내공 있는 프로그래머로 길러주는

컴파일러의 이해

Chapter 12
목적 코드 생성

목차

학습목표

- 목적 코드 생성의 개념에 대해 이해할 수 있다.
- 트리-패턴 매칭 기법에 대해 이해할 수 있다.
- 산술식과 논리식에 대한 목적 코드 생성에 대해 이해할 수 있다.
- 지역과 전역 레지스터 할당과 배정과 그래프 착색에 의한 레지스터 할당에 대해 이해할 수 있다.

12.1 목적 코드 생성의 개념

- **목적 코드 생성**(target code generation)은 컴파일러의 마지막 단계로, 중간 코드와 기호표의 정보를 받아서 중간 코드 형태의 프로그램을 목적 기계에 맞는 기계어로 생성하거나 어셈블리어로 생성하는 것이다.
- 목적 코드 생성을 담당하는 도구를 목적 코드 생성기target code generator라고 하며, 이를 [그림 12-1]에 나타냈다.

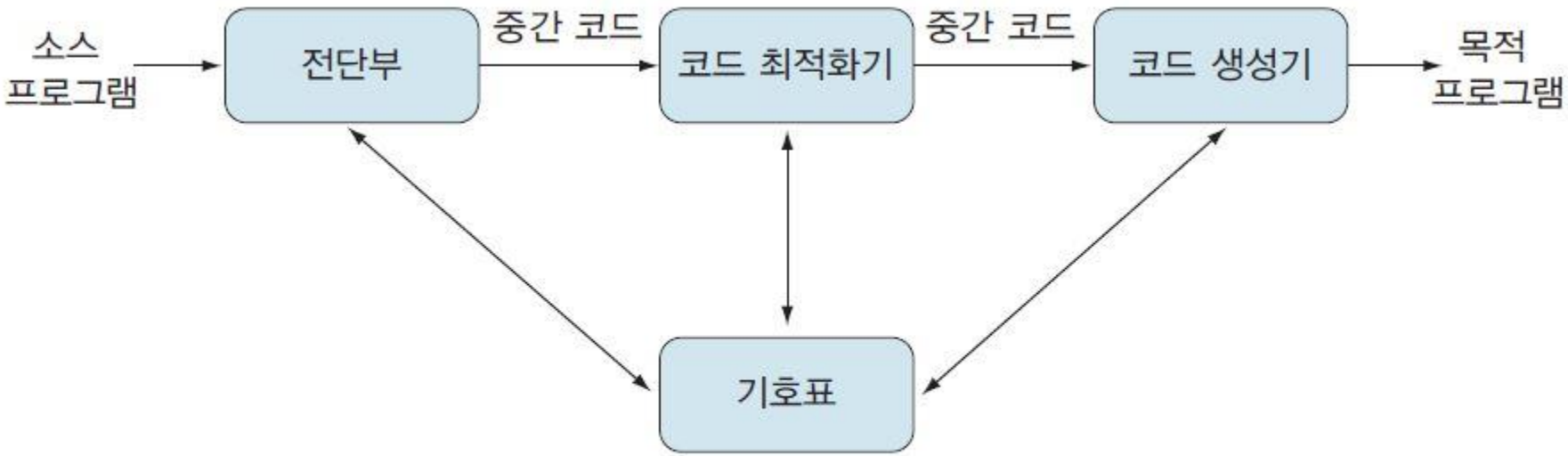


그림 12-1 목적 코드 생성기

12.1 목적 코드 생성의 개념

- 목적 코드 생성기에서는 명령어 선택, 명령어 스케줄링, 레지스터 할당과 배정 등이 가장 중요한 요소이다.
  - 명령어 선택은 목적 기계에 따라 적절한 명령어를 선택하는 것이고,
  - 레지스터 할당과 배정은 프로그램의 각 명령어가 어느 레지스터에 저장되어야 하는지 결정하는 것이다.
  - 명령어 스케줄링은 명령어들의 실행을 어떤 순서로 나열할지 결정하는 것이다.
- 이를 정리하면 다음과 같다.
  - 명령어 선택은 대부분의 기계가 같은 연산에 대해 다양한 명령어를 가지고 있기 때문에 그 중 무엇을 선택할지 결정하는 것이다.
  - 명령어 스케줄링은 어떤 순서로 계산 과정을 수행할지 결정하는 것이다. 연산 순서에 따라 레지스터의 개수나 메모리의 요구량이 달라지기 때문이다.
  - 레지스터 할당과 배정은 어떤 피연산자를 어떤 레지스터에 어떻게 배정할 것인지에 대한 것이다. 일반적으로 메모리보다 레지스터를 사용하면 연산 속도가 훨씬 빨라지므로 가급적이면 연산에 레지스터를 많이 사용하도록 노력해야 한다.

12.1 목적 코드 생성의 개념

- 목적 컴퓨터는 로드와 저장 연산, 계산 연산, 분기 연산, 조건 분기를 가진 3-주소 기계를 모델링한다.
- 목적 컴퓨터는 n개의 범용 레지스터 R0, R1, ..., $Rn^{-}1$을 가진 바이트 주소 가능 기계(byte-addressable machine)이다.
- 대부분의 명령어는 하나의 연산자와 다음에 오는 하나의 목적 피 연산자, 그 다음에 오는 소스 피연산자로 구성된다.
- 목적 컴퓨터는 다음과 같은 명령어가 사용된다고 가정한다.
  - 로드(LOAD) : LOAD dst, addr은 addr 위치에 포함된 값을 dst 위치에 적재한다.
  - LOAD R1, R2는 레지스터 R2의 내용을 레지스터 R1에 복사한다.
  - 저장(STORE) : STORE x, R0은 레지스터 R0에 포함된 값을 x에 저장한다.
  - OP dst, str : OP는 ADD, SUBT, MULT와 같은 연산자이고 dst와 str에 있는 값을 적용하여 dst에 저장한다.
  - 무조건 분기(unconditional jump) : 명령어 BR L은 제어가 라벨 L로 이동한다. BR은 branch를 나타낸다.
  - 조건 분기(conditional jump) : 명령어 Bcond R, L은 R의 값이 조건(cond)에 맞으면 라벨 L로 분기하라 는 것이다. BLTZ R, L은 R의 값이 0보다 작으면 라벨 L로 분기하라는 것이다.

12.1 목적 코드 생성의 개념

- 위 명령어를 적용하여 다음 3-주소 문장을 레지스터 R0 하나만 사용해서 목적 코드로 변환 해보자.
  - x = y + z
  - u = x + v
- 3-주소 문장은 다음과 같다.
  - LOAD R0, y
  - ADD R0, z
  - STORE x, R0
  - LOAD R0, x
  - ADD R0, v
  - STORE u, R0

12.1 목적 코드 생성의 개념

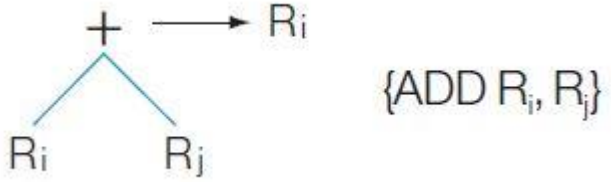
- **[예제 12-1] 기계 명령어로 변환하기**
- 3-주소 문장 x = y - z에 대해 레지스터를 2개 사용하여 기계 명령어로 변환해보자.
- [풀이] 다음과 같이 변환할 수 있다.
  - LOAD R1, y
  - LOAD R2, z
  - SUBT R1, R2 // R1 = R1 - R2
  - STORE x, R1

12.2 트리-패턴 매칭 기법

- 목적 코드 생성이란 흔히 명령어 선택을 가리킨다.
- 예를 들어 x = y + z 형식(여기서 x, y, z는 정적으로 할당된다)의 모든 3-주소 문장은 다음과 같은 코드로 변환될 수 있다.
  - LOAD R0, y // y 값을 레지스터 R0에 적재하라.
  - ADD R0, z // R0 = R0 + z
  - STORE x, R0 // x = R0
- 이러한 기법은 대개 로드(LOAD)와 저장(STORE) 명령어를 중복되게 생성한다. 다음의 3-주소 문장을 살펴보자.
  - x = y + z
  - u = x + v
- 변환하면 다음과 같다.
  - LOAD R0, y
  - ADD R0, z
  - STORE x, R0
  - LOAD R0, x
  - ADD R0, v
  - STORE u, R0
- 그런데 네 번째 문장은 이미 R0에 x가 있으므로 불필요하고, 세 번째 문장도 x가 다음에 사용되지 않으면 불필요하다.

