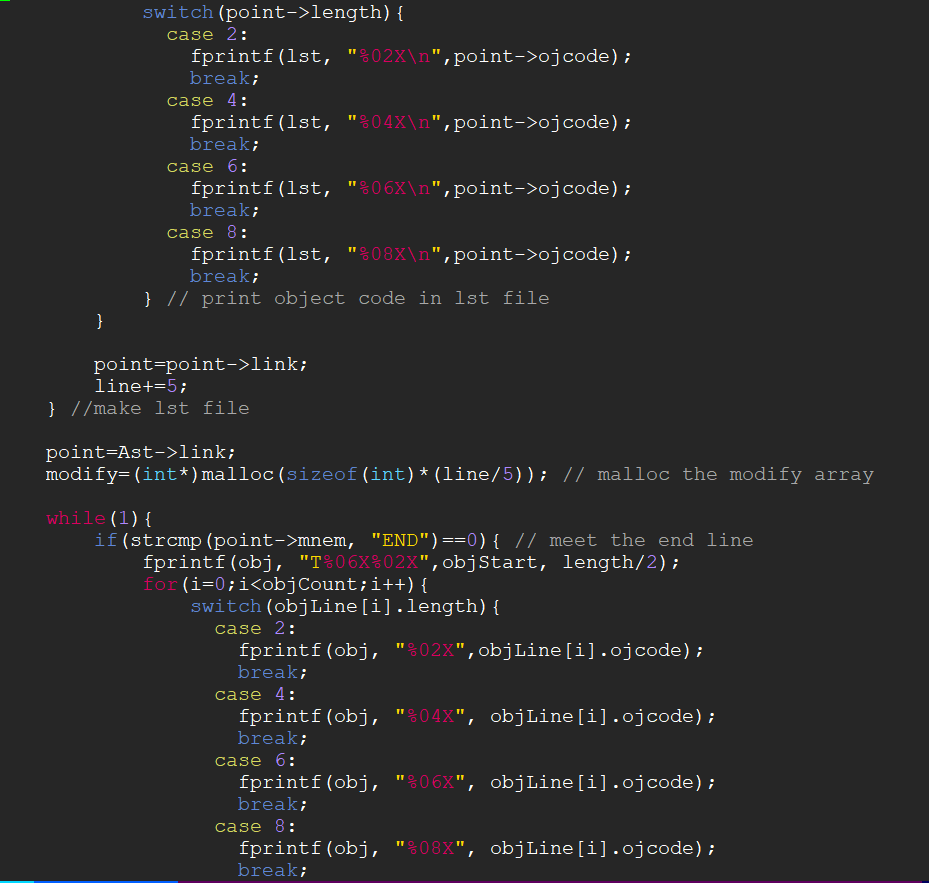
새로운 state가 더 크다면 다음 노드와 비교하도록 point를 옮겨준다. 새로운 state가 더 작다면 그곳이 새 노드가 들어가야 할 자리이므로 연결해준다.

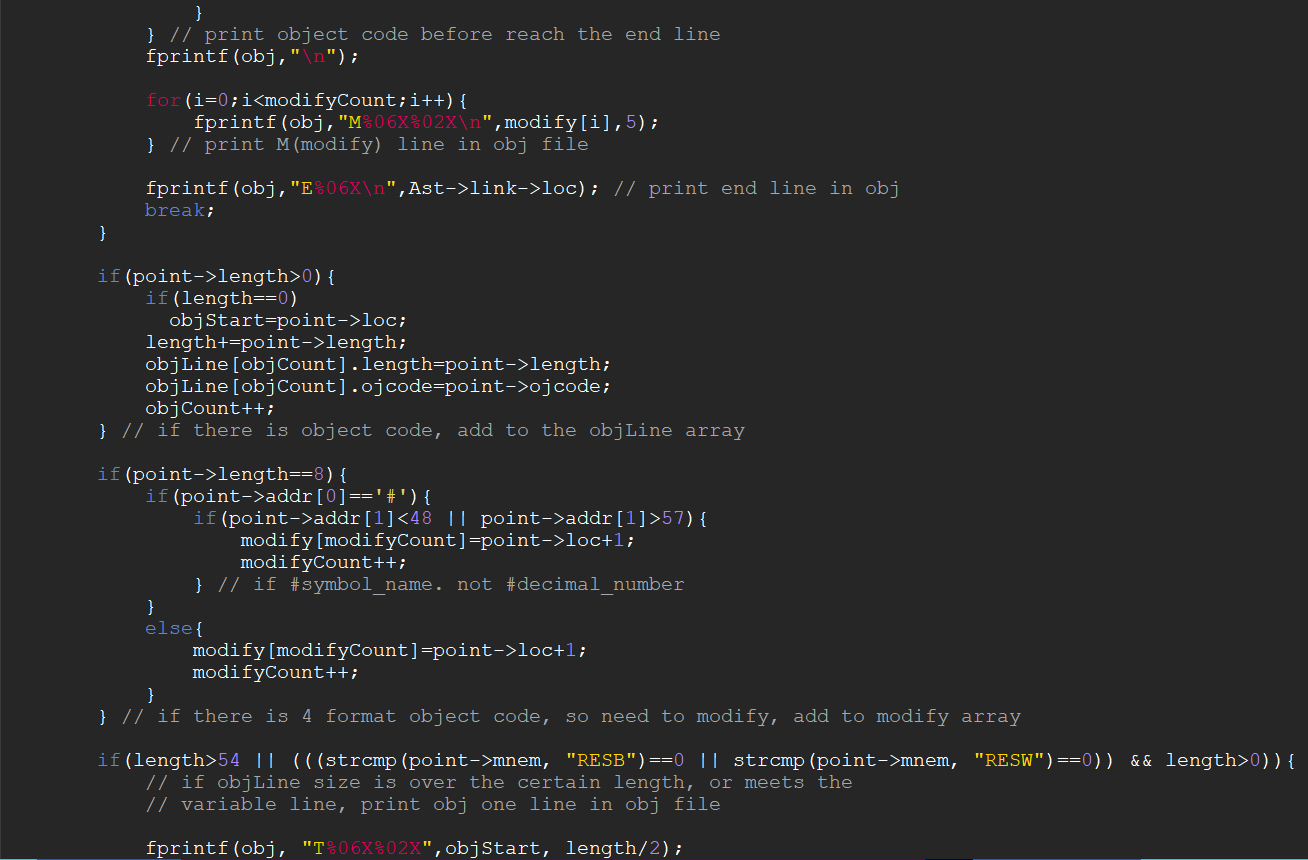
중간에 return을 안 만났다면 성공적으로 새 노드가 들어간 것이므로 1을 반환해준다.

