어드벤처디자인

(레벨3)

2021년

김영학, 황준하, 김성영, 윤현주

이 자료는 금오공과대학교 컴퓨터공학과 2학년의 “어드벤처디자인” 과목에서 사용하기 위해 제작되었습니다.

차 례

3. 레벨3

3.1 Human Vs. Com 대전 테트리스 게임

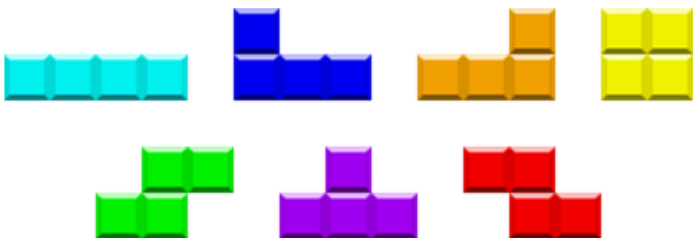
테트리스 게임 개요 (by 위키백과)

테트리스(러시아어: Тéтрис, 영어: Tetris)는 퍼즐 게임으로, 소련의 프로그래머 알렉시 파지노프가 처음 디자인하고 프로그래밍한 게임이다. 테트리스는 1984년 6월 6일에 처음 만들어졌다. 알렉시 파지노프는 모스크바에 위치한 소비에트 과학원(현 러시아 과학원)의 Dorodnicyn 컴퓨터 센터에 근무하고 있었다. 그는 이 게임의 이름을 그리스 숫자 접두어인 Tetra(4)와 파지노프가 좋아하던 테니스를 합쳐서 만들었다.

테트리스(혹은 테트리스의 변종)는 거의 대부분의 비디오 게임기와 컴퓨터 운영 체제에서 가동되며, 휴대폰, PMP, PDA, 네트워크 음악 재생기와 심지어 오실로스코프 같은 기기의 이스터 에그(숨겨진 메시지나 기능)에서도 발견된다. 또한 테트리스는 여러 빌딩의 겉면에서도 실행된 적이 있다.

테트리스는 1980년대 가정용 컴퓨터 시장에서 선을 보였으며, 게임보이의 휴대용 테트리스는 이 게임을 유명게임의 반열에 올려놓게 된다. Eletronic Gaming Monthly란 잡지의 "이제까지 나온 최고의 게임" 100개에서 테트리스는 1위를 차지하였으며, 2007년 IGN의 이때까지 나온 100가지 비디오 게임에서 테트리스는 2위를 차지하였다. 또한 테트리스의 사본은 7억카피 이상 팔린 것으로 집계되었다. 2010년 1월, 테트리스는 2005년부터 시작한 휴대폰 테트리스 다운횟수가 10억 회가 넘었다고 발표되었다.

게임 방법



네 개의 사각형으로 이루어진 '테트로미노'는 무작위로 나타나 바닥과 블록 위에 떨어진다. 이 게임의 목표는 이 테트로미노를 움직이고 90도씩 회전하여, 수평선을 빈틈없이 채우는 것이다. 이러한 수평선이 만들어질 때 이 선은 없어지며 그 위의 블록이 아래로 떨어지는데 테트리스 게임이 진행될수록 테트로미노는 더 빨리 떨어지며 게임을 즐기는 사람이 블록을 꼭대기까지 가득 메워, 테트로미노가 더 들어갈 공간이 없게 되면 게임이 끝나게 된다.

테트리스에서 테트로미노는 I, J, L, O, S, T, Z와 같은 7개의 단면 모양을 일컫는다. 비슷한 글자 모양을 가지고 있기 때문에 이렇게 일컫지만, 게임을 즐기는 사람들은 이러한 조각을 다르게 부르기도 한다.

기본 문제

1인용 테트리스 게임, 2인용 대전 테트리스 게임, 사람과 컴퓨터 간의 대전 테트리스 게임이 가능한 테트리스 게임 프로그램을 작성한다.

테트리스 인공지능 구현을 위한 기본 지식

본 과제에서 가장 중요한 것은 컴퓨터 스스로 테트리스 게임을 할 수 있도록 만드는 것이다. 즉, 일명 테트리스 인공지능(AI)을 만들어야 하며, 이를 위한 한 가지 방법은 다음과 같다.

기본적인 방법은 사람이 생각하는 방식과 유사하다. 사람은 현재 떨어지고 있는 테트로미노와 다음(Next) 테트로미노를 모두 고려하여(가능하다면!) 현재 쌓여있는 테트로미노들을 가능한 많이 없앨 수 있도록 현재 테트로미노의 회전 및 위치를 결정하게 된다. ~~물론 때로는 그 다음 특정 테트로미노가 나올 것을 기대하며 현재 테트로미노의 회전 및 위치를 결정할 수도 있다(아마도 확률적인 기대치가 높다면 좋은 결과를 낳을 수도 있다).~~ 여기서 현재 쌓여있는 테트로미노들을 가능한 많이 없앤다는 것은 달리 생각하면 쌓여있는 테트로미노의 높이를 최소화하는 것과 같다고 할 수 있다.

그렇다면 인공지능 AI가 해야 할 일은 현재 테트로미노와 다음 테트로미노가 가질 수 있는 모든 회전 및 위치를 고려하여 테트리스 판에 쌓이게 되는 테트로미노들의 최대 높이를 최소화하는 것이다. 이 문제는 일종의 탐색(Search) 문제에 해당된다. 다행히 현재 테트로미노와 다음 테트로미노가 가질 수 있는 회전 및 위치의 경우의 수가 많지 않기 때문에 빠른 시간 내에 모든 경우에 대한 평가가 가능하다. 따라서 모든 경우에 대해 테트로미노 판에 쌓이는 테트로미노들의 최대 높이를 구한 후 그 값이 가장 작은 경우가 되도록 현재 테트로미노의 회전과 위치를 결정하면 된다.

물론 여기에 다양한 전략을 추가하여 성능을 향상시킬 수도 있을 것이다.

제한요소 및 요구사항

- 테트로미노 자동 낙하
- 키보드 입력에 따른 테트로미노 이동
- 1인용 테트리스 게임
- 2인용 테트리스 게임 (단순 수행)
- 2인용 대전 테트리스 게임
- 테트리스 인공지능
- 사람과 컴퓨터 간 대전 테트리스 게임

확장 기능

① GUI 구현

3.2 허프만 코딩

개요

허프만 코딩(Huffman coding)은 자료 압축의 가장 오래되고 기초적인 방법 중의 하나이며 최소 중복 코딩(minimum redundancy coding)에 기반한 알고리즘을 사용한다. 최소 중복 코딩은 문자들이 자료 집합에서 얼마나 자주 발생하는지를 안다면 발생되는 문자들의 비율에 따라 자주 반복되는 문자들을 더 적은 비트로 부호화 하는 방법이다. 일반적인 경우 한 문자를 표현하기 위해 한 개의 바이트(ASCII 코드)를 사용하나 허프만 코드는 문자 발생 비율에 따라 다른 크기의 비트로 표현한다.

기본 문제

임의의 텍스트 파일이 주어지면 허프만 코딩 방법을 이용하여 자료를 압축하며 압축된 파일을 해제하는 프로그램을 작성한다. 다수의 텍스트 파일에서 실험을 하여 각각의 압축률과 평균적인 압축률을 계산해 본다.