12.2 트리-패턴 매칭 기법

- 명령어 선택 기법 가운데 가장 많이 사용되는 트리-패턴 매칭 기법에 대해 살펴보자.
- 트리-패턴 매칭 기법을 사용하면 명령어 선택을 쉽게 표현할 수 있다.
- 코드 생성을 트리-패턴 매칭으로 변환하려면 중간 코드 형태의 프로그램과 목적 기계의 명령어 집합이 트리로 표현되어야만 한다.
- 트리-패턴 매칭 규칙은 (식 12.1)과 같다.
  - 형판 → 대체 {수행} ... (식 12.1)
  - 여기서 형판(template)은 트리, 대체(replacement)는 단일 노드, 수행(action)은 코드이다.
- 예를 들어 레지스터-레지스터 덧셈 명령어에 대한 규칙을 생각해보자.



  - 이 규칙은 입력 트리가 (식 12.1)의 형판과 매칭되는 부분 트리를 포함하고, 그 트리는 루트가 + 연산자이다. 왼쪽 자식이 레지스터 Ri, 오른쪽 자식이 레지스터 Rj라면 이 부분 트리를 단일 노드 Ri로 대체한 뒤 명령어 ADD Ri, Rj를 출력하는데, 이러한 대체를 부분 트리의 타일링(tiling)이라 한다.

12.2 트리-패턴 매칭 기법

- 목적 기계 명령어에 대한 몇 개의 트리-패턴 매칭 기법은 [그림 12-2]와 같다. 처음 2개의 규칙은 로드 명령어에 대한 것이고, 그 다음 2개의 규칙은 저장 명령어에 대한 것이며, 나머지는 인덱스를 가진 로드와 덧셈 명령어에 대한 것이다. 규칙 ⑧은 상수 값 1을 요구한다. 이러한 조건은 의미 서술자에 의해 서술되어야 한다.

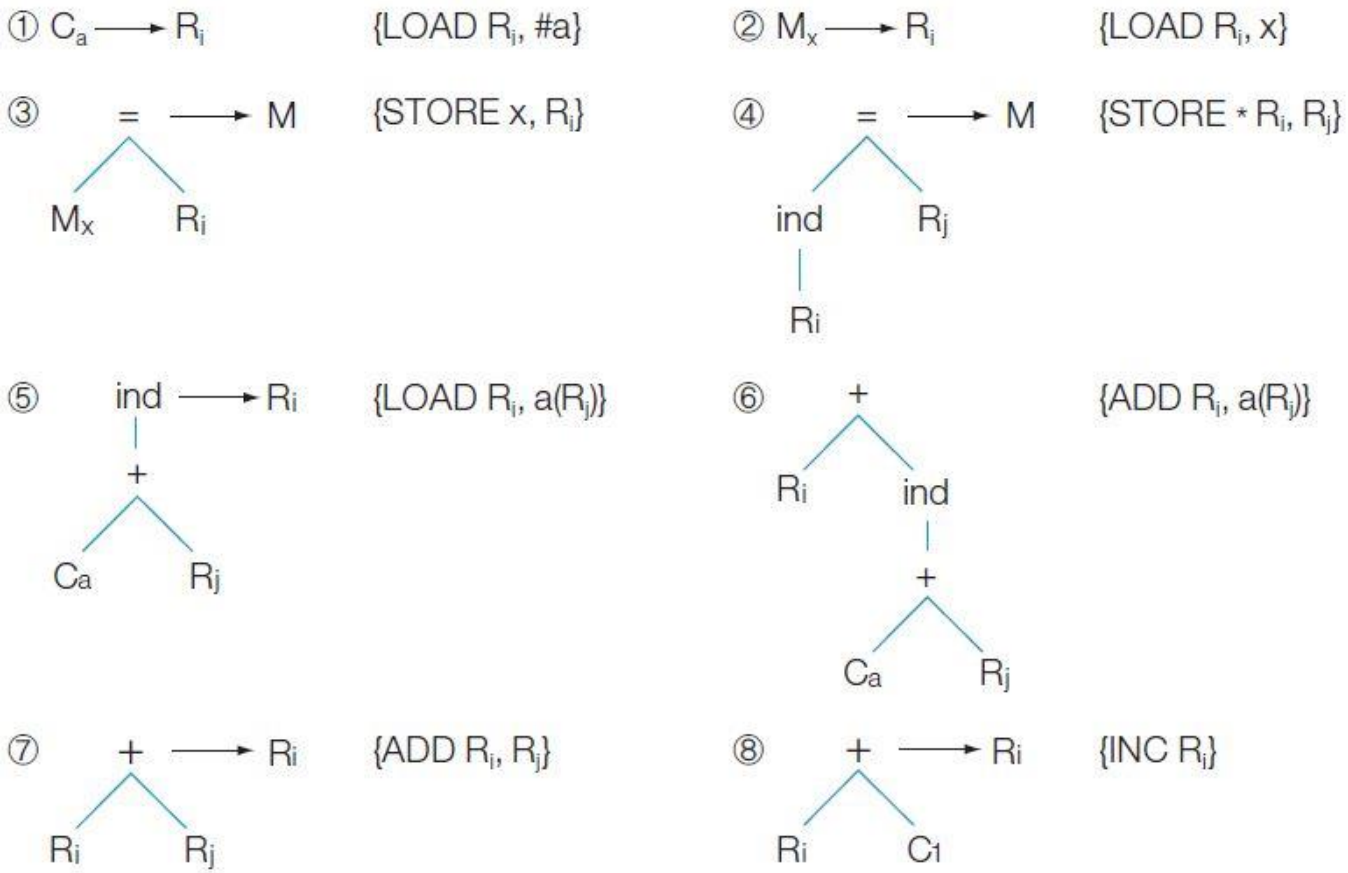


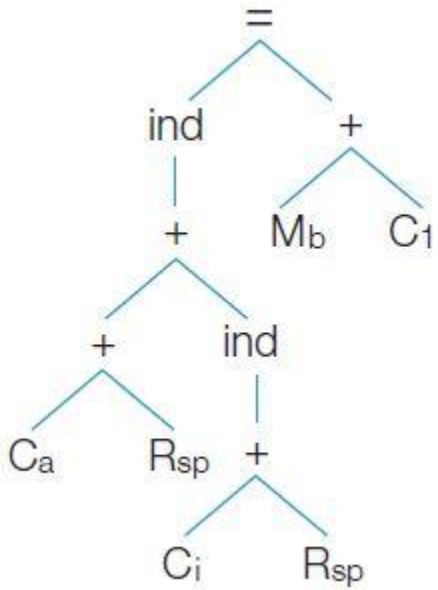
그림 12-2 목적 기계에 대한 트리-패턴 매칭 기법

12.2 트리-패턴 매칭 기법

- 트리-패턴 매칭 기법은 다음과 같이 동작한다.
  - 하나의 입력 트리가 주어지면 트리-패턴 매칭 기법의 형판이 그 부분 트리에 적용된다.
  - 한 형판이 매칭되면 입력 트리에 속한 매칭되는 그 부분 트리가 해당 규칙에 의해 대체되고, 그 규칙에 연관된 행동이 수행된다.
  - 행동이 일련의 기계 명령어를 포함하고 있으면 그 명령어들이 생성된다.
  - 이러한 과정을 트리가 하나의 노드 로 대체될 때까지 또는 더 이상 형판이 매칭되지 않을 때까지 반복한다.
  - 입력 트리가 단일 노드로 감축되면서 생성된 기계 명령어들의 순서는 주어진 입력 트리상에서 트리-패턴 매칭 기법의 출력을 구성한다.

12.2 트리-패턴 매칭 기법

- **[예제 12-2] 중간 코드 트리 만들기**
- 치환문 a[i] = b + 1에 대한 중간 코드 트리를 만들어보자. 지역 변수 a와 i는 현재 활성 레코드의 시작을 가리키는 포인터인 스택 포인터로부터 상수 오프셋 $C_a(Const_a)$와 $C_i(Const_i)$ 를 가지고 있는 지역 변수이고, 배열 a는 실행 시간 스택에 저장된다. a[i]에 대한 치환문은 a[i]의 위치에 대한 r-값이 산술식 b + 1의 r-값에 치환되는 간접 치환문이다. 배열 a와 변수 i의 주소는 각각 상수 C_a와 C_i의 값을 레지스터 스택 포인터의 내용에 더함으로써 주어진다. 여기서는 계산을 간단히 하기 위해 모든 값이 1바이트 문자라고 가정한다.
- [풀이] 치환문 a[i] = b + 1을 트리로 표현하면 다음과 같다.



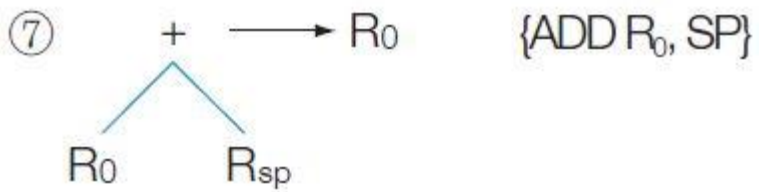
12.2 트리-패턴 매칭 기법

- 이 트리에서 ind 연산자는 그 매개변수를 메모리 주소로 가진다. 치환 연산자의 왼쪽 자식으로서의 ind 노드는 그 치환 연산자의 오른쪽 부분 r-값이 저장될 위치를 가리킨다. + 연산자나 ind 연산자의 한 매개변수가 메모리 위치이거나 레지스터라면 메모리 위치나 레지스터의 내용을 값으로 취한다. 이 트리의 터미널 노드는 속성의 라벨을 갖는데 첨자는 그 속성 값을 나타낸다.