* 1. **lstObjMake**











가장 먼저 END 줄에 다다를 때까지 assemble list를 돌며 ojcodeMake를 부른다. 이 때 object code가 만들어지고 이 함수의 반환 값이 1이면 object code 생성 중 에러 발생으로 현재 line number를 반환하고 함수를 끝낸다.

Object code를 모두 생성했으면 인자로 받은 파일 이름을 이용해 lst파일과 obj 파일을 생성하고 w 모드로 열어준다.

가장 먼저 obj 파일의 첫 번째 START 줄인 H 줄을 출력해준다. 이후 lst 파일의 첫 번째 줄인 START 줄 또한 출력해준다.

이후 lst 파일을 출력하는 과정을 거친다. Mnem이 END줄에 갈 때까지 아래 과정을 반복한다.

Lst 파일 출력 시 먼저 object code가 있는지를 확인한다. object code가 없다면 object code의 길이인 length가 초기값인 0인 것을 이용해 구분한다. Length가 0이고 mnem이 BASE일 때에는 loc 값도 출력할 필요가 없으므로 나머지 line과 state, mnem, addr만 출력해준다.

나머지의 경우에는 object code를 뺀 나머지 부분을 출력해준다.

Object code가 존재할 때에는 먼저 line, loc, state, mnem, addr 까지만 출력한다. 이 때, addr의 길이가 8을 넘어간다면 출력 시 정렬이 망가지므로 이 때를 구분해서 탭을 넣어준다.

이후 length에 따라 자릿수를 맞춰서 object code를 출력한다. 이를 반복하다 END 줄을 만났을 때에는 loc과 object code이 없으므로 이를 뺀 나머지만 출력 후 반복문을 종료한다.

Obj 파일 또한 END를 만날 때까지 아래 과정을 반복한다. 이 때, objLine은 obj 파일의 T 한 줄을 저장하고 있는 objPrint 구조체 배열이다. objCount는 objLine의 인덱스, objStart는 T가 시작하는 object code의 loc를 저장하고, length는 T 한 줄의 길이를 저장하고 있다. Modify는 4형식에서 modification이 필요한 object code의 시작 주소를 저장하는 배열이고 modifyCount는 이 배열의 인덱스이다.

Assemble list의 length가 0보다 크면 object code가 있으므로 이 때 objLine에 그 때 object code의 길이와 object code를 넣고 objCount를 다음 인덱스로 넘겨준다. Length 또한 이 때의 length를 더해 T 한 줄의 object code들의 길이를 저장하게 한다. 이 때, length가 0이면 새로운 T 라인이 생긴 것이므로 현재의 loc를 objStart에 저장한다.

Assemble list의 length가 8이면 4형식으로 modify가 필요 할 수 있으므로 확인한다. 4형식이면서 #숫자 형식의 immediate addressing은 modify가 필요 없으므로 이 경우를 제외하고 modify 배열에 현재 loc+1 값을 저장한다. 이후 modifyCount를 갱신한다.

