내공 있는 프로그래머로 길러주는

컴파일러의 이해

Chapter 10
코드 최적화

목차

학습목표

- 코드 최적화에 대해 이해할 수 있다.
- 기본 블록과 제어 흐름 그래프, 흐름 분석에 대해 이해할 수 있다.
- 핍홀 최적화 기법, 지역 최적화 기법, 루프 최적화 기법, 전역 최적화 기법, 기계 종속적 최적화 기법 등 최적화 기법에 대해 이해할 수 있다.

10.1 코드 최적화

- **코드 최적화**
  - 주어진 코드에 대해 동등한 의미를 가지면서 실행 시간을 줄이거나 메모리를 줄이는 것
  - 최적화는 '효과가 가장 좋다'는 뜻인데, 가장 효과가 좋은 코드를 만들기는 상당히 어려운 일이기 때문에 일반적으로 '기존의 방법보다 계산 횟수를 줄이거나 실행 시간을 더 짧게 혹은 기억 용량을 더 적게'라는 의미로 사용한다.
- 코드 최적화는 프로그램 전체에서 이뤄질 수 있으며 이를 [그림 10-1]에 나타냈다.

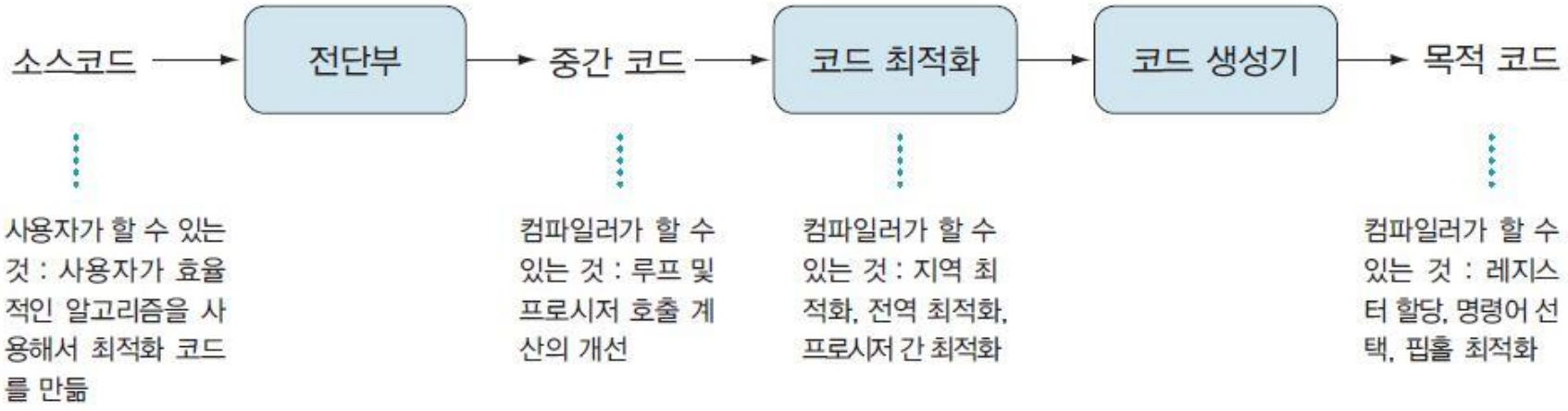


그림 10-1 코드 최적화의 단계별 과정

10.1 코드 최적화

- 코드 최적화를 분류
  - 최적화가 적용되는 프로그램의 영역
    - 지역 최적화(local optimization), 전역 최적화(global optimization, intra-procedural optimization), 프로시저 간 최적화(inter-procedural optimization)
  - 기능적인 측면
    - 실행 시간 최적화와 메모리 최적화
  - 최적화가 많이 이뤄지는 부분
    - 루프(혹은 반복문) 최적화와 단일문 최적화
  - 목적 기계의 의존성
    - 기계 독립적 최적화(machine independent optimization)와 기계 종속적 최적화(machine dependent optimization)
- 통합 환경인 비주얼 스튜디오에서의 C 언어에 대한 컴파일을 생각해보자.
  - 디버깅이나 몇 번 수행하고 버려지는 학생들의 리포트는 빠르지만 최적화 기능이 없는 컴파일러가 더 적합하다.
  - 통합 환경의 에디터에서 선택 메뉴에 RUN과 COMPILE의 차이점은 무엇일까?