프로그램 입출력

□ 압축의 경우
- 입력 : 압축되지 않은 임의의 텍스트 파일
- 출력 : 허프만 코드로 압축된 파일

□ 압축 해제의 경우
- 입력 : 허프만 코드로 압축된 파일
- 출력 : 압축 해제된 일반 텍스트 파일

문제 해결을 위한 기본 지식

□ **엔트로피와 최소 중복**

먼저 이 문제를 해결하기 위한 엔트로피(entropy)의 개념을 살펴본다. 모든 자료 집합은 엔트로피라는 어떤 정보화된 내용을 갖는다는 점을 상기하라. 자료 집합의 엔트로피는 자료의 각 문자들의 엔트로피의 합이다. 문자 z의 엔트로피 S_z는 다음과 같이 정의된다.

$$S_z = -\log_2 P_z$$

여기에서 P_z는 자료에서 문자 z를 찾을 확률이다. 문자 z가 발생하는 횟수를 정확히 알 수 있다면 P_z를 z의 빈도라고 한다. 한 예로, 문자 z가 32개의 문자들 중 8번 발생한다면 z의 엔트로피는 다음과 같다.

$-\log_2(1/4) = 2$ bits

이것은 z를 표현하는 데에 2비트보다 많은 비트를 사용하는 것은 필요 이상이라는 것을 의미한다. 일반적으로 기호를 표현하는 데에 8비트(1바이트)를 사용한다면 이 압축은 표현을 많이 개선할 잠재력을 갖는다. (표 1)은 5개의 문자들의 72개 인스턴스를 갖는 자료의 엔트로피 계산 예를 보여준다. 예를 들어 문자 'U'의 경우, 총 엔트로피는 다음과 같이 계산된다. 총 72개 중 'U'는 12번 발생하므로 'U'의 각 인스턴스는 다음과 같은 엔트로피를 갖는다.

$-\log_2(12/72) = 2.584963$ bits

결과적으로 문자 'U'는 자료에서 12번 발생하므로 자료의 엔트로피(문자 'U'의 총 비트수)에 미치는 크기는 다음과 같이 계산된다.

$2.584963 \times 12 = 31.01955$ bits

자료의 전체 엔트로피는 다음과 같이 각 문자들의 엔트로피를 합하여 계산한다.

31.01955 + 36.00000 + 23.53799 + 33.94552 + 36.95994 = 161.46300 bits

각 문자를 표현하는 데에 8비트를 사용하면 자료의 크기가 72×8=576 비트가 되는데, 이 방법을 사용하면 이론상으로 다음과 같은 비율로 자료를 압축할 수 있다.

1-(161.463000/576) = 72.0%

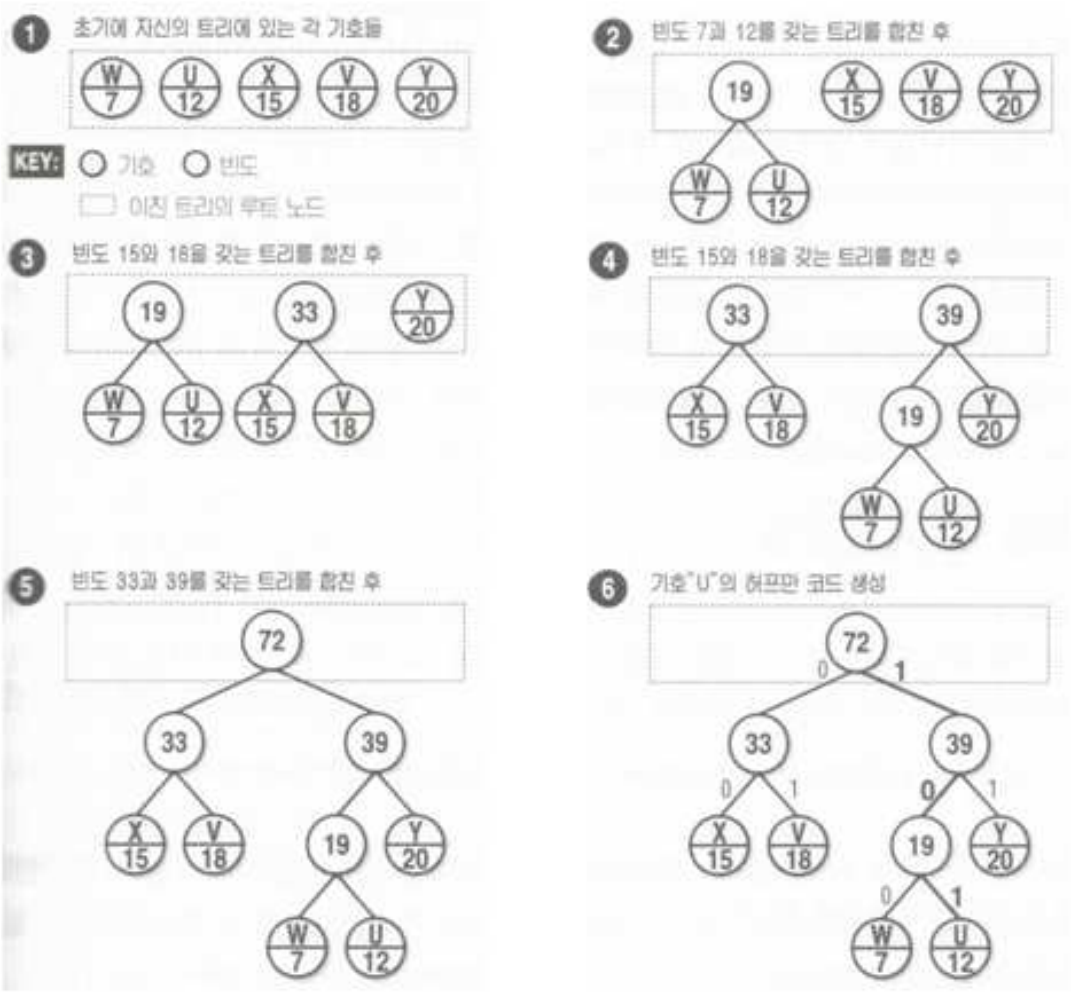
(표 1) 5개 다른 문자의 72개 인스턴스를 갖는 자료 집합의 엔트로피

기호	확률	각 인스턴스의 엔트로피	총 엔트로피
U	12/72	2.584963	31.01955
V	18/72	2.000000	36.00000
W	7/72	3.362570	23.53799
X	15/72	2.263034	33.94522
Y	20/72	1.847997	36.95994

□ **허프만 트리 만들기**

허프만 코드를 생성하기 위해서 자료구조 교과목에서 배운 이진트리를 사용할 수 있다. 허프만 코드는 압축을 위해 문자를 발생 빈도에 따라 서로 다른 크기의 비

트로 할당한다. 그러나 (표 1)을 살펴보면 문자들의 엔트로피가 비트의 소수점으로 계산되기 때문에 허프만 코드는 자료의 엔트로피를 근사하는 압축 방법이다. 허프만 코드에 사용되는 비트의 수가 실제로는 소수점이 될 수 없으므로 어떤 코드들은 최적보다 약간 많은 비트로 지정된다.