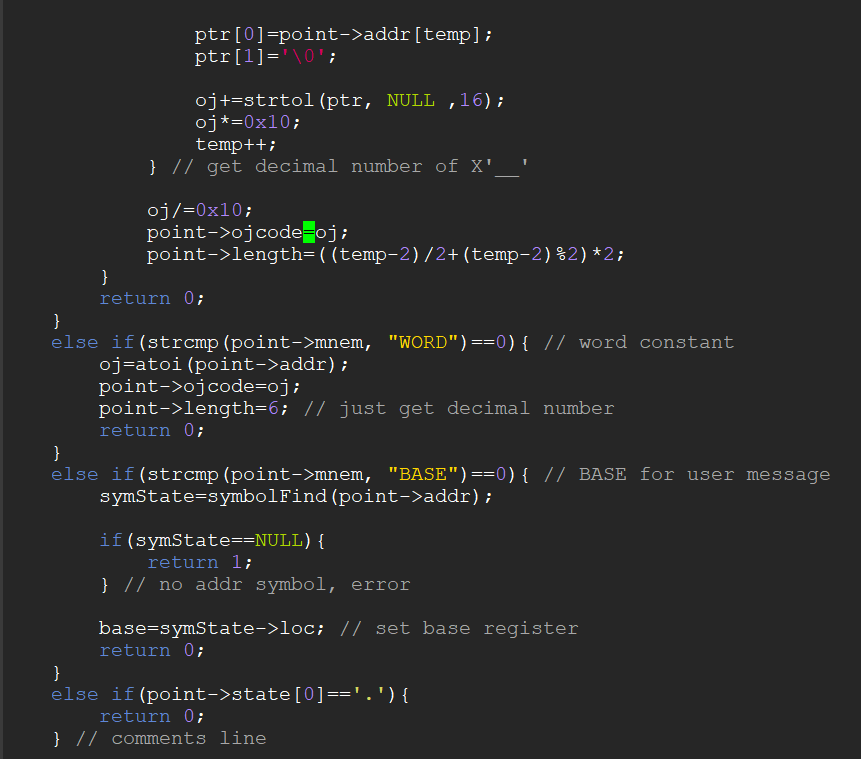
저장 후 만약 T 줄의 길이, length가 54를 넘거나, 변수를 만나면 출력해야 하므로 출력을 시작한다. 먼저 T와 시작 주소인 objStart, 길이를 바이트로 표현한 length/2를 출력한다. 이후 objLine을 돌면서 object code의 길이에 따라 자릿수를 맞춰 출력해준다. 출력이 끝났다면 다시 다음 T 줄을 받아야 하므로 objCount와 length를 초기화해준다

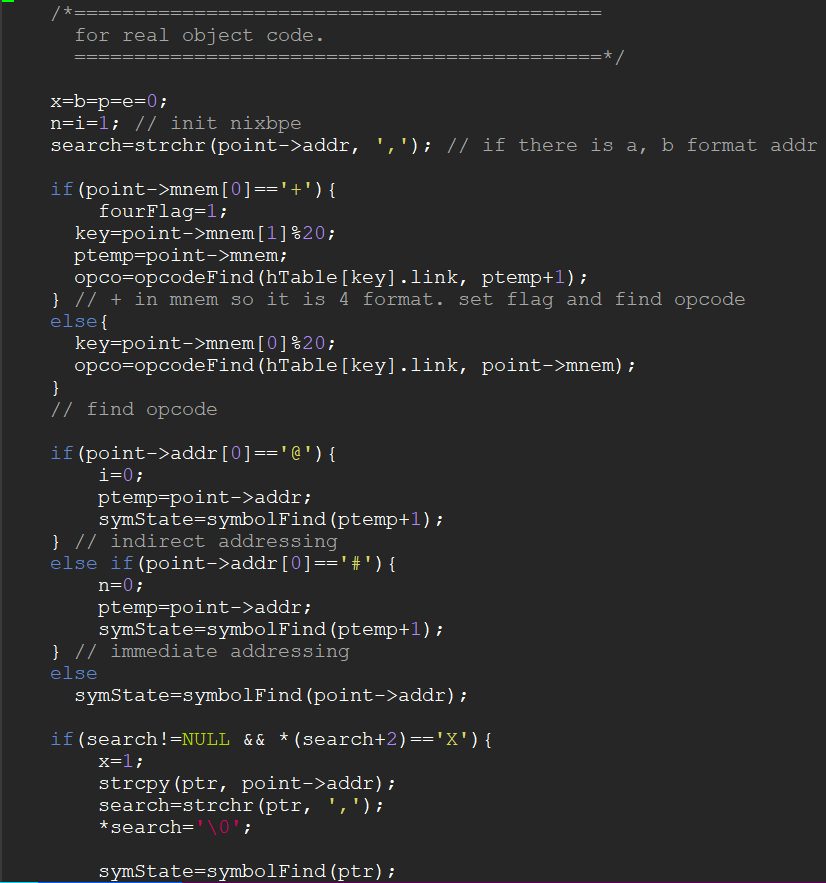
END를 만난다면, 위의 과정과 마찬가지로 이전까지 들어갔던 objLine을 출력한다. 이후 modify 배열을 돌면서 M 줄을 출력한다. 그리고 마지막 E줄을 출력하며 obj 작성을 마친다.