10.1 코드 최적화

- 컴파일러 개발자들은 최적화 기법 중에 최소한의 노력으로 최대한의 효과를 낼 수 있는 가장 효율적인 방법을 골라서 컴파일러 최적화에 적용해야 한다.
  - 기계 독립적인 최적화 기법 가운데 프로 그램의 범위에 따라 핍홀 최적화(peephole optimization), 지역 최적화, 전역 최적화, 루프 최적화, 프로시저 간 최적화에 대해 설명하고, 또한 기계 종속적인 최적화도 살펴본다.
  - **핍홀 최적화**는 일반적으로 목적 코드가 생성된 다음 수행된다. 이 기법은 몇 개의 연속적인 명령어를 하나의 명령어나 더 짧은 명령어로 변환하는 것으로 중복(redundant) 명령어 제거, 도달 불 가능한 코드(unreachable code) 제거, 제어 흐름 최적화(flow of control optimization), 대수학적 간소화(algebraic simplification), 세기 감축(strength reduction), 하드웨어 명령어 사용(use of machine idioms) 등이 있다.
  - **지역 최적화**는 부분적인 관점에서 일련의 비효율적인 코드를 구분해내고 좀 더 효율적인 코드로 만드는 방법으로, 코드가 분기해 나가거나 분기해 들어오는 부분이 없는 기본 블록(basic block) 내에서 최적화를 수행하기 때문에 상대적으로 쉽게 수행할 수 있다. 기본 블록은 어떠한 제어 흐름도 가지지 않으므로 분석이 거의 필요 없다. 지역 최적화 기법에는 공통 부분식 제거(common subexpression elimination), 복사 전파(copy propagation), 죽은 코드 제거(dead-code elimination), 상수 폴딩(constant folding), 대수학적 간소화 등이 있다.

10.1 코드 최적화

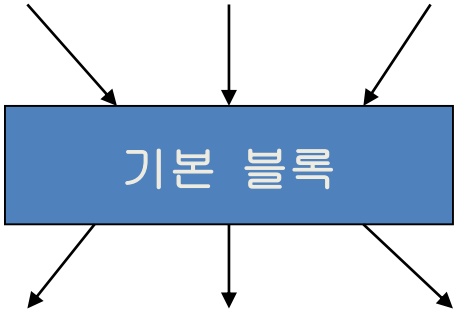
- **루프 최적화**는 한 루프 안에서의 최적화 기법이다. 루프는 전체 코드의 10%가 실행 시간의 90%를 차지하는 부분이므로 루프에 대해 품질 좋은 코드를 생성하는 것은 특히 중요하다. 루프는 보통 많은 횟수로 반복될 수 있으므로 루프 내부(inner loop)의 코드에 대해 계산의 복잡 도를 줄이는 데 더욱 관심을 기울여야 한다. 따라서 많은 코드 변환 기법은 흐름 그래프(flow graph)에서 루프를 찾는 데 많은 시간을 보낸다. 루프 최적화 기법에는 코드 이동(code motion), 세기 감축, 귀납 변수(induction variable) 최적화, 루프 융합(loop fusion), 루프 교환(loop interchange), 루프 전개(loop unrolling) 등이 있다.
- **전역 최적화**는 기본 블록을 넘어서지만 하나의 프로시저 내에서 일련의 비효율적인 코드를 구분해내고 좀 더 효율적인 코드로 만드는 방법이다. 이 기법을 적용하는 데는 지역 최적화보다 더 많은 정보와 비용이 필요하므로 수행하기가 어렵지만 지역 최적화보다 효과가 훨씬 좋다. 전역 최적화에는 일반적으로 자료 흐름 분석(data flow analysis)이라는 기법을 사용하는데, 이는 프로그램 안에서 사용되는 변수의 통로(path)에 관한 정보를 수집해서 처리하는 과정이다. 전역 최적화 기법에는 전역적 공통 부분식 제거, 상수 폴딩, 도달 불가능한 코드 제거 등이 있다.