(그림 1) 허프만 코드 생성 과정

(그림 1)은 (표 1)의 자료로부터 허프만 트리를 만드는 과정을 보여 준다. 허프만 트리를 만드는 과정은 단말 노드에서 시작하여 위쪽으로 진행된다. 우선 (그림 1)의 단계 1에서와 같이 각 문자와 빈도를 구성 요소로 하는 노드를 생성하여 이를 루트로 하는 독립적인 트리로 구성한다. 다음에 각 트리의 루트 노드를 비교해 빈도가 가장 낮은 두 트리를 하나로 합한다. 이 때 (그림 1)의 단계 2에 보인 것과 같이 두 트리에서 루트 노드의 빈도의 합을 계산하여 새로 생성된 트리의 루트 노드에 저장한다. 이 과정을 (그림 1)의 단계 5에서 보인 것과 같이 최종적으로 하나의 트리가 될 때까지 반복한다. 그러면 최종적으로 생성된 트리가 우리가 원하는 허프만 트리가 된다. 이 트리의 루트 노드는 자료의 문자들의 총 개수를 가지고

있으며, 단말 노드들은 원래 문자들과 빈도를 가지고 있다.

□ **자료의 압축과 복원**

허프만 트리를 만드는 것은 자료 압축과 복원의 일부이다. 특정 문자에 대해 허프만 트리를 사용해서 자료를 압축하려면 트리의 루트에서 시작해서 해당 문자를 갖는 단말 노드까지의 경로를 추적한다. 루트 노드에서 경로를 따라 내려오면서 왼쪽으로 이동할 때에는 현재 코드에 0을 할당하고, 오른쪽으로 이동할 때에는 현재 코드에 1을 할당한다. 즉 (그림 1)의 단계 6에서 'U'의 허프만 코드를 결정하기 위해 오른쪽으로 이동하고(1), 다음에 왼쪽으로 이동하고(10), 다음에 다시 오른쪽으로 이동한다(101). 그러면 최종적으로 문자 'U'에 할당된 허프만 코드는 101의 3비트로 구성된다. (그림 1)에서 같은 방법으로 계산하면 각 문자들의 허프만 코드는 다음과 같다.

U = 101, V = 01, W = 100, X = 00, Y = 11

허프만 트리를 이용해서 자료를 복원하기 위해서는 압축된 자료를 한 비트씩 읽는다. 트리의 루트 노드에서 시작해서 자료에서 0을 만나면 트리의 왼쪽으로 이동하고, 1을 만나면 오른쪽으로 이동한다. 단말 노드에 도달하면 그 노드가 갖는 문자를 생성하고, 트리의 루트 노드로 다시 이동해서 압축된 자료를 다 사용할 때까지 이 과정을 반복한다.

□ **허프만 코딩의 유효성**

허프만 코딩을 사용해서 압축된 자료의 크기를 계산하려면 각 문자들의 빈도와 허프만 코드의 비트수를 곱한 후 모두 더한다. 따라서 (표 1)과 (그림 1)에 표현된 자료의 압축된 크기를 계산하면 다음과 같다.

(12)×(3) + (18)×(2) + (7)×(3) + (15)×(2) + (20)×(2) = 163 bits

72개의 문자들을 압축 없이 8비트로 표현한다고 가정하면 총 자료의 크기는 576비트이므로 다음의 압축률을 얻는다.

1-(163/576) = 71.7%

허프만 코드에서 소수점 비트를 고려할 수 없다는 사실을 생각하면 이 경우에는 자료의 엔트로피가 제안하는 값, 즉 72.0%와 아주 가깝지만, 많은 경우에서 이 값은 엔트로피가 제안하는 값만큼 좋지는 않다. 일반적으로 허프만 코딩이 압축의 가장 효과적인 형태는 아니지만 자료 압축과 복원이 빠르게 실행된다는 장점이 있

다. 허프만 코딩으로 자료를 압축할 때 가장 시간이 많이 걸리는 측면은 자료를 두 번, 즉, 빈도를 모으는 데 한 번, 실제로 자료를 압축하는 데 한 번 스캔해야 한다는 점이다.

일단 압축한 결과를 파일로 저장할 때는 허프만 코드표, 즉 각 문자에 대응하는 비트열을 파일의 앞부분에 먼저 저장하고 압축된 내용을 그 뒤에 저장한다. 모든 내용은 이진(binary) 데이터로 기록되어야 함을 명심해야 한다. 또한 압축 파일을 해독하는 것은 주어진 허프만 코드표를 참조하여 허프만 트리를 구성하고 이를 단순히 따라가 보면 되므로, 복원 과정은 압축보다 매우 짧은 시간에 이루어진다.

제한요소 및 요구사항

- 허프만 코드로 압축한 파일은 이진 파일로 저장한다.
- 허프만 코드로 압축한 파일은 허프만 테이블과 허프만 부호화코드를 포함한다.
- 50,000자, 100,000자, 500,000자 이상의 텍스트 파일을 직접 준비한다.
- 여러 가지 텍스트 파일에 대해 압축을 수행하여 파일별 및 평균적인 통계정보(압축률, 문자당 평균 비트 수, 압축 및 복원 시간 등)를 계산한다.

확장 기능

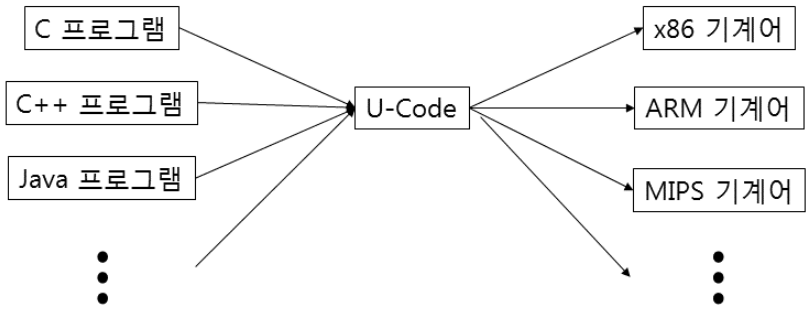
① GUI 구현

② ASCII 문자 이외의 문자(한글, 유니코드 등)에 대해서도 동작하도록 한다. (본 기능의 구현은 쉽지 않을 것으로 판단됨 - 문제 분석 시 구현 가능성에 대해 검토)

3.3 U-Code Interpreter

기본 문제

U-Code는 스택 기반으로 동작하는 간단한 CPU에서 실행되는 것을 가정하는 중간 언어(intermediate language)이다. 중간 언어란 C와 같은 고급 언어로 된 프로그램을 Intel Pentium이나 Xeon 등의 실제 CPU에서 실행될 수 있는 기계어로 변환하는 컴파일 과정의 중간 단계로서, 다양한 고급 언어와 다양한 CPU를 연결하는 역할을 한다. 중간 언어를 실행하는 프로그램은 일종의 가상 머신이라고 볼 수 있다. 본 프로젝트는 U-Code로 작성된 프로그램을 읽어서 실행하는 인터프리터, 즉 가상 머신 시뮬레이터를 작성한다.