12.2 트리-패턴 매칭 기법

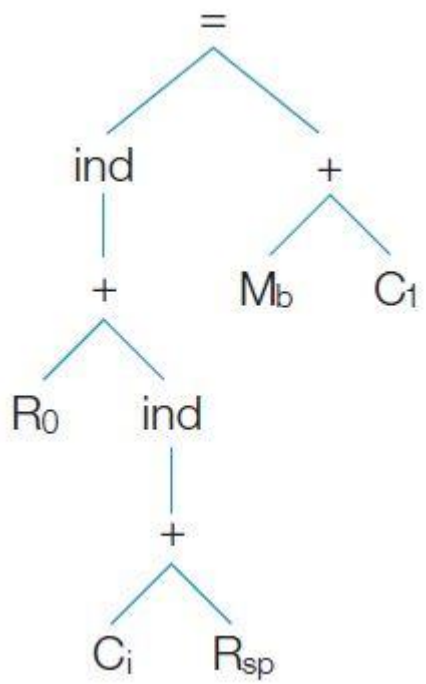
- **[예제 12-3] 트리-패턴 매칭 기법을 사용하여 코드 생성하기**
- [그림 12-2]의 목적 기계에 대한 트리-패턴 매칭 기법을 사용하여 [예제 12-2]의 트리를 입력으로 받아 코드를 생성해보자.
- [풀이] 상수 a를 레지스터 R0에 로드하기 위해 ①번 규칙이 사용된다.

- 그러면 가장 왼쪽 터미널 노드의 라벨은 C_a로부터 R_0으로 변경되고 LOAD R_0, #a가 생성된다. 다음으로 ⑦번 규칙과 부분 트리가 매칭된다



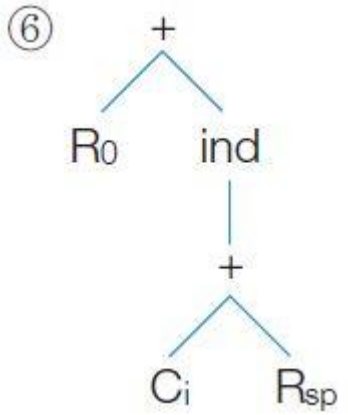
- ⑦번 규칙을 적용하면 이 부분 트리를 R_0의 라벨을 가진 단일 노드로 변경하고 ADD R_0, SP가 생성된다. 이제 트리는 다음과 같다.

12.2 트리-패턴 매칭 기법

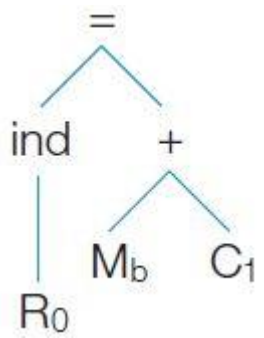


- 여기서 부분 트리를 찾으면 ⑤번과 ⑥번 규칙을 적용할 수 있다. 작은 부분 트리보다는 큰 부분 트리로 대체하는 것이 효율적이므로 ⑥번 규칙과 부분 트리를 매칭한다.

12.2 트리-패턴 매칭 기법

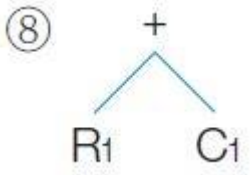


- ⑥번 규칙을 적용하면 이 부분 트리를 R_0의 라벨을 가진 단일 노드로 변경하고 ADD R_0, i(SP)가 생성된다. 이제 트리는 다음과 같다.

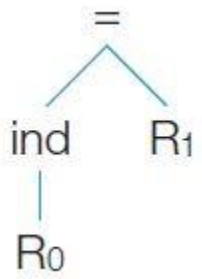


12.2 트리-패턴 매칭 기법

- 오른쪽 부분 트리에서 ②번 규칙을 터미널 노드 M_b에 적용한다. 이 규칙은 b를 레지스터 R_1에 로드 하는 명령어를 생성한다. 즉 LOAD R_1, b가 생성된다.



- 증가 명령어 INC R_1을 생성한다. 이제 트리는 다음과 같다.



- 이 트리는 ④번 규칙과 매칭되므로 트리를 단일 노드로 감축하고 명령어STORE $*R_0$, R_1을 생성한다.

12.2 트리-패턴 매칭 기법

- 지금까지 생성된 코드는 다음과 같다.
- LOAD R_0, #a
- ADD R_0, SP
- ADD R_0, i(SP)
- LOAD R_1, b
- INC R_1
- STORE *R_0, R_1

12.3 목적 코드 생성

- **산술식에 대한 목적 코드 생성**
  - 산술식에 대한 목적 코드 생성은 구성된 시스템의 형태에 따라 모두 다르다.
  - 가장 쉬운 예로 연산 레지스터가 1개이고 중간 언어가 트리 구조라면 목적 코드를 생성하는 방법은 [알고리즘 12-1]과 같다.
- **[알고리즘 12-1]** 연산 레지스터 1개를 사용하여 트리 구조로부터 목적코드 생성
- [입력] 중간 언어로 트리 구조(단, 연산 레지스터를 Acc로 표현한다)
- [출력] 산술식에 대한 목적 코드 생성
- [방법]
- ❶ 트리에서 루트로부터 순회한다.
- ❷ 왼쪽 부분 트리가 하나의 변수이고 오른쪽 부분 트리가 하나의 변수이거나 Acc 또는 wi이면 목적 코드 생성 루틴을 호출한다. 왼쪽 부분 트리가 Acc이고 오른쪽 부분 트리가 하나의 변수나 wi이면 목적 코드 생성 루틴을 호출한다. 또한 왼쪽 부분 트리가 부분 트리이고 오른쪽 부분 트리가 Acc이면 목적 코드 생성 루틴을 호출한다.위와 같은 경우가 아닐 때, 2개의 부분 트리 중 한쪽이 하나의 변수이거나 Acc 또는 wi이면 다른 쪽 부분 트리를 순회한다. 이때 부분 트리에 대해 루트와 왼쪽 부분 트리, 오른쪽 부분 트리의 이름은 T, L, R로 다시 정의한다.
- ❸ 2개의 부분 트리가 모두 터미널 노드가 아니면 오른쪽 부분 트리를 먼저 순회한다.

12.3 목적 코드 생성

- ❹ 계속 같은 방법으로 순회하다가 양쪽의 부분 트리가 모두 터미널 노드이면 목적 코드 생성 루틴을 호출한다.
- ❺ 터미널 노드에 대한 목적 코드 생성 루틴이 복귀되면 그 부분의 트리 대신 Acc로 노드를 바꾼다.
- ❻ 구문 트리에서 루트에 대해 목적 코드 생성 루틴을 호출하면 목적 코드를 생성하고 알고리즘을 끝내며, 루트에 대한 목적 코드 생성 루틴이 아니면 ❷번으로 돌아가서 계속 반복한다.

12.3 목적 코드 생성

- 목적 코드 생성 루틴은 [표 12-1]과 같이 재귀적 프로시저로 표현할 수 있다.

표 12-1 목적 코드 생성 루틴

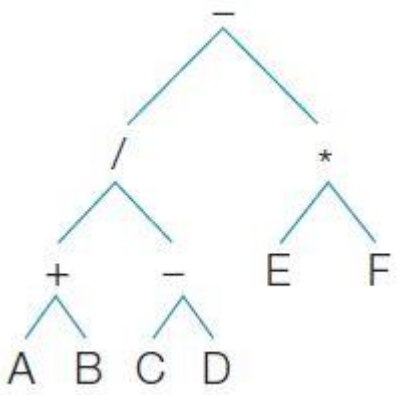
L \ R	변수 V_2	Acc	W_1
변수 V_1	LOAD R_0, V_1 op′ R_0, V_2 T ← Acc	+, *일 때 op′ R_0, V_1 T ← Acc −, /일 때 i ← i+1 STORE W_i, R_0 R ← W_i	LOAD R_0, V_1 op′ R_0, W_i T ← Acc i ← i−1
Acc	op′ R_0, V_2 T ← Acc		op′ R_0, W_i T ← Acc i ← i−1
부분 트리		i ← i+1 STORE W_i, R_0 R ← W_i	

* 단, i의 초깃값은 0이다.