10.1 코드 최적화

- **프로시저 간 최적화**는 한 프로시저의 한계를 넘어서 전체 프로그램에 적용되는 최적화를 말한다. 이 기법에는 다양한 매개변수 전달 방법, 비지역 변수의 참조, 서로 호출 가능한 모든 프로시저 정보를 동시에 구하는 작업 등이 관련되기 때문에 구현하기가 더욱 어렵다. 이렇게 되면 컴파일러는 컴파일러가 생성한 정보를 이용하여 최적화를 수행하는 링커 없이는 프로시저 간 최적화를 전혀 수행할 수 없다. 이런 이유로 많은 컴파일러에서는 극히 기본적인 프로시저 간 최적화만 수행하거나 프로시저 간 최적화를 전혀 수행하지 않는다.
- **기계 종속적 최적화**는 기계의 특성에 따라 아주 달라질 수 있다. 이 기법에는 중복된 LOAD 명령어 제거, 효율적인 명령어 선택(instruction selection), 레지스터 할당(register allocation)과 레지스터 배정(register assignment)등이 있다.

10.2 기본 블록과 흐름 그래프

- 최적화 기법을 수행하기 위해 여러 가지 방법을 사용
  - 지역 최적화, 전역 최적화, 프로시저 간 최적화, 루프 최적화 기법에서는 최적화를 수행하기 위해 전체 프로그램을 기본 블록 단위로 분할하여 최적화
  - 기본 블록 안에서 최적화를 수행할 수 있는데 이것을 지역 최적화
  - 기본 블록을 노드로 가진 흐름 그래프를 만들고 흐름 그래프에서 자료의 흐름을 분석한다. 전역 최적화, 프로시저 간 최적화, 루프 최적화 등은 흐름 그래프와 자료 흐름 분석 등을 이용한다.
- **기본 블록**
  - 지역 최적화의 기본 단위
  - 제어가 시작점으로 들어와서 끝점으로 나갈 때까지 정지(halt)나 분기의 가능성이 없는 연속적인 코드들의 집합



10.2 기본 블록과 흐름 그래프

[예제 10-1] 기본 블록인지 확인

- 다음 3-주소 코드가 하나의 기본 블록인지 살펴보자.
- t1 = a + b
- t2 = a * b
- t3 = t1 * 5
- t4 = t1 + t2
- t5 = t3 + t4
- t6 = t2 * t5
- [풀이] 6개의 3-주소 코드는 시작점인 t1 = a + b에서 시작하여 끝점인 t6 = t2 * t5까지 제어가 정지되거나 분기되지 않는다. 그러므로 하나의 기본 블록이다

10.2 기본 블록과 흐름 그래프

- 기본 블록은 리더와 다음 리더 이전에 나타나는 모든 코드로 구성되기 때문에 먼저 리더를 찾아야 한다.
- **[정의 10-1] 리더**
  - (1) 프로그램의 시작 문장
  - (2) 조건부 분기 또는 무조건 분기의 목적지에 있는 문장
  - (3) 조건부 분기 바로 다음에 위치하는 문장

- 다음 예는 n 값을 읽어 들여 1부터 n까지의 곱을 구하는 C 프로그램이다. 일반적으로 재귀적인 프로그램을 작성할 때 가장 많이 사용하지만, 여기서는 while 반복법을 이용하여 프로그램을 작성하면 [그림 10-2]와 같다.

10.2 기본 블록과 흐름 그래프


```c
#include <stdio.h>
int main(void)
{
  int mult = 1, i=1, n=0;
  printf("1부터 n까지의 곱을 구할 n을 입력하세요.");
  scanf("%d", &n);
  while (i <= n)
  {
    mult *=i;
    i++;
  }
  printf("1부터 %d까지의 곱은 %d입니다. \n", n, mult);
  return 0;
}
```

그림 10-2 1부터 n까지의 곱을 구하는 프로그램

10.2 기본 블록과 흐름 그래프


```
① mult = 1
② i = 0
③ n = 0
④ scanf(&n)
⑤ if n > i goto ⑫
⑥ t1 = mult * i
⑦ mult = t1
⑧ t2 = i + 1
⑨ i = t2
⑩ if i <= n goto ⑥
⑪ printf(mult)
⑫ halt
```