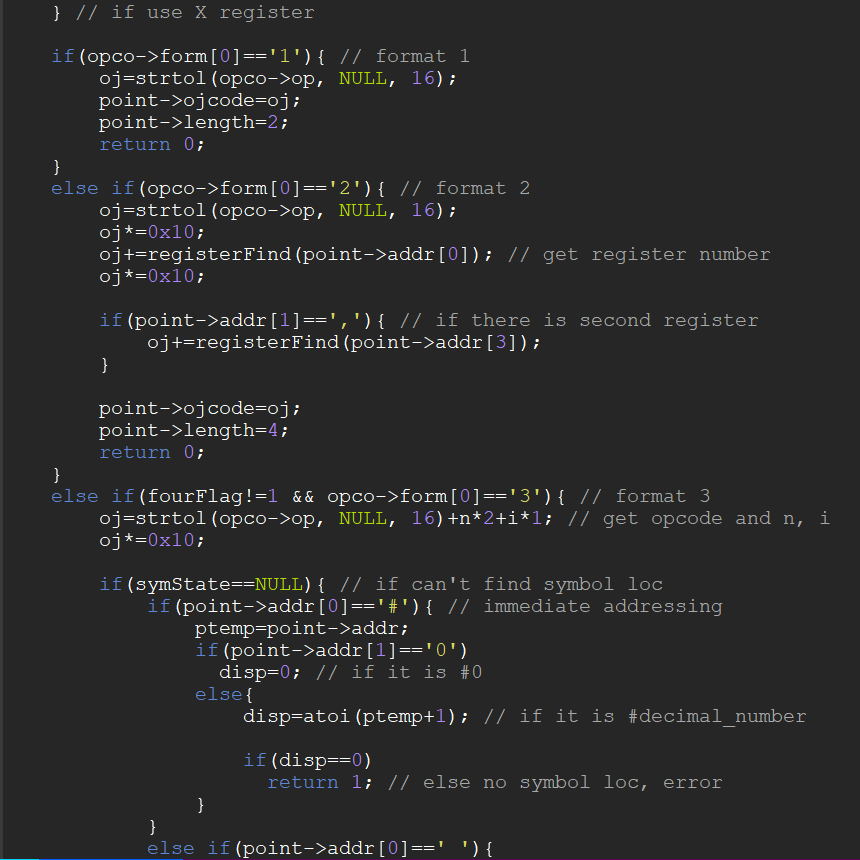
모든 출력을 마친 후 fclose로 각각의 파일을 닫고 0을 반환하며 함수를 종료한다.