12.3 목적 코드 생성

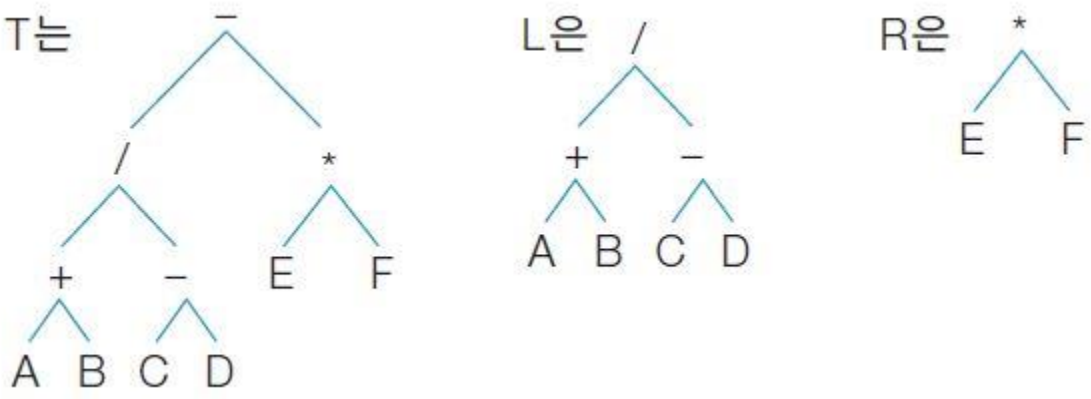
- 여기서 부분 트리 T는 연산자 op와 왼쪽 부분 트리 L, 오른쪽 부분 트리 R로 이뤄진다. 부분 트리에는 피연산자인 변수와 Acc, 부분 트리가 있다. op′는 op를 실행하는 기계어 명령 코드로 op가 +이면 ADD가 된다. T ← Acc는 T를 연산 레지스터 표시로 치환하라는 의미이고, R ← Wi는 부분 트리 R을 Wi로 치환하라는 의미이다.

- **[예제 12-4] 목적 코드 생성하기**
- 산술식 (A + B) / (C - D) - E * F에 대한 구문 트리가 다음과 같을 때 [알고리즘 12-1]을 사용하여 목적 코드를 구해보자.



- [풀이] [알고리즘 12-1]에 따라 처음에 T, R, L은 다음과 같다.

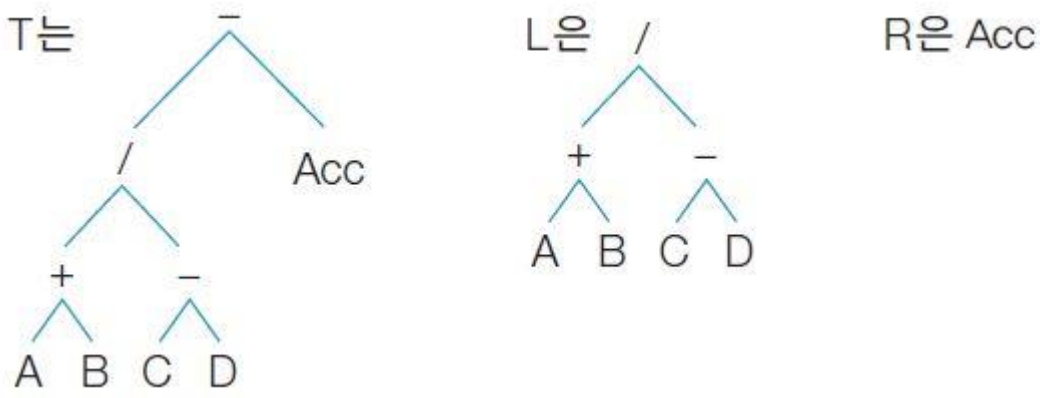
12.3 목적 코드 생성



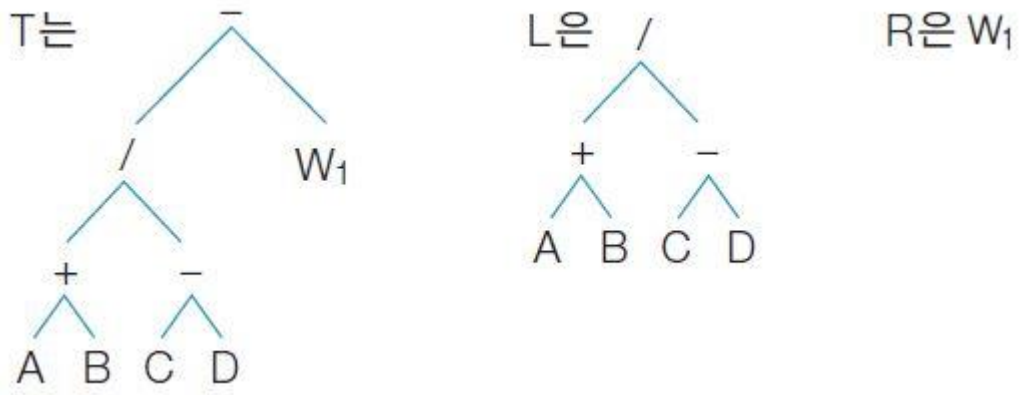
- [알고리즘 12-1]의 ❸에 따라 L과 R이 모두 터미널 노드인지 확인한다. L과 R이 모두 터미널 노드가 아니므로 오른쪽 부분 트리 R을 순회한다. 오른쪽 부분 트리에 대해 다시 T, L, R을 구한다.

- L과 R이 모두 터미널 노드이므로 [알고리즘 12-1]의 ❹에 의해 목적 코드를 생성한다. [표 12-1]의 목적 코드 생성 루틴을 호출한다. L과 R이 모두 변수이므로 목적 코드는 다음과 같다.
- LOAD R_0, E
- MULT R_0, F
- 그리고 T는 Acc로 치환한다. 이제 T, L, R은 다음과 같다.

12.3 목적 코드 생성

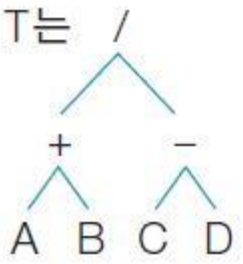


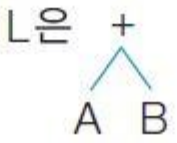
- 다시 [알고리즘 12-1]의 ❷에 의해 왼쪽이 부분 트리이고 오른쪽이 Acc이므로 [표 12-1]의 목적 코드 생성 루틴을 호출한다. i = 1이 되고 STORE W_1, R_0이 생성된다. 그리고 R은 W_1이 된다. 이제 T, L, R은 다음과 같다.

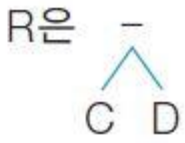


12.3 목적 코드 생성

- 다시 [알고리즘 12-1]을 수행한다. [알고리즘 12-1]의 ❷에 의해 부분 트리인 L을 T로 바꾸고 L과 R을 구한다. 그러면 T, L, R은 다음과 같다.



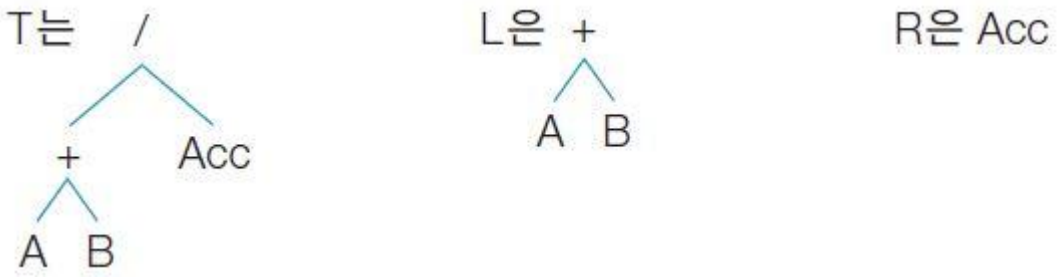




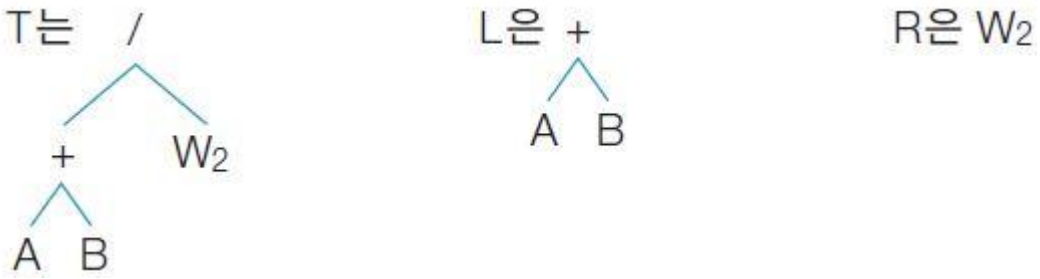
- [알고리즘 12-1]의 ❸에 의해 L과 R이 모두 터미널 노드가 아니므로 오른쪽 부분 트리 R을 순회한다. 오른쪽 부분 트리에 대해 다시 T, L, R을 구한다.