그림 10-3 [그림 10-2]에 대한 3-주소 코드

10.2 기본 블록과 흐름 그래프

[예제 10-2] 리더 찾기

- [그림 10-3]의 3-주소 코드에서 리더를 찾아보자.
- [풀이] ①번은 시작 문장이므로 리더이다.
- ⑫번은 조건 분기 문장인 ⑤번 문장의 목적지에 있는 문장이므로 리더이다.
- ⑥번 문장은 ⑤번 문장이 조건 분기 문장이므로 리더이다.
- ⑥번 문장은 조건 분기 문장인 ⑩번 문장의 목적지에 있는 문장이므로 리더이다.
- ⑪번 문장은 ⑩번 문장이 조건 분기 문장이므로 리더이다.
- 따라서 리더는 ①, ⑥. ⑪, ⑫번 문장이다.

[알고리즘 10-1] 3- 주소 코드로부터 기본 블록을 구하는 방법

- [입력] 3-주소 코드
- [출력] 기본 블록의 리스트
- [방법] 리더와 다음 리더 사이에 있는 모든 코드이다.

[예제 10-3] 기본 블록 구하기

- [그림 10-3]에 대해 [알고리즘 10-1]을 적용하여 기본 블록을 구해보자.
- [풀이]
- [예제 10-2]에서 모든 리더를 찾았다.
- [그림 10-3]에서 첫 번째 리더는 ①번 문장이고 두 번째 리더는 ⑥번 문장이므로 ①~⑤번 문장이 기본 블록인데 이를 B1이라고 하자.
- 마찬가지로 두 번째 리더는 ⑥번 문장이고 그 다음 리더는 ⑪번 문장이므로 ⑥~⑩번 문장이 기본 블록인데 이를 B2라고 하자.
- 또한 ⑪번 문장이 기본 블록이므로 B3이라 하고,
- ⑫번 문장이 기본 블록이므로 B4라고 한다.
- 따라서 [그림 10-3]의 코드는 4개의 기본 블록으로 구성되어 있으며, 이는 [그림 10-4]와 같이 나 타낼 수 있다.