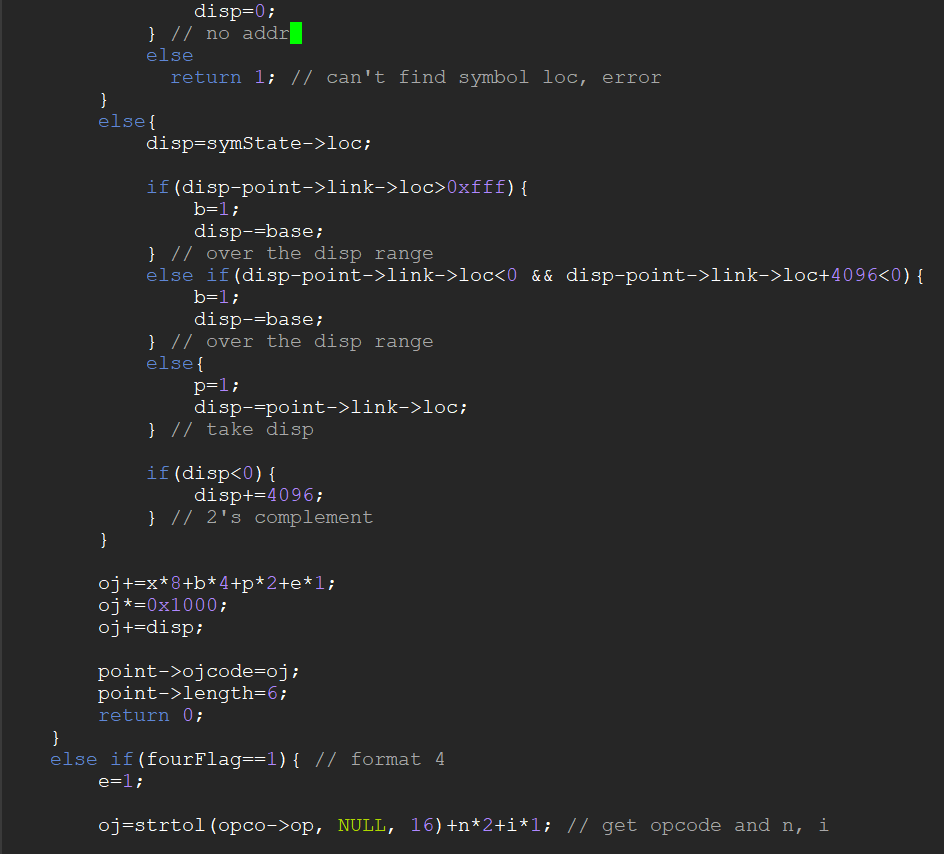
* 1. **ojcodeMake**

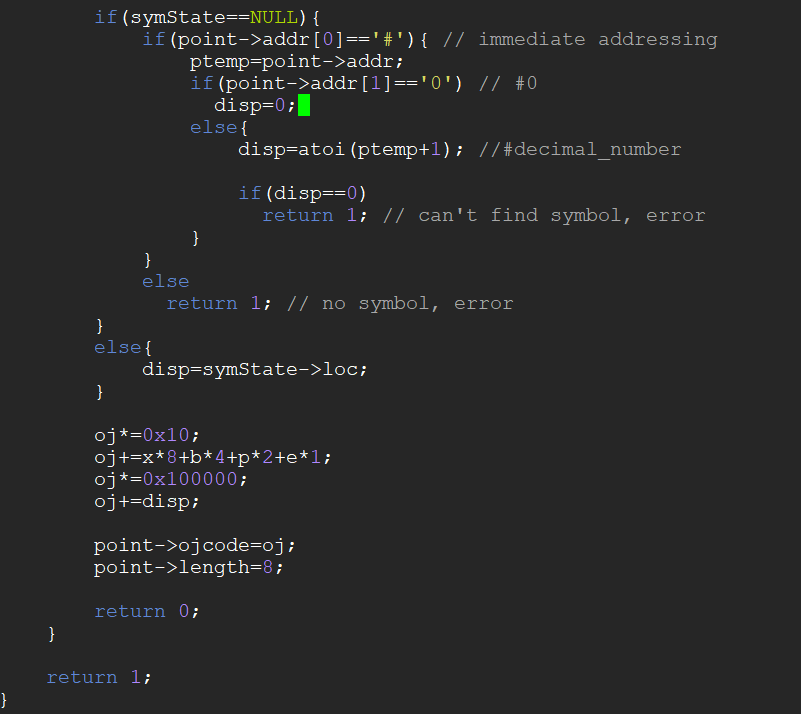












int형인 oj에 object code를 표현하기 위해서 다음과 같은 방식을 사용한다.

d

c

b

a

0

0

b

a

b

a

\*0x100 +cd

이런 식으로 16진수 연산 시 차지할 자릿수를 곱하고, 값을 더하는 방식을 반복한다. 이렇게 하면 int로 10진수 값을 넘기더라도 %X가 16진수로 바꿀 때 우리가 원하는 자릿수에 원하는 값이 들어갈 수 있게 된다.

먼저 format이 없는 특이 경우에 대해서 처리를 해준다. RESB나 RESW인 변수의 경우, object code가 없으므로 만들지 않고 그냥 0을 반환하고 종료한다.

BYTE 상수의 경우 또한 C와 X의 경우로 나뉜다. C의 경우 C’\_\_’ 내부 문자를 하나씩 ascii code 값으로 바꾼 후 각 아스키코드 당 2칸을 차지하므로 \*0x100+아스키값 을 반복한다. 총 길이는 (전체 문자열 길이의 -3)\*2이므로 이를 변환해 length와 얻은 object code를 넣는다.

X의 경우 X’\_\_’ 내부 hex 값을 하나씩 decimal값으로 변환 후 1칸씩 차지하므로 \*0x10+변환한 decimal값 을 반복한다. 이 때 총 길이는 ((전체 문자열 길이-3)/2+(전체 문자열 길이-3)%2)\*2 이므로 이를 변환해 length와 얻은 object code를 넣는다. 이후 0을 반환하고 함수를 종료한다.

WORD 상수의 경우 oj에 그냥 값을 넣어주고 3바이트 이므로 길이는 6으로 넣어준다. 그리고 0을 반환해 종료한다.

BASE의 경우 먼저 BASE 뒤의 addr이 있는 symbol인지 확인한다. 없다면 잘못된 명령이므로 1을 반환하고 종료한다. 있다면 그 symbol의 loc를 base에 넣어 base register 연산을 할 수 있게 해준다. 이후 0을 반환해 종료한다.

‘.’의 경우 주석문장이므로 object code가 없다. 그러므로 그냥 0을 반환해 종료한다.

특이 경우를 모두 선별했다면 이제 1,2,3,4 형식의 object code를 생성한다. 가장 먼저 x,b,p,e를 0으로, n,i를 1로 초기화 해준다. 그리고 형식에 맞는 object code를 생성하기 전, addressing 경우를 나눠 미리 opcode와 symbol을 얻는다.

먼저 mnem이 +로 시작할 경우 4형식이므로 플래그를 1로 활성화시킨 후, +이후의 문자열을 이용해 opcodeFind로 opcode를 얻는다.

나머지는 mnem 문자열을 이용해 opcodeFind로 opcode를 얻는다.

Addr이 @로 시작할 경우 indirect addressing으로 i를 0으로 세팅하고 @이후의 문자열로 symbolFind를 이용해 symbol을 얻는다.

Addr이 #으로 시작할 경우 immediate addressing으로 n을 0으로 세팅 후 #이후의 문자열로 symbolFind를 이용해 symbol을 얻는다.

나머지는 addr 문자열을 이용해 symbolFind로 symbol을 얻는다.

마지막으로 x register를 사용하는지를 확인한다. Addr 문자열에 ‘,’가 존재하고, addr의 마지막 문자가 X라면 x register를 사용하는 것이므로 x를 1로 세팅하고 ‘,’이전의 addr 문자열로 symbolFind를 이용해 symbol을 얻는다.

이제 형식에 따라 object code를 생성한다. 1형식의 경우 opcode만 필요하므로 이를 int로 바꿔 object code를 만들고, 길이인 2를 넣어 종료한다.