- [알고리즘 12-1]의 ❹에 해당되므로 [표 12-1]의 목적 코드 생성 루틴을 호출한다. L과 R이 모두 변수이므로 목적 코드는 다음과 같다.
- LOAD R_0, C
- SUBT R_0, D
- 그리고 T는 Acc로 치환한다. 이제 T, L, R은 다음과 같다.

12.3 목적 코드 생성



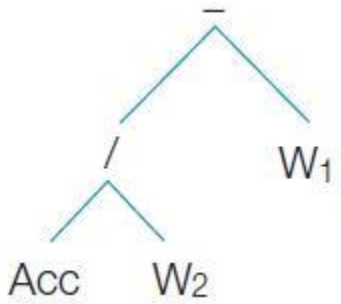
- 다시 [알고리즘 12-1]의 ❷에 의해 왼쪽이 부분 트리이고 오른쪽이 Acc이므로 [표 12-1]의 목적 코드 생성 루틴을 호출한다. i = 2가 되고 STORE W_2, R_0이 생성된다. 그리고 R은 W_2가 된다. 이 제 T, L, R은 다음과 같다.



- 다시 [알고리즘 12-1]을 수행한다. [알고리즘 12-1]의 ❷에 의해 부분 트리인 L을 T로 바꾸고 L과 R을 구한다. 그러면 T, L, R은 다음과 같다.

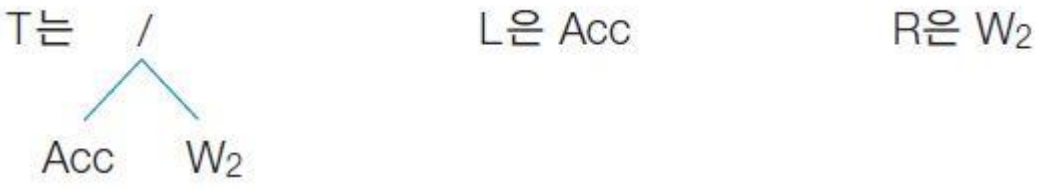
12.3 목적 코드 생성

- [알고리즘 12-1]의 ❹에 해당되므로 [표 12-1]의 목적 코드 생성 루틴을 호출한다. L과 R이 모두 변수이므로 목적 코드는 다음과 같다.
- LOAD R_0, A
- ADD R_0, B
- 이제 T는 Acc로 치환한다. 다시 T, L, R을 구하기 전에 지금까지의 전체 구문 트리를 그려보자.

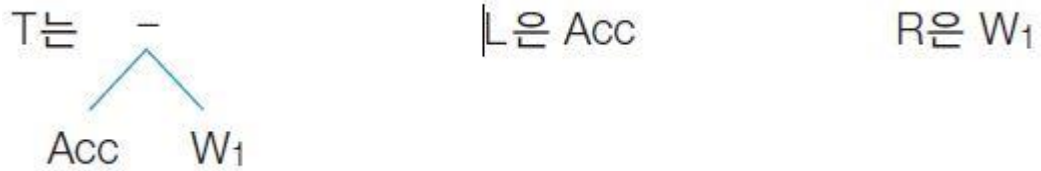


12.3 목적 코드 생성

- 알고리즘을 다시 적용하여 T, L, R을 구하면 다음과 같다.



- [알고리즘 12-1]의 ❷에 해당되므로 [표 12-1]의 목적 코드 생성 루틴을 호출한다. 목적 코드는 다음과 같다.
- DIV R_0, W_2
- 이제 T는 Acc이므로 i = 1로 치환한다. 다시 T, L, R은 다음과 같다.



- [알고리즘 12-1]의 ❷에 해당되므로 [표 12-1]의 목적 코드 생성 루틴을 호출한다. 목적 코드는 다음과 같다.
- SUBT R_0, W_1
- 이제 T는 Acc로 치환하고 i = 0이 된다. 전체 구문 트리의 루트에 대해 목적 코드를 생성했으므로 알고리즘을 끝낸다.
- 즉 산술식 (A + B) / (C - D) - E * F에 대해 이제까지 생성된 목적 코드를 나열하면 다음과 같다.

12.3 목적 코드 생성

- 즉 산술식 (A + B) / (C - D) - E * F에 대해 이제까지 생성된 목적 코드를 나열하면 다음과 같다.
- LOAD R_0, E
- MULT R_0 , F
- STORE W_1, R_0
- LOAD R_0 , C
- SUBT R_0 , D
- STORE W_2, R_0
- LOAD R_0 , A
- ADD R_0 , B
- DIV R_0 , W_2
- SUBT R_0 , W_1

12.3 목적 코드 생성

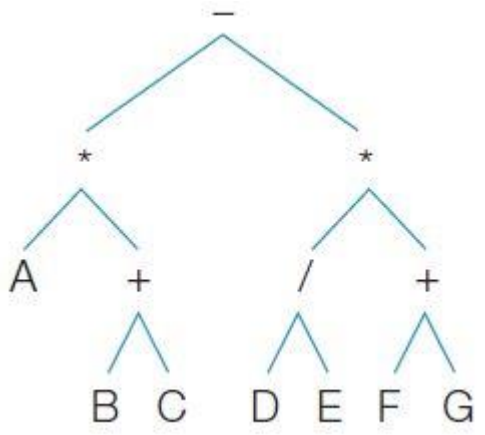
- 연산 레지스터가 여러 개인 경우에는 목적 코드를 어떻게 생성할까? 프로그램에서 구문 트리를 만들 때 각 노드의 부분 트리를 계산하는 데 필요한 레지스터의 수를 먼저 계산한 다음 구문 트리에 표시하면 된다. 이런 방법을 사용하면 한 번도 STORE 명령을 사용하지 않고 계산에 필요한 레지스터의 수를 구할 수 있다. 레지스터의 수를 계산하여 레지스터의 수를 라벨로 가진 구문 트리(즉 구문 트리에 라벨을 붙인)에서 목적 코드를 생성해낸다.
- 식 E를 계산하는 데 필요한 레지스터의 수를 CR(E)로 표시하면 [알고리즘 12-2]에 의해 구문 트리로부터 CE(E) 값을 구할 수 있다. 라벨을 붙이는 순서는 터미널 노드에서 루트 노드로 진행하며, 필요한 레지스터의 수가 큰 쪽의 부분 트리부터 목적 코드로 변환해가면 된다.

12.3 목적 코드 생성

- **[알고리즘 12-2] 레지스터 수(CR) 계산**
- [입력] 구문 트리
- [출력] 라벨을 가진 구문 트리
- [방법] begin
- if 노드 n이 터미널 노드 then
- begin
- if n이 자식들 중에서 가장 왼쪽 끝 자식 노드
- then CE(n) = 1
- else CE(n) = 0
- end
- else
- if 자식 노드들의 라벨이 l1, l2인 경우 then
- begin
- if l1 ≠ l2 then CE(n) = max(l1, l2)
- else CE(n) = l1+1
- end
- end.

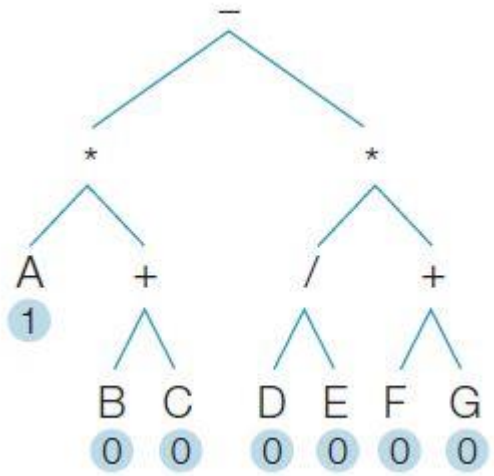
12.3 목적 코드 생성

- **[예제 12-5] 라벨을 가진 구문 트리 만들기 1**
- 산술식 A * (B + C) - D / E * (F + G)에 대해 레지스터 수를 라벨로 가진 구문 트리를 만들어보자.
- [풀이] 먼저 구문 트리를 만들면 다음과 같다.

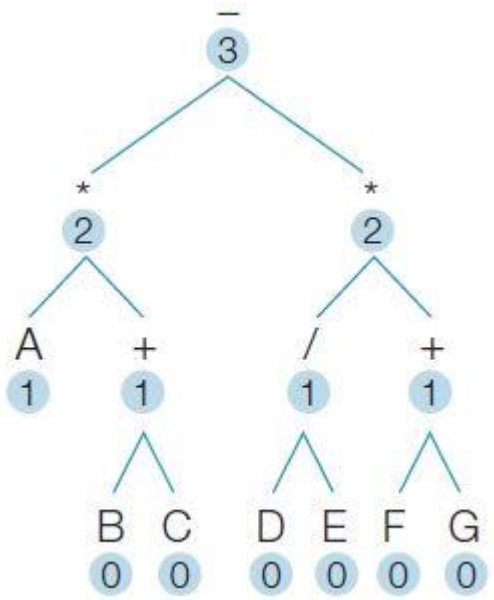


- 위의 구문 트리에 [알고리즘 12-2]를 적용해보자. 첫 번째 단계로 터미널 노드에 대해 적용한다. 터미널 노드 중 가장 왼쪽에 있는 노드만 라벨1 을 갖고 나머지 터미널 노드는 모두 라벨0 을 갖는다.