10.2 기본 블록과 흐름 그래프

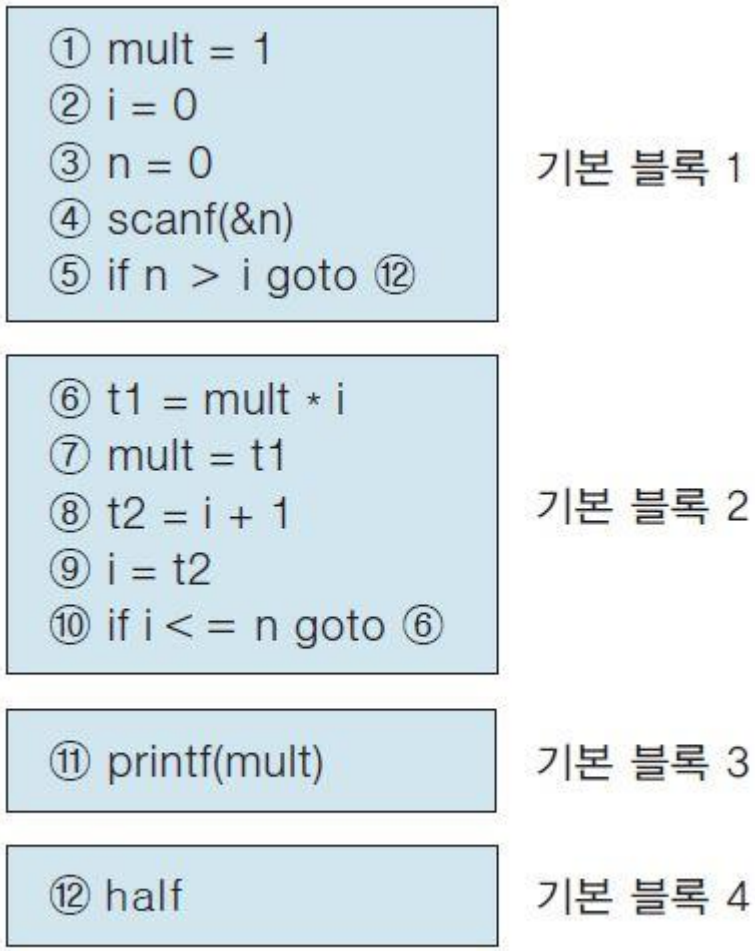


그림 10-4 [그림 10-3]에 대한 기본 블록

제어 흐름 그래프와 DAG

- 최적화에 이용하기 위해 기본 블록의 집합에 제어 흐름에 관한 정보를 추가해서 만든 유향 그래프(direct graph)를 **흐름 그래프**
- 흐름 그래프는 노드는 기본 블록이고 간선은 조건 분기와 무조건 분기를 나타낸다.
- 흐름 그래프는 모든 제어 흐름의 정보를 포함한다. 수집된 정보는 전역 최적화를 수행하는데 매우 중요하다.
- 흐름 그래프에서 기본 블록 B에서 기본 블록 C로의 간선을 추가하는 방법
  - B의 마지막 문장에서 C의 첫 번째 문장으로 조건 분기 혹은 무조건 분기가 존재한다.
  - 프로그램의 실행 순서상 B 바로 다음에 C가 나타나며, 이때 B의 마지막 문장이 무조건 분기로 끝나지 않는다.
- 이와 같은 경우에 B는 C의 선행자(predecessor)라 하고, C는 B의 계승자(successor)라 한다. 마지막으로 진입 노드(entry node)와 진출 노드(exit node)라 불리는 2개의 노드를 추가한다.