2형식의 경우 먼저 oj에 opcode를 int로 바꾼 값을 넣는다. 이후 registerFind를 이용해 addr의 register 번호를 얻는다. 1자리를 차지하므로 oj를 \*0x10+register번호 로 계산하고 2번째 register 번호를 넣을 자리인 \*0x10을 해준다. 만약 addr문자열에 ‘,’가 있어 2개의 register를 사용한다면, 한번 더 registerFind를 불러 번호를 얻고 이 번호를 oj에 더해 object code를 완성한다. 이후 길이인 4와 object code를 넣고 함수를 종료한다.

3형식의 경우 먼저 opcode와 뒤 2자리에 들어갈 n, i를 계산해 oj에 넣어준다. 이후 xbpe가 들어갈 한자리를 위해 \*0x10을 해준다. Symbol이 존재하는 경우와 아닌 경우로 먼저 검사를 진행한다.

Symbol이 없는 경우, ‘#숫자’ 형일때는 disp를 그 숫자로 설정하고 정상적으로 다음 단계로 넘어간다. #가 있지만 숫자가 아닌 경우와 #이 없는 경우는 symbol이 없는 비정상적인 assemble line이므로 에러 1을 반환한다. 하지만, RSUB처럼 addr이 없는 경우에는 disp를 0으로 세팅하고 다음단계로 넘어간다.

Symbol이 있는 경우 disp는 먼저 symbol의 loc으로 세팅해준다. 만약 disp-program counter가 3비트의 범위를 넘어가거나, 0보다 작아 2의 보수를 취했을때도 3비트 범위를 넘어간 경우 b를 1로 세팅하고 base register를 사용해 disp를 symbol loc-base register로 설정한다.

3비트 범위를 넘어가지 않았다면 p를 1로 세팅하고 disp를 symbol loc- program counter로 설정한다. 이렇게 disp를 얻은 후 만약 0보다 작다면 disp에 2의 보수를 취해 준다.

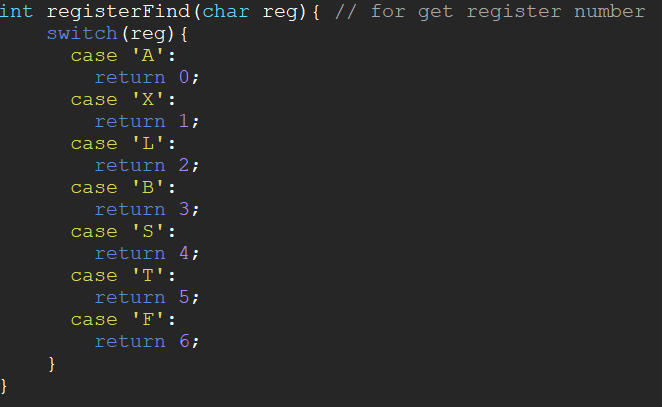
Nixbpe, disp를 모두 얻었으므로 oj에 xbpe의 값을 계산해 + 해준다. 이후 disp에 3자리가 필요하므로 \*0x1000을 해주고 disp를 더해 oj를 구한다. Object code와 길이 6을 넣어주고 0을 반환하며 종료한다.

4형식의 경우 먼저 e를 1로 세팅하고 opcode와 n, i를 계산해 oj에 넣는다. 3형식과 symbol이 없는 경우는 동일하게 disp를 얻는다.

Symbol이 있는 경우 program counter나 base를 계산하지 않고 그냥 symbol의 loc을 disp에 넣어준다. 이후 xbpe를 넣기 위해 \*0x10을 하고 xbpe 값을 계산해 + 해준다. 이후 disp 자리가 5자리 필요하므로 \*0x100000 후 disp를 더해준다. Object code와 길이 8을 넣어주고 0을 반환하며 종료한다.

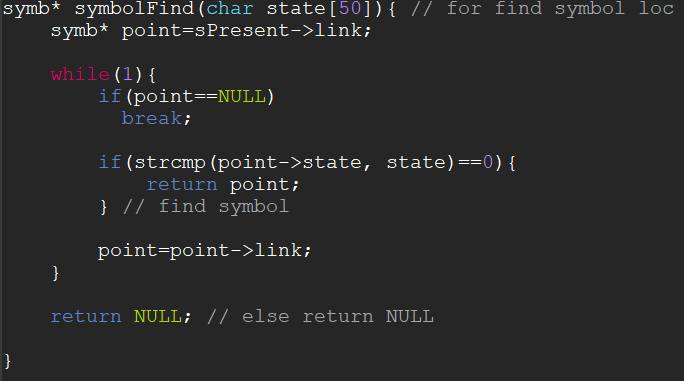
만약 이 모든 과정을 거쳤음에도 중간에 함수가 종료되지 않았다면, 어느 경우에도 걸리지 않는 예외의 명령이므로 1을 반환하고 종료한다.

* 1. **registerFind**



문자를 인자로 받아 그 문자가 어느 register인지 구분해준다. 각 register에 따른 register number를 반환해준다.