12.3 목적 코드 생성



- 이제 터미널 노드가 아닌 다른 노드에 대한 라벨을 계산할 수 있다. 계속 적용하면 다음과 같은 라벨을 가진 구문 트리를 만들 수 있다.



12.3 목적 코드 생성

- 레지스터 수에 대한 라벨을 가진 구문 트리로부터 목적 코드를 생성하는 방법은 [알고리즘 12-3]과 같다.
- **[알고리즘 12-3]** 여러 개의 연산 레지스터에 대한 라벨을 가진 구문 트리로부터 목적 코드생성
- [입력] 여러 개의 연산 레지스터에 대한 라벨을 가진 구문 트리
- [출력] 목적 코드
- [방법]
- ❶ 라벨을 가진 구문 트리의 루트 노드를 순회한다.
- ❷ 만약 양쪽에 있는 자식의 라벨이 모두 0이 아니라면 다음을 재귀적으로 수행한다. 왼쪽 부분 트리와 오른쪽 부분 트리 중 라벨이 큰 쪽의 부분 트리를 먼저 순회한다. 만약 왼쪽 부분 트리와 오른쪽 부분 트리의 라벨이 같으면 오른쪽 부분 트리를 먼저 순회한다 .
- ❸ 만약 양쪽에 있는 자식의 라벨이 모두 0이면 자식 노드의 목적 코드를 생성한다. 라벨에 있는 레지스터 번호로 그 노드를 치환하고(라벨은 0으로 한다) 이 노드 위에 있는 부모 노드로 되돌아가서 ❷, ❸을 다시 수행한다.

12.3 목적 코드 생성

- **[예제 12-7] 목적 코드 생성하기 1**
- [예제 12-6]의 라벨을 가진 구문 트리에 [알고리즘 12-3]을 적용하여 목적 코드를 생성해 보자.
- [풀이]
- [예제 12-6]의 라벨을 가진 구문 트리에 [알고리즘 12-3]을 적용한다. 루트로부터 출발하여 왼쪽 부분 트리와 오른쪽 부분 트리에 대한 라벨을 보면 왼쪽 부분 트리가 더 크므로 왼쪽 부분 트리를 순회한다. 왼쪽 부분 트리에서 다시 왼쪽과 오른쪽 부분 트리에 대한 라벨을 보면 같기 때문에 이번 에는 오른쪽 부분 트리를 순회한다.
- 여기서 자식들의 라벨이 모두 0이므로 3-주소 코드로 표현하면 다음과 같다.
- R1 ← C - D
- 이 부분 트리를 R1로 대체하고 라벨은 0으로 한다. 부모 노드에 가서 양쪽 부분 트리를 보면 왼쪽 부분 트리의 라벨이 더 크다. 여기서 자식 노드를 보면 둘 다 라벨이 0이므로 3-주소 코드로 표현하면 다음과 같다.
- R2 ← A + B
- 이 부분 트리를 R2로 대체하고 라벨은 0으로 한다. 부모 노드에 가서 양쪽 자식 노드를 보면 둘 다 라벨이 0이므로 3-주소 코드로 표현하면 다음과 같다.
- R2 ← R2 / R1

12.3 목적 코드 생성

- 이 부분 트리를 R2로 대체하고 라벨은 0으로 한다. 부모 노드에 가서 양쪽 부분 트리를 보면 오른쪽 부분 트리의 라벨이 더 크다. 여기서 자식 노드를 보면 둘 다 라벨이 0이므로 3-주소 코드로 표현하면 다음과 같다.
- R1 ← E * F
- 이 부분 트리를 R1로 대체하고 라벨은 0으로 한다. 부모 노드에 가서 양쪽 자식 노드를 보면 둘 다 라벨이 0이므로 3-주소 코드로 표현하면 다음과 같다.
- R2 ← R2 - R1
- 3-주소 중간 코드를 나열하면 다음과 같다.
- R1 ← C - D
- R2 ← A + B
- R2 ← R2 / R1
- R1 ← E * F
- R2 ← R2 - R1

12.3 목적 코드 생성

- 이 3-주소 중간 코드를 가지고 목적 코드로 바꾸면 다음과 같다.
- LOAD R1, C
- SUBT R1, D
- LOAD R2, A
- ADD R2, B
- DIV R2, R1
- LOAD R1, E
- MULT R1, F
- SUBT R2, R1

12.3 목적 코드 생성

- 이와 같이 라벨을 가진 구문 트리에서 가장 큰 라벨은 2이므로 연산 레지스터가 2개 이상 있는 컴퓨터에 대해서는 한 번도 STORE를 실행하지 않는 목적 코드를 생성할 수 있다.
- [예제 12-4]와 [예제 12-7]을 비교해보면 [예제 12-4]의 목적 코드는 연산 레지스터를 하나만 사용했고 [예제 12-7]의 목적 코드는 연산 레지스터를 2개 사용했다. 그런데 여기서도 문제가 발생한다. 라벨을 가진 구문 트리를 구한 다음 목적 코드를 생성하려고 하는데 라벨의 최대값이 컴퓨터가 가지고 있는 연산 레지스터의 최대 개수보다 많으면 어떻게 할까? 이런 경우에는 도중에 STORE 문을 사용해야 한다. [예제 12-4]와 [예제 12-7]을 비교해보았을 때 예상할 수 있었을 것이다. 이러한 문제를 해결하기 위한 방법은 [알고리즘12-4]와 같다.

12.3 목적 코드 생성

- **[알고리즘 12-4] 목적 코드 생성**
- [입력] 라벨을 가진 구문 트리
- [출력] 목적 코드
- [방법]
- 1) 레지스터 R0, R1, ..., Rn을 사용할 수 있는 것으로 하고 트리의 루트 노드부터 아래 2)를 적용한다.
- 2) 노드 m에 대해 Ri, Ri+1, ..., Rn이 사용 가능한 레지스터라고 하자.
- a) 노드 m이 터미널 노드라면 m의 값을 Ri에 넣는다. 즉 목적 코드 LOAD Ri, m을 생성한다.
- b) 노드 m이 터미널 노드가 아닌 경우, m의 자식 노드를 m1, m2라고 하자.
- ❶ L ABEL(m1) ≧ N, LABEL(m2) ≧ N이면 오른쪽 자식 노드에 대해 2)를 적용하고 얻은 값을 STORE 한다. 다음으로 왼쪽 자식 노드에 대해 2)를 적용하여 값을 Ri에 넣는다. Ri와 STORE 해놓은 값을 연산하여 결과를 Ri에 넣는다.
- ❷ L ABEL(m1) ≠ LABEL(m2)로 적어도 한쪽이 N보다 작다고 하자. 큰 라벨을 가진 자식 노드에 대해 2)를 적용하여 결과를 Ri에 넣는다. 다음으로 작은 라벨의 자식 노드에 대해 Ri+1, ..., Rn을 사용 할 수 있는 것으로 하여 2)를 적용한다. 결과가 Ri+1에 저장되어 있으므로 그것과 Ri를 연산하여 Ri에 넣는다. 작은 라벨이 0일 때 직접 그 자식 노드와 Ri를 연산하여 결과를 Ri에 넣는다.
- ❸ LABEL(m1) = LABEL(m2) < N일 때 왼쪽에 있는 자식 노드의 라벨이 크다고 생각하여 ❷를 수행 한다.