10.2 기본 블록과 흐름 그래프

[예제 10-4] 흐름 그래프 그리기

- [그림 10-4]에서 기본 블록에 대한 흐름 그래프를
- 구해보자.
- [풀이]
- [그림 10-4]에 대한 흐름 그래프는 다음과 같다
-

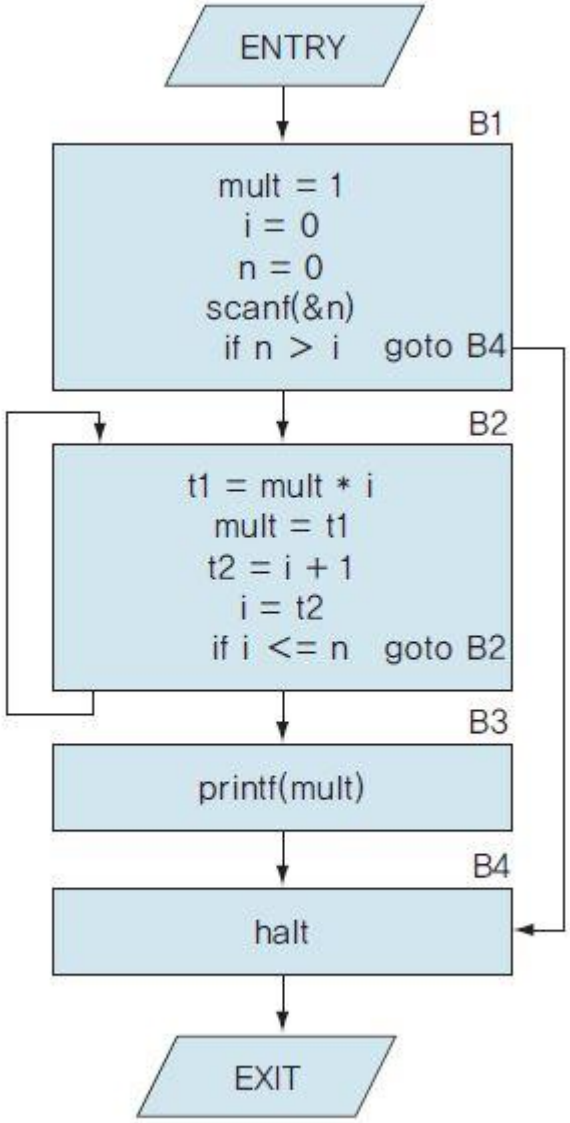


그림 10-5 [그림 10-4]에 대한 흐름 그래프

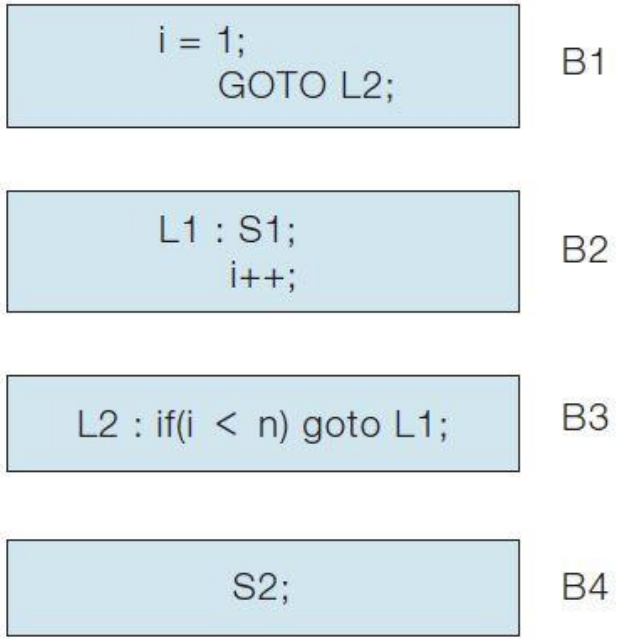
[예제 10-5] 3-주소 코드, 리더, 기본 블록을 구하고 흐름 그래프 그리기

- 다음과 같은 아주 간단한 반복문에 대해 3-주소 코드, 리더, 기본 블록을 구하고 흐름 그래프를 그려보자.
- For(i = 1; i < n; i++)
- S1;
- S2;
- [풀이] 3-주소 코드로 변환하면 다음과 같다.
- i = 1;
- GOTO L2;
- L1 : S1;
- i++;
- L2 : if(i < n) goto L1;
- S2;
-

10.2 기본 블록과 흐름 그래프

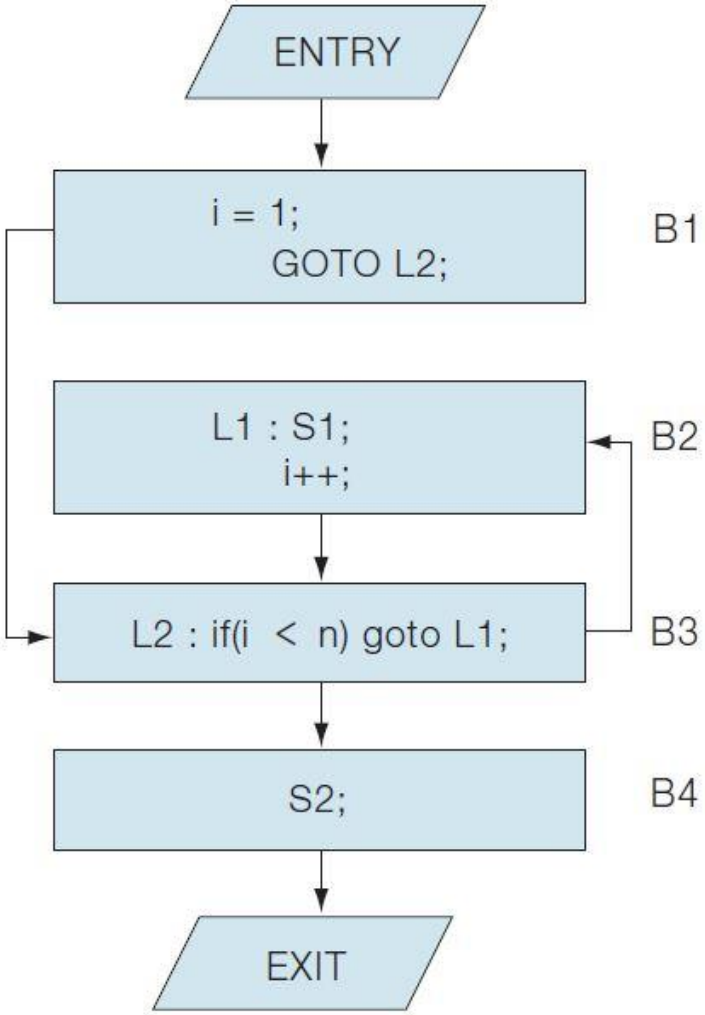
- 여기서 리더를 찾는다.
  - i = 1; 리더(프로그램의 시작)
  - GOTO L2;
  - L1 : S1; 리더(분기의 목적지, 분기의 다음 문장)
  - i++;
  - L2 : if(i < n) goto L1; 리더(분기의 목적지)
  - S2; 리더(분기의 다음 문장)

- 이제 기본 블록을 구하면 다음과 같다.