* 1. **symbolFind**

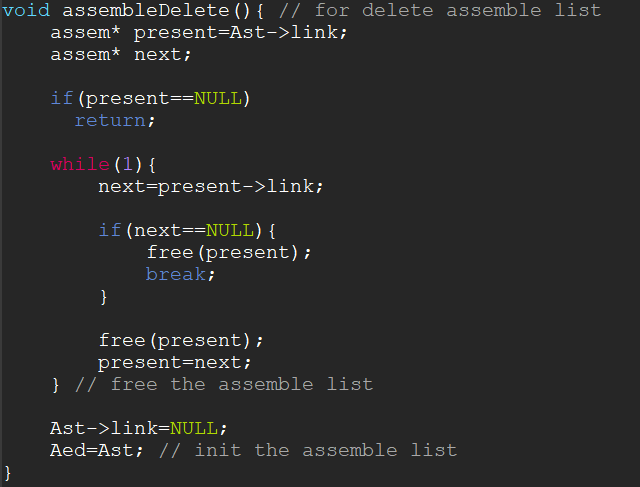


찾고자 하는 symbol을 인자로 받는다. 이후 현재 생성한 symbol table, sPresent의 처음부터 순회하며 state와 같은 symbol이 있는지 찾는다.

만약 symbol table의 끝에 도착했다면 찾지 못한 것이므로 NULL을 반환해준다.

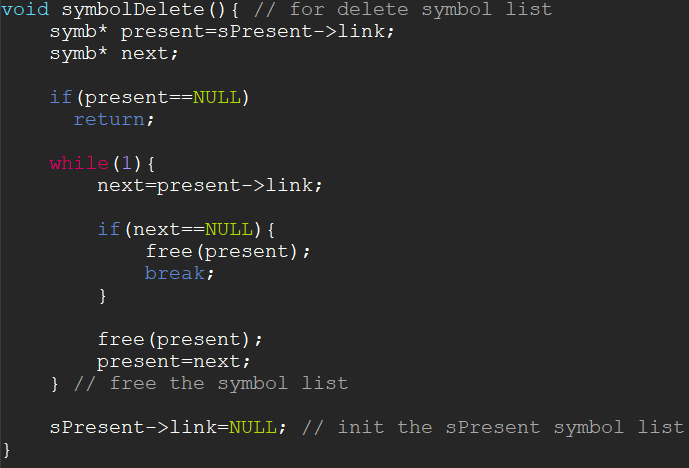
Symbol을 찾았다면 그 때의 포인터를 반환해준다.

* 1. **assembleDelete**



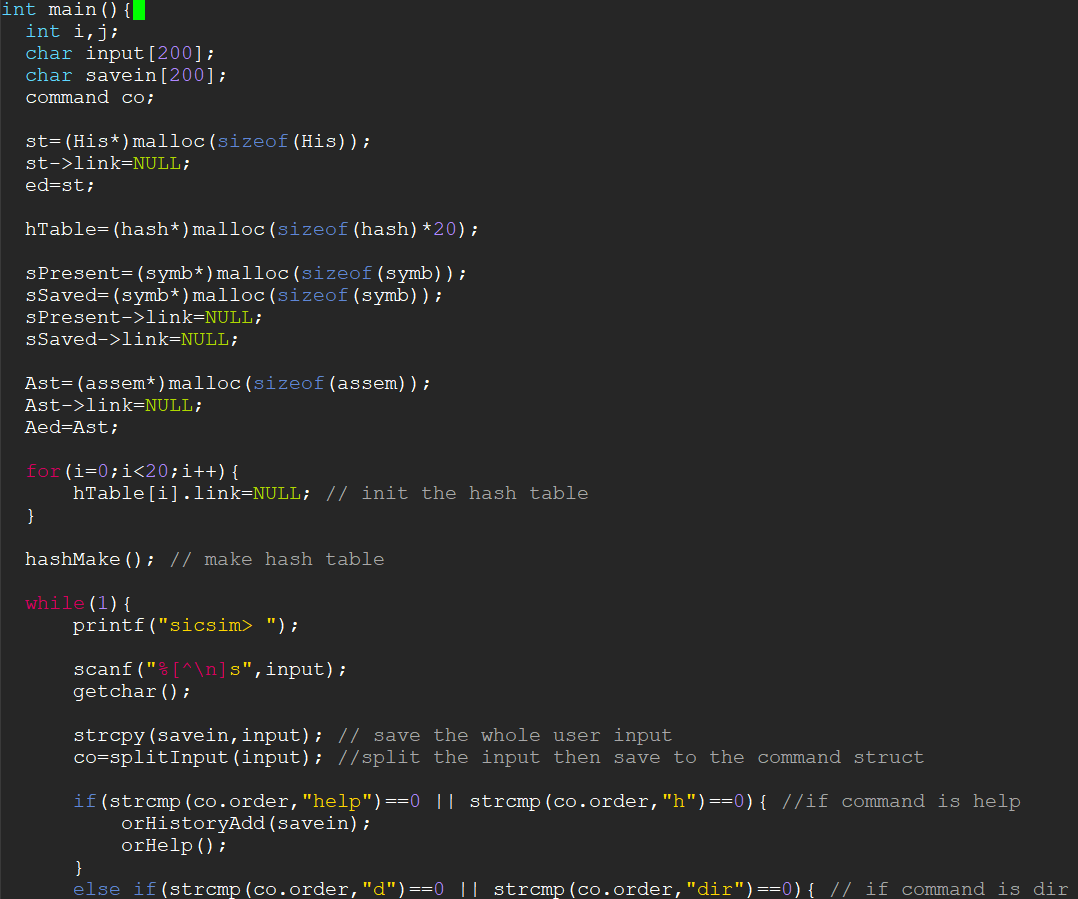
Assemble list의 처음부터 방문하며 각각 free 해준다. 전부 free를 한 이후에는 Ast->link에 NULL을 넣어 초기화하고 Aed 또한 초기화해준다.