U-Code 인터프리터 문제 정의

입력: U-Code로 작성한 프로그램 (*.uco)
(5개의 프로그램이 주어지며 그 외 자신이 작성한 프로그램 추가 가능)
출력: 프로그램의 실행 결과(화면 출력)
실행 과정 및 통계 정보를 보여 주는 파일(*.lst)
(입력된 소스 uco 코드 및 어셈블 결과, 명령어별 사용 횟수, 명령어별 실행 횟수, 메모리 접근 횟수 등)
확장 문제: 실행 과정을 GUI로 보여 준다.

U-Code 예

임의의 C 소스에 대한 U-Code의 예를 보면서 U-Code가 어떻게 구성되고 동작하는지 개념을 잡아 보자. 특별한 설명이 없으면 스택은 스택 꼭대기(top)을 의미한다. 스택의 내용은 최초 실행 시(i = 0)의 값을 나타내고 있다.

아직까지 U-Code 명령어들에 대한 정확한 의미를 파악하기 힘들다. 따라서 바로 이어 설명되는 U-Code 상세 명세를 함께 참고하여 파악한다.

```
const int MAX = 10;     // 상수형 전역 변수 선언 및 초기화

void main(void) {
   int i;               // 정수형 지역 변수 선언
   int a[10];           // 10개의 원소를 가지는 정수형 배열 선언
   int j;               // 정수형 지역 변수 선언
   i = j = 0;           // 변수에 값 배정
   while (i < MAX) {    // 반복문의 시작 및 조건 검사
      a[i] = j;         // 배열 변수에 저장
      a[i] = sub(i, a); // 함수 호출
      j = j + a[i];     // 덧셈
      ++i;              // 증가 연산
   }
   write(j);            // 출력문 - 일반 C의 출력함수와 다르게 하나의 정수 출력
}

int sub (int i, int a[]) { // 함수의 시작 및 전달 인자 선언
   int j;               // 정수형 지역 변수 선언
   read(j);             // 입력문 - 일반 C의 입력함수와 다르게 하나의 정수 입력
   j = j + a[i];        // 변수와 인자 덧셈
   return j;            // 함수에서 값 반환
}
```

U-Code		해설	CPU 스택
	bgn 1	프로그램의 시작 - 전역 변수의 개수는 1	
	sym 1 0 1	MAX 변수의 블록(1)과 오프셋(0)과 크기(1) 설정	
	ldc 10	상수 10을 스택에 적재	10
	str 1 0	스택의 값을 블록 1, 오프셋 0에 저장	
	ldp	함수 호출 준비 명령	
	call main	main 함수 호출(항상 main을 호출해야 함)	
	end	프로그램 종료	
main	proc 12	함수 시작 - 지역 변수의 총량 선언	
	sym 2 0 1	i 변수의 블록(2)과 오프셋(0)과 크기(1) 설정	
	sym 2 1 10	a 배열의 블록(2)과 오프셋(1)과 크기(10) 설정	
	sym 2 11 1	j 변수의 블록(2)과 오프셋(11)과 크기(1) 설정	
	ldc 0	상수 0을 스택에 적재	0
	dup	스택의 값을 복사하여 스택에 적재	0 0
	str 2 11	스택의 값을 블록 1, 오프셋 11에 저장 (j = 0)	0
	str 2 0	스택의 값을 블록 1, 오프셋 0에 저장 (i = 0)	
$$0	nop	while 문의 시작점 - 반복 시 돌아올 위치 $$0	
	lod 2 0	블록 2, 오프셋 0의 값을 스택에 적재(i)	0
	lod 1 0	블록 1, 오프셋 0의 값을 스택에 적재(MAX)	0 10
	lt	스택 꼭대기의 두 값을 비교하여 결과를 적재	$-1^{(1)}$
	fjp $$1	스택의 값이 거짓이면 $$1로 점프(반복 종료)	
	lod 2 0	블록 2, 오프셋 0의 값을 스택에 적재(i)	0
	lda 2 1	블록 2, 오프셋 1의 주소를 적재(a의 주소)	$0\ 2^{(2)}$
	add	스택 꼭대기의 두 값을 더하여 결과를 적재(a[i])	2
	lod 2 11	블록 2, 오프셋 11의 값을 스택에 적재(j)	2 0
	sti	스택 top의 값을 top-1번지에 저장	
	lod 2 0	블록 2, 오프셋 0의 값을 스택에 적재(i)	0
	lda 2 1	블록 2, 오프셋 1의 주소를 적재(a의 주소)	0 2
	add	스택 꼭대기의 두 값을 더하여 결과를 적재(a[i])	2

	ldp	함수 호출 준비	2
	lod 2 0	블록 2, 오프셋 0의 값을 스택에 적재(i)	2 0
	push	스택의 값을 메모리 스택에 적재	2
	lda 2 1	블록 2, 오프셋 1의 주소를 적재(a의 주소)	2 2
	push	스택의 값을 메모리 스택에 적재	2
	call sub	sub 함수 호출	2 100[3]
	sti	스택 top의 값을 top-1번지에 저장	
	lod 2 11	블록 2, 오프셋 11의 값을 스택에 적재(j)	0
	lod 2 0	블록 2, 오프셋 0의 값을 스택에 적재(i)	0 0
	lda 2 1	블록 2, 오프셋 1의 주소를 적재(a의 주소)	0 0 2
	add	스택 꼭대기의 두 값을 더하여 결과를 적재(a[i])	0 2
	ldi	스택 값의 번지에 있는 값을 적재	0 100
	add	스택 꼭대기의 두 값을 더하여 결과를 적재	100
	str 2 11	스택의 값을 블록 2, 오프셋 11에 저장(j)	
	lod 2 0	블록 2, 오프셋 0의 값을 스택에 적재(i)	0
	inc	스택의 값을 하나 증가	1
	str 2 0	스택의 값을 블록 2, 오프셋 0에 저장(i)	
	ujp $$0	$$0 위치로 무조건 점프	
$$1	nop	$$1 위치 표시 (아무 동작 없음 - nop)	
	ldp	함수 호출 전 인자 적재	
	lod 2 11	블록 2, 오프셋 12의 값을 스택에 적재(j)	52405[4]
	push	스택의 값을 메모리 스택에 적재	
	call write	값을 출력	
	ret	main 함수 종료	
sub	proc 3	함수 시작 - 지역 변수의 총량 선언	2
	sym 2 0 1	i 인자의 블록(2)과 오프셋(0)과 크기(1) 설정	2
	sym 2 1 1	a 인자의 블록(2)과 오프셋(1)과 크기(1) 설정	2
	sym 2 2 1	j 변수의 블록(2)과 오프셋(2)과 크기(1) 설정	2
	ldp	함수 호출 전 인자 적재	2
	lda 2 2	블록 2, 오프셋 2의 주소를 스택에 적재(j 주소)	2 15[5]
	push	스택의 값을 메모리 스택에 적재	2
	call read	read 함수 호출. 지정 주소에 입력값 저장	2
	lod 2 2	블록 2, 오프셋 2의 값을 스택에 적재(j)	2 100[6]
	lod 2 0	블록 2, 오프셋 0의 값을 스택에 적재(i)	2 100 0
	lod 2 1	블록 2, 오프셋 1의 값을 스택에 적재(a)[7]	2 100 0 2
	add	스택 꼭대기의 두 값을 더하여 결과를 적재(a[i])	2 100 2
	ldi	스택 값의 번지에 있는 값을 적재	2 100 0
	add	스택 꼭대기의 두 값을 더하여 결과를 적재	2 100
	str 2 2	스택의 값을 블록 2, 오프셋 2에 저장(j)	2
	lod 2 2	블록 2, 오프셋 2의 값을 스택에 적재	2 100
	ret	sub 함수 종료	2 100