12.3 목적 코드 생성

- **[예제 12-8] 목적 코드 생성하기 2**
- [예제 12-5]에 주어진 산술식에 대해 목적 코드를 생성해보자. 단, 사용할 수 있는 레지스터 의 수는 N = 2라고 가정한다. [예제 12-5]에 주어진 산술식에 대해 라벨을 가진 구문 트리 를 만들어보면 STORE를 하지 않는 목적 코드를 생성하기 위해 연산 레지스터 3개가 필요하다. 하지만 연산 레지스터 2개를 이용하여 목적 코드를 생성해보자.
- [풀이]
- [예제 12-5]에서 라벨을 가진 구문 트리를 만들었는데 이 구문 트리에 [알고리즘 12-4]를 적용해 보자. 그러면 다음과 같은 목적 코드가 생성된다.
- LOAD R0, D
- MULT R0, E
- LOAD R1, F
- ADD R1, G
- MULT R0, R1
- STORE R0, W
- LOAD R0, A
- LOAD R1, B
- ADD R1, C
- DIV R0, R1
- SUBT R0, W

12.4 레지스터 할당과 배정

- 코드 생성에서 가장 중요한 핵심은 어떤 값을 어떤 레지스터에 저장할지 결정하는 것
- 레지스터는 메모리 계층에서 가장 빠른 장치이지만, 모든 값을 저장할 수 있는 충분한 레지스터를 갖기는 힘들기 때문에 레지스터에 저장되지 않은 값을 메모리에 저장할 필요가 있다.
- 프로그램의 각 위치에서 어떤 값을 레지스터에 저장할 것인지, 즉 레지스터 할당(register allocation) 그리고 각 값을 어떤 레지스터에 저장할 것인지, 즉 레지스터 배정(register assignment)에 대해 다양한 기법을 살펴본다.
- 오늘날의 컴파일러에서 레지스터 할당기(register allocator) 의 목표는 목적 기계에 의해 제공되는 레지스터를 효율적으로 사용하는 것으로 레지스터 할당과 레지스터 배정이라는 두 가지 문제를 동시에 해결한다.
- 해당 할당기는 어떤 값이 안전하게 레지스터에 저장될 수 있는지 결정해야 한다. 그리고 과연 레지스터에 저장하는 것이 유용한지도 판단해야 한다. 좋은 성능을 얻기 위해 할당기는 가능한 한 많은 메모리 기반 값을 레지스터에 할당할 필요가 있다

12.4 레지스터 할당과 배정

지역 레지스터 할당과 배정

- 현재 대부분의 컴퓨터는 일반 목적 레지스터와 부동 소수점 레지스터를 가지고 있다.
  - 일반 목적 레지스터는 정수 값과 메모리 주소를 저장하는 데 사용되고, 부동 소수점 레지스터는 실수 값을 저장하는 데 사용된다.
  - 하나의 기본 블록 안에서 레지스터 할당 방법을 지역 레지스터 할당이라 한다.
- 지역 레지스터 할당에서 발생할 수 있는 문제를 생각해보자. 문제를 단순화하기 위해 기본 블록에서 시작해서 끝나는 프로그램이 있다고 하자. 이 프로그램은 이전에 실행되는 기본 블록으로부터 어떠한 값도 전달받지 않으며, 이후에 실행되는 기본 블록을 위해 어떠한 값도 남겨두지 않는다. 목적 기계는 k개의 일반 목적 레지스터만 제공한다. 이 문제에 대해 두 가지 방법을 살펴보자.
  - 하향 방식(top-down)은 해당 기본 블록의 한 값에 대한 참조의 개수를 세고, 이 빈도수를 사용하여 어떤 값을 레지스터를 통해 유지할지 결정하는 것이다. 이처럼 참조 빈도수에 의존하기 때문에 하향 방식이라고 한다.
  - 상향 방식(bottom-up)은 해당 블록 전체를 각 연산별로 확인하면서 레지스터 고르기(register spill)가 필요한지 여부를 결정한다. 어떤 계산을 할 때 레지스터가 필요한데 이미 가능한 모든 레지스터가 사용되고 있을 때, 레지스터를 확보하기 위해 사용되고 있는 레지스터 중에서 하나를 선택하여 그 내용을 메모리 위치에 저장한다. 이렇게 레지스터를 선택하는 것을 레지스터 고르기라고 하며, 이는 많은 저차원(low-level) 사실을 사용하기 때문에 상향 방식이라고 하는 것

12.4 레지스터 할당과 배정

- 하향 지역 할당기는 가장 많이 사용되는 값이 레지스터를 사용해야 한다는 기본 원칙을 토대로 동작한다. 이 방식을 구현하기 위해 지역 할당기는 해당 블록에서 가상 레지스터가 발생 하는 횟수를 세고 이 빈도수를 사용하여 가상 레지스터를 실제 레지스터에 할당한다.
  - 하향 지역 할당 방식의 기본은 자주 사용되는 가상 레지스터에 실제 레지스터를 할당한다는 것이다. 이 방식의 가장 큰 단점은 전체 기본 블록에 대해 하나의 실제 레지스터를 하나의 가상 레지스터를 위해 사용한다는 데 있다. 따라서 하나의 값이 해당 기본 블록의 앞부분에서 자주 사용되고 뒷부분에서 사용되지 않는데도 블록 전체에서 해당 값을 위해 하나의 실질적인 레지스터가 사용되기 때문에 해당 블록의 뒷부분에서 실제 레지스터가 더 이상 사용되지 않는다.

12.4 레지스터 할당과 배정

- 이런 문제점들을 해결하기 위한 방식이 상향 지역 레지스터 할당이다.
  - 이는 각 연산이 실행될 때 발생하는 레지스터 할당에 집중하는 방법으로, 사용되고 있지 않은 모든 레지스터를 사용하여 시작된다. 상향 지역 할당기는 각 연산이 실행되기 전에 피연산자가 반드시 레지스터에 저장되어 있어야 한다는 가정에서 시작한다. 연산의 결과를 위해서도 하나의 레지스터를 할당해야 한다.
  - 상향 지역 할당기는 해당 블록 내에서 연산을 반복적으로 분석하면서 할당에 필요한 결정을 수행한다. 상향 지역 할당 방법은 먼저 실제 레지스터가 사용되지 않는다고 간주하고 실제 레지스터를 무료 리스트(free list)에 추가한다. 첫 번째 몇몇 연산 동안 무료 리스트로부터 실제 레지스터를 반환하여 사용한다. 할당기가 다른 레지스터를 필요로 하고 무료 리스트가 비어 있음을 발견하면 한 레지스터의 값을 메모리로 옮긴다. 이때 메모리에 옮기는 값은 할당기가 다음에 사용하는 값 중에서 가장 나중에 사용할 값이다. 하나의 값을 고르도록 하는 데 드는 비용이 모든 레지스터에 대해 동일하다면 가장 긴 시간 동안 사용되는 레지스터를 선택한다. 몇몇 측면에서 이는 레지스터 고르기의 비용으로 얻는 이득을 최대화한다.

12.4 레지스터 할당과 배정

■ 전역 레지스터 할당과 배정

- 일반적인 코드 생성 알고리즘은 단일 기본 블록의 존속 기간 동안 변수 값을 레지스터에 저장하지만, 유효한 모든 변수는 각 블록의 끝 지점에서 레지스터에 저장된다. 레지스터에의 저장과 상응하는 로드 명령어들이 실행되면서도 값을 유지하려면 빈번하게 사용되는 변수를 레지스터에 배정하고, 배정된 레지스터들이 여러 블록에 걸쳐서(전역적) 계속 이 값을 유지하도록 해야 한다. 즉 프로그램 실행 시간의 대부분을 내부 루프에서 소비하므로 루프의 처음부터 끝까지 빈번하게 사용되는 값을 고정된 레지스터에 놓고 계속 사용하는 것이 전역 레지스터 할당(global register allocation) 방법이다. 때때로 흐름 그래프의 루프 구조를 잘 알고 있다면 기본 블록에서 계산된 값 가운데 어떤 값이 그 블록 외부에서도 사용되는지 알게 될 것이다. 기본 블록에서 계산된 값 가운데 어떤 값이 그 블록 외부에서도 사용되는지를 계산하는 데는 대그(DAG)를 이용할 수 있다.
- 전역 레지스터 할당은 각 내부 루프에서 가장 활발하게 사용되는 값을 저장하기 위해 레지스터의 고정된 개수를 배정하는 것이다. 그러면 할당되지 않은 레지스터는 기본 블록에서 지역 적인 값을 저장하는 데 사용된다. 하지만 이 방법은 전역 레지스터 할당에 사용된 레지스터의 개수가 전역 레지스터에서 필요한 정확한 개수가 아닐 수 있다는 것이 단점이다.

12.4 레지스터 할당과 배정

그래프 착색에 의한 레지스터 할당

- 어떤 계산을 할 때 레지스터가 필요한데 가능한 레지스터가 이미 모두 사용 중일 때, 레지스터를 확보하기 위해 이미 사용하고 있는 레지스터 중에서 하나를 선택하여 그 내용을 메모리 위치에 저장해야 한다. 이와 같이 레지스터를 선택하는 것을 레지스터 고르기라고 하며, 그래프 착색(graph coloring)은 간편하고 체계적으로 레지스터를 할당하고 고르는 방법이다.
- 그래프 착색에는 2-패스가 사용된다. 첫 번째 패스에서는 제한된 레지스터를 가지고 목적 기계 명령어를 선택한다. 효율성을 위해 중간 코드에서 사용된 이름은 레지스터 이름이 되고, 3-주소 명령어는 기계어 명령어로 바뀐다.
- 두 번째 패스는 기호 레지스터(symbolic register)에 실제 레지스터를 배정하고 레지스터 고르기의 비용을 최소화하는 방법을 찾는 것이다. 두 번째 패스에서는 각 프로시저마다 레지스터 간섭 그래프(register-interference graph)를 구성한다. 여기서 노드는 기호 레지스터이다. 한 노드가 다른 노드가 정의되는 지점에서 살아 있으면 이 두 노드를 간선으로 연결한다. 내부 루프의 흐름 그래프인 [그림 12-3]을 살펴보자.
- [그림 12-3]에서 레지스터 R0, R1, R2가 루프 내에서 할당될 수 있다고 가정하자. 각 블록의 진입과 진출에서 유효한 변수 각각을 블록의 위아래에 표시했다. e와 f는 모두 B1의 끝에서 유효하나 이 중에서 e만이 B2의 진입에서 유효하고 f만이 B2의 진입에서 유효하다. 또한 이름 a와 d를 생각해보자. 블록 B1에서 a는 d를 정의하는 두 번째 문장에서 유효하다. 따라서 그래프에서 a 노드와 d 노드를 간선으로 연결한다.