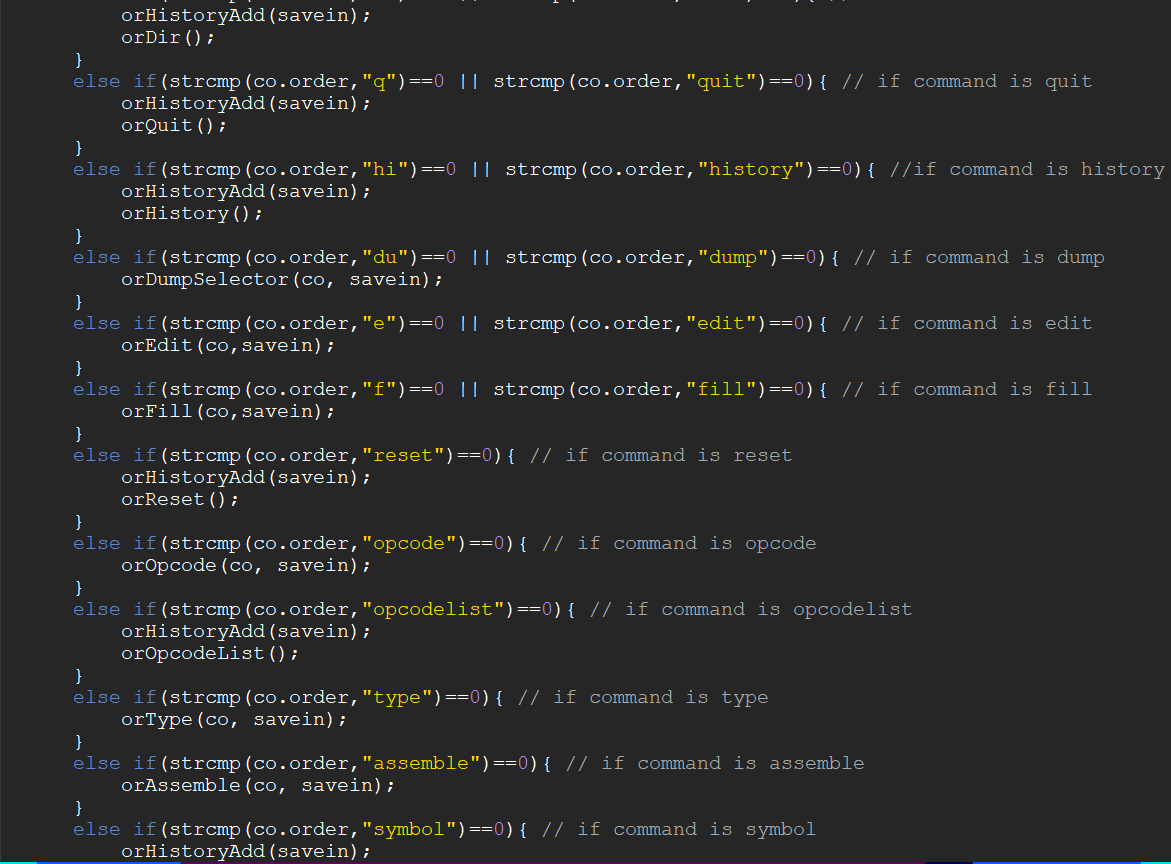
* 1. **symbolDelete**

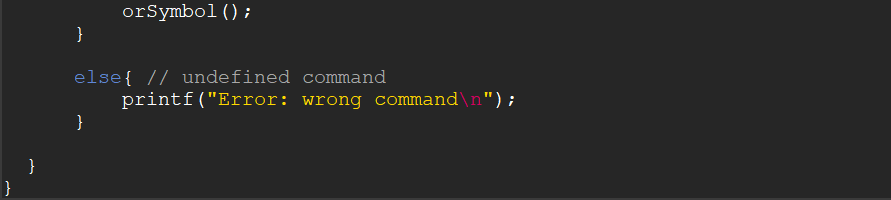


현재 만든 Symbol table, sPresent의 처음부터 방문하며 각각 free 해준다. 이후 sPresent를 초기화해준다.

* 1. **main**







이전 프로젝트와 전체적인 구조는 동일하다. 이전 프로젝트의 main에 type, assemble, symbol을 명령어로 받는 부분을 추가하여 각각 알맞은 함수를 부르게 했다. 또한 처음 Ast, Aed, sPresent, sSaved를 초기화 하는 부분을 추가하였다.