(1) 거짓을 0, 참을 -1로 표시한다고 가정

(2) 블록 1이 1번지부터 시작한다고 할 때

(3) sub 함수에서 반환된 값이 100이라고 가정

(4) while문이 끝나고 j에 저장된 최종 값이 52405라고 가정

(5) 블록 2가 13번지부터 시작한다고 할 때

(6) 입력값이 100이라고 가정

(7) 같은 배열 이름이라도 함수의 인자인 경우 포인터와 같이 취급하여 lda가 아닌 lod 사용

U-Code 상세 명세

U-Code는 가상 머신에서 수행되는 명령어이므로 『컴퓨터구조』에서 배운 CPU 명령어(instruction)와 유사하며 훨씬 단순하다.

1) 데이터는 정수형만 있는 것으로 가정한다. 따라서 데이터의 크기는 바이트 크기가 아닌 데이터의 개수로만 표현한다.
2) Addressing - CPU 내 레지스터가 없고, 모든 연산이 스택을 기준으로 이루어진다. 따라서 ALU 동작에 대한 명령은 피연산자를 표현하지 않고, 스택 꼭대기의 값을 가져 와서(pop) 연산 후 다시 스택 꼭대기에 올려놓는(push) 것을 가정한다(0-addressing). 상수나 메모리에 저장된 값을 스택에 가져 오거나 스택의 값을 메모리로 보내는 경우, 분기(jump) 명령어의 경우 피연산자가 하나 지정된다(1-addressing).
3) 형식 제한 요소: U-Code 프로그램은 정해진 열을 지켜 작성해야 한다. 1~10째 열까지는 레이블에 사용하고 11번째 열은 공백, 12번째 열부터 명령어가 나타날 수 있으며, 명령어 및 피연산자 사이에는 최소한 하나 이상의 공백이 있어야 한다. (이 경우 1~11번째 열의 공간에 자동으로 탭 문자가 삽입되지 않도록 주의한다.)

```
( 12345678901234567890 )
 main        proc 12
             sym 2 0 1
             ldp
```

4) %로 시작하는 부분부터 그 행의 끝까지 주석문(comment)으로 인식하여 아무 작업도 하지 않는다.

다음은 U-Code를 구성하는 명령어를 종류별로 소개한다.

1) 프로그램 구성 명령
   프로그램 구성 명령은 CPU가 실행하는 명령이 아니라 실행 시 필요한 요소를 미리 설정하거나 인간이 프로그램을 이해하는데 필요한 명령들이다. 따라서 어셈블 단계에서만 사용하고 실행 시에는 사용되지 않으며, opcode가 할당되지 않는다.

명령어	의미	동작	예
nop	no operation	아무 작업도 수행하지 않으며 주로 레이블 위치에 사용됨	label00 nop
bgn *n*	begin	프로그램의 시작점. *n*은 전역 변수의 총량	bgn 1
sym *b n s*	symbol	변수(심볼)가 속한 블록(*b*)과 블록 내에서의 오프셋(*n*) 및 크기(*s*) 표현 - 전역 변수의 블록 번호는 1, 지역 변수의 블록 번호는 2 - 오프셋은 0부터 시작 - 일반 정수형은 크기가 1 - 배열형은 배열의 크기만큼 변수 선언부를 어셈블리어로 표현한 것으로, 실제로는 사용되지 않고 인간의 이해를 돕는 코드	% int i; % int a[10]; % int j; sym 2 0 1 sym 2 1 10 sym 2 11 1
end	end	프로그램의 끝. 어셈블을 종료하는 역할	

2) 함수 정의 및 호출

명령어	의미	동작	예

proc *n*	procedure	함수의 시작을 나타내며, n은 매개 변수를 포함하여 함수가 사용하는 모든 지역 변수의 총량	/* int sub(int i,
			int a[]) {
			…
			return i;
			}
			*/
			sub proc 2
			…
			lod 2 0
			ret

ret	return	함수를 종료하고 복귀. 반환값이 있을 때는 ret 전에 CPU 스택에 저장해야 한다.	
ldp	load parameters	함수의 실인자들을 스택에 저장. 함수를 호출하기 전에 함수가 사용할 메모리 영역을 설정하고 매개변수들을 전달할 준비를 한다.	/* j = sub(i, a) */ ldp lod 2 0 push lda 2 1 push call sub str 2 11
push	push parameters	CPU 스택에 올려져 있는 실인자 값을 메모리 스택에 저장한다.	
call *label*	call	label로 지정된 함수를 호출	

3) 입출력 처리

실제 시스템에서 입출력은 운영체제와 연관되는 복잡한 과정을 거치게 되고 시스템 수준의 함수에서 이를 처리하도록 지원한다. U-Code에서는 입출력에 대한 간단한 시스템 함수를 가정하고 사용할 수 있도록 한다.

시스템 함수	U-Code 사용 예	동작
read(i)	ldp lda 2 1 push call read	외부 입력값을 읽어 스택 꼭대기에 저장된 주소로 저장한다.
write(i)	ldp lod 2 0 push call write	스택 꼭대기의 값을 출력한다. 출력된 값의 뒤에 공백을 추가로 출력한다.
lf()	ldp call lf	줄바꿈 문자를 출력한다.

4) 데이터 이동 연산자

메모리에 있는 모든 데이터(변수값)는 연산을 실행하기 전에 스택으로 이동해야 하고, 저장하기 위해서는 스택에서 메모리로 이동해야 한다. 이 때 각 변수의 주소는 sym 명령에 의해 표현된 블록과 오프셋을 이용해 계산한다. 그리고 스택으로 이동하는 경우 스택의 데이터가 하나 늘어나 top의 값이 증가하고, 스택에서 메모리로 이동하는 경우 스택의 데이터가 하나 줄어 top의 값이 감소함을 명심해야 한다.