12.4 레지스터 할당과 배정

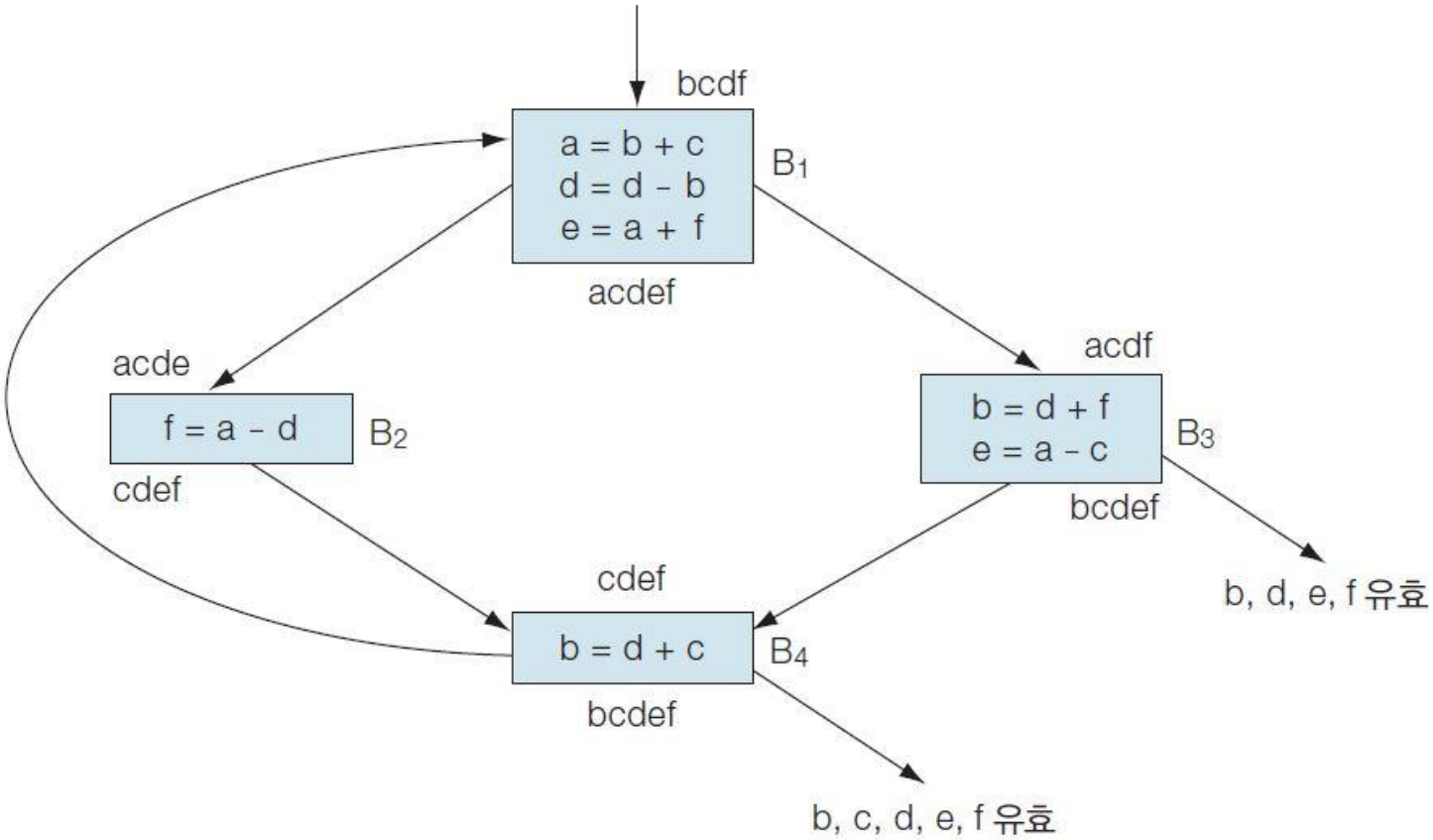


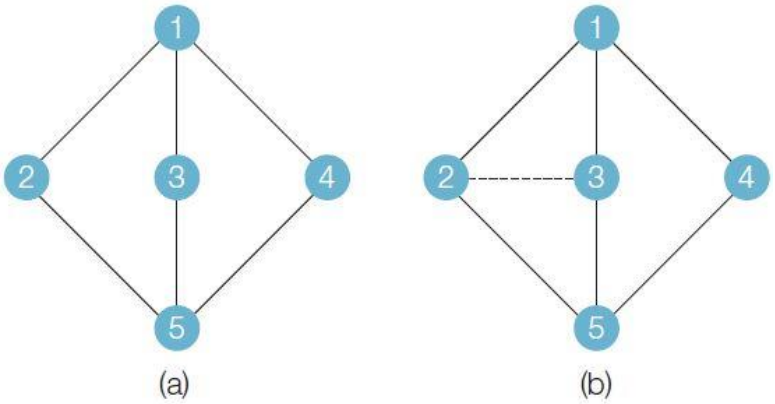
그림 12-3 내부 루프의 흐름 그래프

- 배정 가능한 레지스터의 개수를 k라고 할 때, k-착색(k-coloring) 알고리즘을 이용하여 레지스터-간섭 그래프를 색칠할 수 있다. 인접한 2개의 노드가 동일한 색을 가지지 않는 방식으로 각 노드가 한 색을 배정받았다면 그 그래프는 착색되었다고(colored) 한다. 여기서 각각의 색은 레지스터를 나타낸다.

12.4 레지스터 할당과 배정

- **주어진 그래프가 k-착색 가능한지를 결정하는 문제**
  - 일반적으로 NP-완전이지만, 일반적으로 경험적인 기법(heuristic technique)을 이용하여 그래프 착색을 빠르게 할 수 있다.
  - 그래프 G의 노드 n이 k개보다 더 적은 개수의 인접 노드를 갖는다고 가정하자. n과 그 간선을 G에서 제거하여 그래프 G′를 구한다. G′의 k-착색은 노드 n에는 노드 n의 인접 노드 중에 배정되지 않은 한 색을 배정함으로써 G의 k-착색으로 확장될 수 있다.
- 레지스터-간섭 그래프로부터 k개보다 더 적은 개수의 간선을 가진 노드를 반복적으로 제거함으로써 공백 그래프(empty graph)를 얻게 된다. 이 경우 없어진 순서의 역순으로 노드를 색칠함으로써 원래 그래프에 대한 k-착색을 할 수 있다.
- 또한 각 노드가 k개보다 더 많은 인접 노드를 가진 그래프를 구하게 되는데, 이 경우에 k-착색을 더 이상 수행할 수 없다. 이때 한 노드가 해당 레지스터를 저장하고 재로딩하는(reload) 레지스터 고르기가 일어난다. 이렇게 되면 레지스터 간섭 그래프를 수정하고 색칠하기가 다시 수행된다. 일반적인 내부 루프에서는 레지스터 고르기를 야기하는 코드를 배제한다.

- **[예제 12-10] 그래프 착색하기**
  - 다음 그래프에 대해 그래프 착색을 해보자.



  - [풀이]
  - (a)에 대해 그래프 착색을 해보자. 노드 1을 파란색으로 칠한다면 노드 2, 3, 4는 파란색으로 칠할 수 없지만 노드 5는 파란색으로 칠할 수 있다. 그 다음에 노드 2, 3, 4는 빨간색으로 칠할 수 있다. 그러므로 이 그래프는 2-착색이다.
  - 다음으로 (b)에 대해 그래프 착색을 해보자. 노드 1을 파란색으로 칠한다면 노드 5도 파란색으로 칠 할 수 있다. 또한 노드 2를 빨간색으로 칠한다면 노드 4를 빨간색으로 칠할 수 있다. 그리고 노드 3은 노란색으로 칠할 수 있다. 그러므로 이 그래프는 3-착색이다.