명령어	의미	동작	예
lod *b n*	load	b 블록 n 오프셋의 데이터를 스택에 넣는다. 즉, b와 n으로 계산되는 주소에 있는 변수의 값이 스택에 저장된다.	lod 2 11
lda *b n*	load address	b 블록 n 오프셋의 실제 메모리 번지를 스택에 넣는다. 즉, b와 n으로 계산되는 주소 자체가 스택에 저장되며, 배열 참조를 위해 활용된다.	lda 2 1
ldc *c*	load constant	상수값 c가 스택에 저장된다.	ldc 100
str *b n*	store	스택 꼭대기의 값을 b와 n으로 계산되는 주소의 메모리에 저장한다.	str 2 11
ldi	load indirect	간접 주소법을 이용해 메모리의 값을 스택에 가져 온다. 스택 꼭대기의 값을 pop하여 주소값으로 사용하고, 데이터를 스택에 저장한다.	lod 2 0 % i lda 2 1 % a add % a[i] ldi
sti	store indirect	간접 주소법을 이용해 스택 꼭대기의 값을 메모리에 저장한다. 저장할 변수의 주소와 저장할 값, 두 개가 스택에서 pop된다.	lda 2 1 % a lod 2 0 % i add % a[i] lod 2 11 % j sti % a[i] = j

5) 단항(unary) 연산자

단항 연산자의 모든 피연산자는 스택의 맨 위에 있으며, 결과는 다시 스택에 저장된다. 이해를 돕기 위해 stack을 1차원 배열로 가정한 C 코드로 동작을 표

현하였다. (실제 동작은 pop -> 연산 -> push의 순서로 이루어져야 하나 stack을 직접 조작할 수 있는 것으로 가정함)

명령어	의미	동작	예
not	not	피연산자의 진리값을 변경 stack[top] = !stack[top];	not
neg	negation	피연산자의 음양값을 변경 stack[top] = -stack[top];	neg
inc	increment	피연산자의 값이 하나 증가. 이 연산은 스택의 값만 증가시키므로 C의 ++연산자와 달리 메모리의 값은 증가하지 않음을 명심해야 한다. ++stack[top];	/* ++i */ lod 2 0 inc str 2 0
dec	decrement	피연산자의 값이 하나 감소 --stack[top];	/* --i */ lod 2 0 inc str 2 0
dup	duplicate	스택 꼭대기의 값을 복사 stack[++top] = stack[top];	(swp 예 참조)

6) 이항(Binary) 연산자

이항 연산자의 모든 피연산자는 스택 꼭대기와 그 하나 아래의 값이며, 결과는 다시 스택에 저장된다.

명령어	의미	동작	예
add sub mult div mod	add subtract multiply divide modulo	stack[top-1] = stack[top-1] + stack[top]; top--; (순서대로 -, *, /, % 등으로 연산자만 바뀌며 동작은 동일함)	/* i=i+j */ lod 2 0 lod 2 11 add str 2 0
gt lt ge le eq ne and or	greater than less than gt or equal lt or equal equal not equal and or	stack[top-1] = stack[top-1] > stack[top]; top--; (순서대로 <, >=, <=, ==, !=, &&, || 등으로 연산자만 바뀌며 동작은 동일함)	/* i < j */ lod 2 0 lod 2 11 lt

swp	swap	tmp = stack[top-1];
		stack[top-1] = stack[top];
		stack[top] = tmp;
		
		(증감연산자를 적용한 변수가 다른 연산에 사용될 때 하나의 값이 자신에게도 저장되고 연산에도 사용되어야 하므로 값이 복사(dup)되어야 한다. 순서는 증감연산자가 앞에 붙는지 뒤에 붙는지에 따라 다르다. 배열 변수가 같이 사용될 때는 주소와 값이 스택에 저장되는 순서가 달라질 수 있어 swp 연산이 필요하다.)
			/* i=++a[2] */
			lda 2 1
			ldc 2
			add
			ldi
			inc
			dup
			lda 2 1
			ldc 2
			add
			swp
			sti
			str 2 0

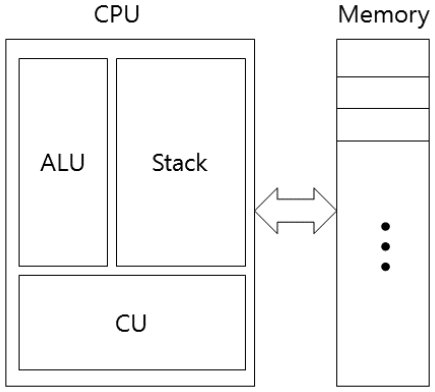
7) 흐름 제어

명령어	의미	동작	예
ujp *label*	unconditional jump	지정한 *label*로 무조건 이동	/* while (i < 100) ++i; */ $$1 nop lod 2 0 ldc 100 lt fjp $$2 lod 2 0 inc str 2 0 ujp $$1 $$2 nop
tjp *label*	jump on true	stack[top]의 값이 참이면 *label*로 이동	
fjp *label*	jump on false	stack[top]의 값이 거짓이면 *label*로 이동	

U-Code 가상 머신 구조와 프로그램 실행 절차

정의된 U-Code 명령어와 하나의 예만으로 U-Code 인터프리터가 어떻게 동작해야 하는지 이해하기 어려울 것이다. 이해와 설계를 돕기 위해 U-Code 가상 머신 구조를 소개하고 프로그램 실행 절차, 특히 함수 호출 및 메모리 사용에 대해 좀 더 상세히 설명한다.

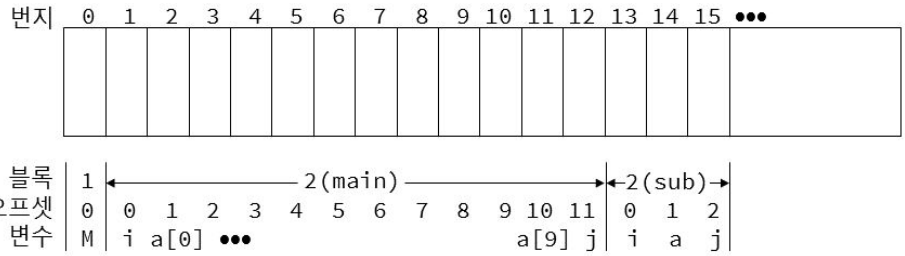
U-Code를 실행하는 가상 머신의 구조는 구현하는 사람이 마음대로 정의해도 되지만, <그림 2>의 기본 구조는 반드시 지켜야 한다.



<그림 2> 가상 머신 기본 구조

1) CPU는 ALU와 CU의 기능을 수행하며, 임시 저장 장치로 여러 개의 범용 레지스터 대신 단 하나의 스택만 사용한다. 모든 명령어는 스택 꼭대기에 있는 데이터에 대해서 작업하고, 그 결과도 스택 꼭대기에 저장한다. 이를 CPU 스택이라 한다.
2) 메모리는 1차원 배열로 정의된다. U-Code의 모든 데이터는 단일한 크기의 정수형이므로 메모리 주소는 정수 하나를 저장할 수 있는 단위로 한다. (바이트 단위가 아님)

이 단순한 구조에서 앞의 예제 프로그램을 실행할 때, 메모리 내부가 다음과 같이 구성될 것으로 가정하고 구현할 수 있다.



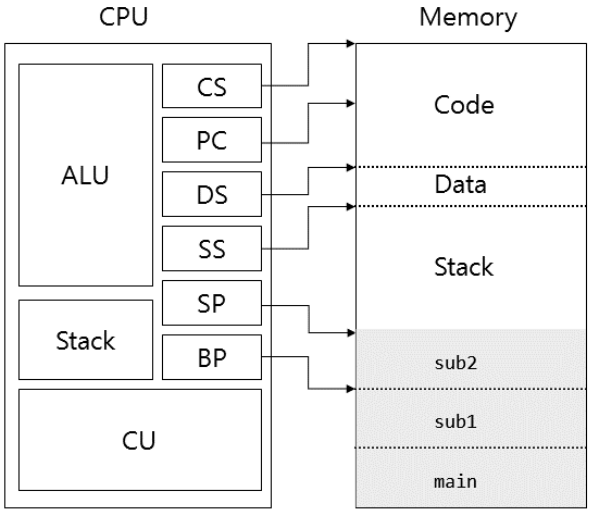
<그림 3> 메모리 사용 예

이 때, 각 함수마다 시작 번지가 어떻게 되는지 알아야 오프셋에 따라 실주소를 계산할 수 있고, 재귀 호출과 같은 경우 동일 함수에 대해 필요한 메모리는 어떻게 할당하는지 생각해야 하며, ret 명령어를 만났을 때 복귀해야 하는 곳이 어딘지도 기억해야 한다. (여기까지만 읽고 나름대로 방법을 고안해 구현해도 좋다.)

문제 해결에 좀 더 빨리 다가가기 위해 현실의 컴퓨터와 메모리 사용법을 조금 빌

려 와 보자. <그림 4>는 보다 상세한 가상 머신과 가상 메모리 구조를 보여 준다.

먼저 메모리는 Code, Data, Stack의 세 영역으로 나누어 관리한다. (일반 프로그래밍 언어에서는 동적 메모리 할당을 위한 Heap 영역이 추가로 존재한다.) 그리고 CPU 쪽에는 각 메모리 영역의 위치를 관리하는데 사용될 6개의 레지스터가 추가되었다.



<그림 4> 현실적인 가상 머신 구조

Code 영역에는 기계어로 구성된, 즉 정수로만 이루어진 프로그램이 저장된다. CS 레지스터는 이 영역의 시작 주소를 가리킨다. 실제 컴퓨터에서는 동시에 여러 개의 프로그램이 실행 중 상태이고 따라서 메모리도 프로그램마다 영역이 나누어져 관리되기 때문에 현재 실행 중인 프로그램의 메모리 위치를 가리키기 위해 이 레지스터의 값이 변화하지만, 우리 가상 머신은 한 번에 하나의 프로그램만 실행하므로 사실 이 레지스터는 불필요하다. (이 메모리는 현실 컴퓨터와 운영체제에서 "가상메모리"라고 생각할 수 있다.)

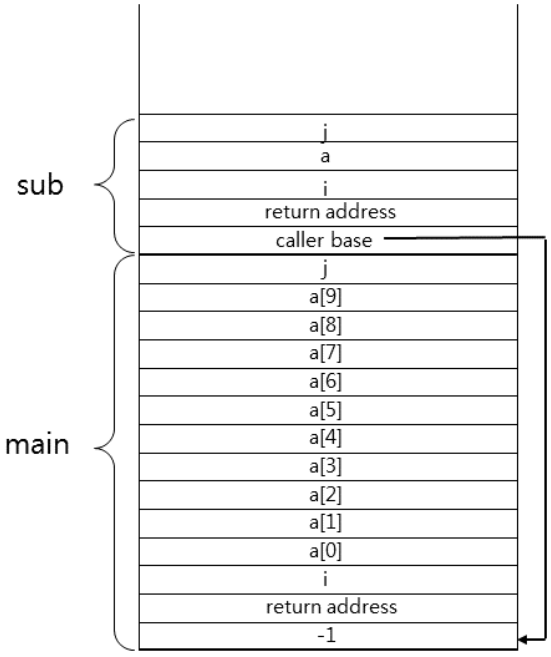
흔히 **PC(Program Counter)** 또는 **IP(Instruction Pointer)**라고 부르는 레지스터는 현재 실행 중인 명령어의 주소를 저장한다. 순차적인 명령 수행 과정에서는 PC의 값이 하나씩 증가하지만, 점프나 함수 호출 등과 같은 경우에는 지정된 위치의 주소를 PC에 넣어 주어야 한다. 또한 함수 호출이 종료한 후 돌아올 위치를 어딘가에 저장해 두어야 한다.

Data 영역은 전역 변수를 저장하는 곳이다. 즉, U-Code 정의에서 블록 1로 표현된 곳이 이 영역이다. Data 영역의 시작 주소는 DS 레지스터에 저장해 둔다. 이 위치는 Code의 크기에 따라 달라질 수 있다. 전역 변수의 실주소는 다음과 같이 계산할 수 있을 것이다.

$$addr = DS + offset$$

Stack 영역은 지역 변수를 저장하는 곳이다(CPU 스택과 혼동하지 말자). 함수가 호출될 때마다 그 함수에서 사용할 변수의 크기와 순서대로 스택에 쌓이게 된다. 보통 하나의 함수가 사용하는 영역 전체를 프레임이라고 한다. 그런데, 현실의 환경에서는 이 프레임에 지역 변수만 저장되는 것이 아니다. 호출이 완료되었을 때 복귀할 코드의 주소가 필요하고, 자신을 호출한 함수의 실행을 계속하기 위해 그 함수의 프레임 위치도 필요하다. 추가로 전역 및 함수 내 지역 변수뿐만 아니라 for 문이나 다른 블록에서 변수를 선언하는 것이 가능한 경우, 각 변수의 사용 범위를 계산하기 위한

정보도 필요할 수 있다. 여기서는 문제를 단순화하기 위해 코드 블록 내에서 새 변수를 선언할 수 있는 기능은 없고, 모든 변수는 전역 또는 함수에 소속된 지역 변수 2단계만 있다고 가정한다. 또한 객체지향언어에 있는 여러 가지 변수의 종류와 사용 범위도 고려하지 않는다. 추후 『프로그래밍 언어 개념』, 『컴파일러』 등의 교과목에서 보다 다양한 형태의 변수 사용 범위와 그를 실행하기 위한 구조를 학습한 후 U-Code와 U-Code 가상 머신을 발전시켜 보는 것도 좋을 것이다. <그림 5>는 앞의 예제에 대한 메모리 스택의 구조를 보여 준다.



<그림 5> 메모리 스택 구조

일반적으로 스택 영역의 바닥은 전체 메모리의 가장 마지막 주소가 되고, 데이터는 주소의 역순으로 쌓인다. 함수가 새로 호출될 때마다 스택에 새 프레임이 쌓이게 되므로 재귀 호출과 같이 함수 호출이 매우 많을 경우 스택 영역이 부족할 수도 있다. 이 때 스택의 넘침(overflow)을 체크하기 위해 SS 레지스터에 스택의 끝 주소를 미리 넣어둘 수 있다(현실 CPU에서 SS 레지스터는 약간 다른 용도로 사용되지만...). 재귀 호출 예제를 이용해 메모리의 크기와 호출 횟수의 관계를 테스트해 보라.

SP 레지스터는 스택 프레임의 꼭대기를 가리키고, BP(Base Pointer)는 현재 실행 중인 함수가 사용하는 프레임의 첫 번째(base) 위치를 저장하고 있다. <그림 5>의 스택 구조에서 지역 변수의 주소는 다음과 같이 계산될 수 있을 것이다.

$$addr = BP - 2 - offset$$

BP는 새로운 함수가 호출될 때 그 값이 바뀌므로 함수가 종료되었을 때 호출한 함수의 메모리 영역으로 돌아가기 위해 이전 BP의 값을 어딘가 저장해 두어야 한다. 그림의 “caller base”가 그 값이다. “return address”는 실행 코드 중 call 명령어 다음의 명령어 위치로서, 이 값을 PC에 넣어 주면 그 위치부터 실행을 재개할 수 있다.

그러면 이러한 메모리 구조를 사용하는 인터프리터의 동작 알고리즘을 대략 설계해 보자. 인터프리터는 문자로 된 U-Code 파일을 읽어 각 명령어를 지정된 opcode, 즉 정수로 변경하고 필요한 피연산자를 추가로 저장하는 1단계 어셈블 과정과, Code 영역에 저장된 명령어를 하나씩 읽어 실행하는 2단계로 구성된다. 각 단계에서 수행해야 할 주요 요소를 살펴보자.

<table>
<tr><th>단계</th><th colspan="2">작 업</th></tr>
<tr><td>어셈블
단계</td><td colspan="2">- 레이블, 명령어, 피연산자를 분리하여 문자열로 인식한다.
- 명령어에 해당하는 opcode를 메모리의 Code 영역에 저장한다.
- 피연산자가 있는 경우 이를 정수로 변경하여 opcode 다음에 저장한다.
- 레이블의 경우 위치(주소)를 별도로 저장했다가 해당 레이블이 피연산자로 사용될 때 그 주소를 기록해 주어야 한다. main 함수 선언보다 call main이 먼저 나오는 것처럼, 피연산자인 레이블의 정확한 위치를 알 수 없는 경우도 처리할 수 있어야 한다.
- bgn 명령어를 만나면 시작 위치로 기록해 두어야 한다.
- 어셈블이 끝나 코드의 크기를 알게 되면 DS에 값을 저장한다.</td></tr>
<tr><td rowspan="2">실행
단계</td><td colspan="2">- 어셈블 단계에서 기록한 시작 위치를 PC에 넣고 시작한다.
- 명령어를 하나 수행 시마다 PC의 값을 다음 명령어 위치로 변경한다. 명령어의 피연산자 수에 따라 이 위치 계산이 달라진다. jump 명령어의 경우 레이블의 값을 PC에 넣으면 된다.
- 메모리를 사용하는 명령어의 경우, 피연산자의 블록과 오프셋 값을 이용해 실제 메모리 주소를 계산한다.</td></tr>
<tr><td>함수
호출
과정</td><td>- ldp: 프레임 기초를 만든다. 즉, SP를 SP-2의 위치로 옮긴다.
- push: CPU 스택의 값을 메모리 스택에 쌓는다. 즉, 실인자의 값을 함수의 매개변수에 저장하는 과정이 된다. (예에서 i와 a)
- call: 프레임 기초의 값을 채운다. 즉, 현재 PC의 값을 이용해 return address를 계산하고, BP의 값이 caller base값이 된다. 그 다음 BP의 값을 새로운 프레임 베이스 위치로 변경하고, PC에 호출한 함수의 위치를 저장한다.
- proc: 지역 변수의 크기를 이용해 SP의 값을 변경한다.
- ret: SP의 값을 BP+1로 변경한다(스택 전체를 삭제하는 의미). return address를 PC에 저장하고, BP의 값을 caller base값으로 변경한다.</td></tr>
</table>

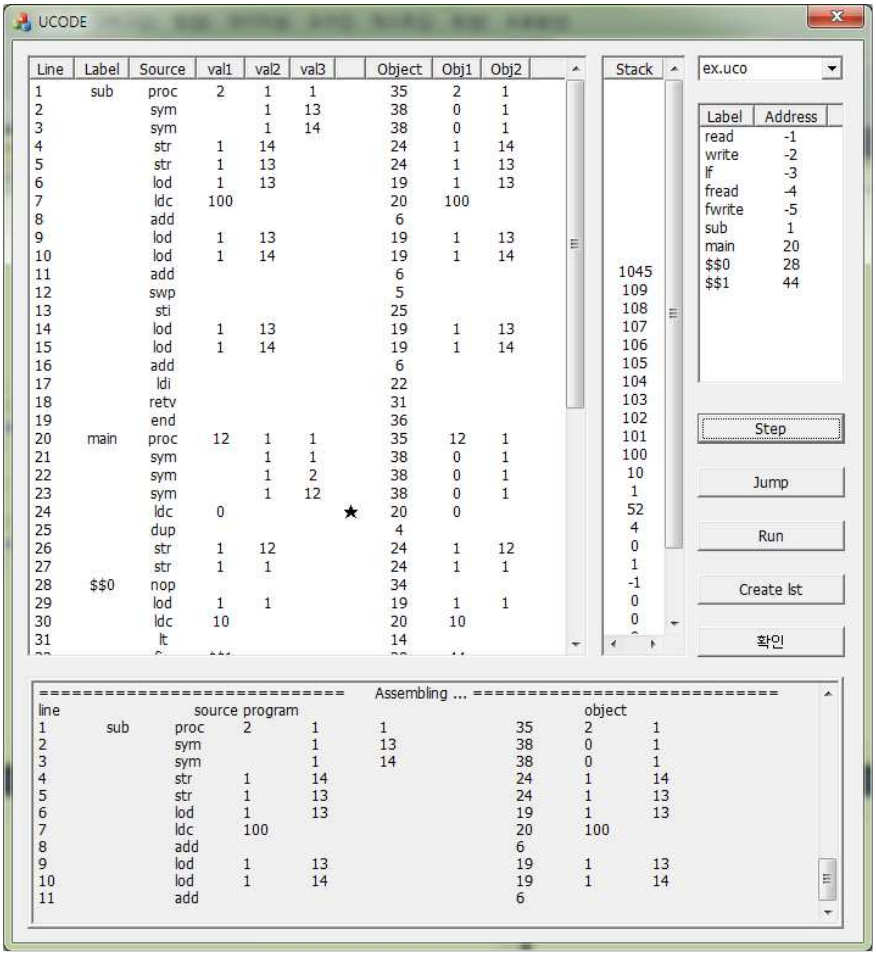
출력 예

<텍스트 모드>

```
juyoon@linda:~/test$ ucode ex.uco
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
100
2
1
3
5
8
9
2
1
3
52405
```

<GUI 모드>

(예시일 뿐이며, U-Code 정의와 가상 머신 구조가 다르며 프로그램도 다름)
(현재 실행 중인 코드를 ★로 표시)



<.lst 파일 예> (형식만 보여줄 뿐 내용은 실제 실행과 다를 수 있음)

```
 addr  object              ucode  source program
-------------------------------------------------
   0    37  0                      bgn    1
   2    20  10                     ldc    10
   4    24  1  0                   str    1      0
   7    23                         ldp
   8    29  11                     call   main
  10    36                         end
  11    35  12           main      proc   12
  13    20  0                      ldc 0
  15    4                          dup
  16    24  2  12                  str    2      12
  19    24  2  0                   str    2      0
  22    19  2  0                   lod    2      0

... (중간 생략)

   ****    Result    ****
 52405
   **********************

   ****  Static Instruction Counts ****

inc = 2       add = 5       lt = 1        lod = 12
... (중간 생략)


   **** Dynamic Instruction Counts ****

inc = 10      add = 20      lt = 10       lod = 98
... (중간 생략)

Total Instruction Execution: 785
```