10.2 기본 블록과 흐름 그래프

■ 마지막으로 흐름 그래프를 그리면 다음과 같다.

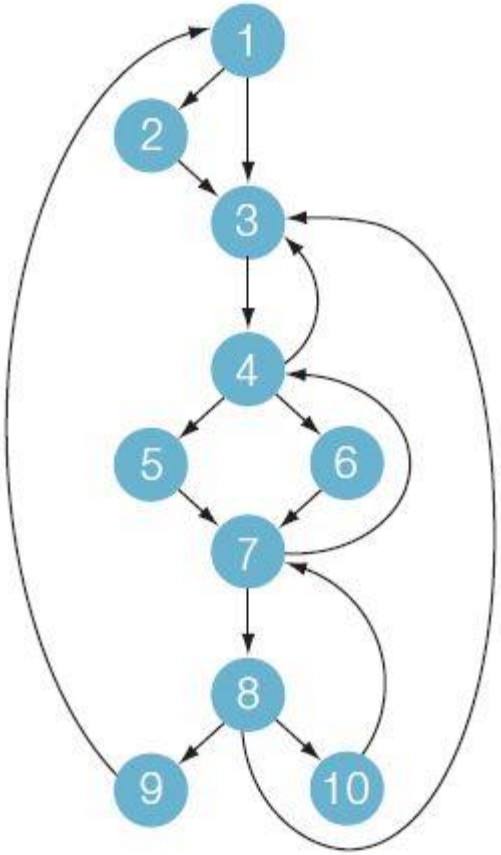


10.2 기본 블록과 흐름 그래프

- 흐름 그래프에서 루프는 무엇인가? 그리고 모든 루프는 어떻게 찾을 수 있는가? 대부분의 경우 이런 질문에 쉽게 답할 수 있다. 예를 들면 [그림 10-5]에서 루프는 B2 블록이다.
- **[정의 10-2] 루프**
- 다음 조건을 만족하면 루프(loop)라고 한다.
  - 루프는 입구(entry)가 단 하나이다. 루프 밖에서 루프 안으로 도달하기 위해 항상 먼저 지나는 노드가 입구이며, 루프는 이 입구가 하나이다.
  - 루프 안의 모든 노드는 강하게 연결되어 있다

- 다른 루프를 포함하지 않는 루프를 **내부 루프**
- 흐름 그래프의 시작 노드(initial node)에서 어떤 노드 n에 이르는 모든 통로가 노드 d를 통과하는 경우, 노드 d는 노드 n을 지배하고(dominates) 이를 d dom n으로 표현한다. 즉 모든 노드는 자기 자신을 지배한다. 그리고 앞에서 설명한 한 루프의 진입 노드는 그 루프의 모든 노드를 지배한다.

- **[예제 10-6] 지배자 찾기**
- 다음과 같은 흐름 그래프에서 모든 노드에 대한 지배자를 찾아보자.

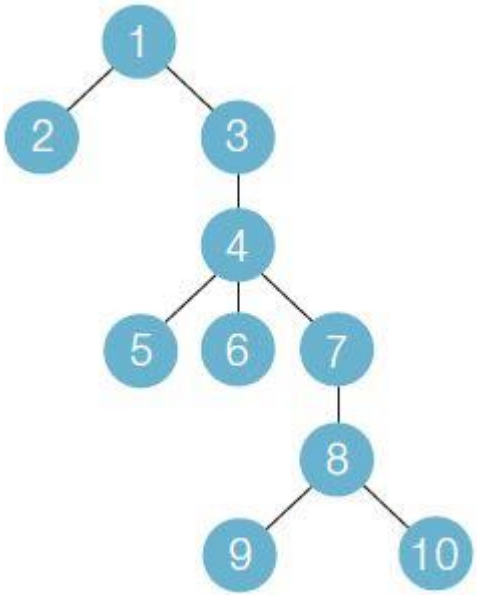


10.2 기본 블록과 흐름 그래프

- [풀이]
- 시작 노드는 1이고 이 노드는 모든 노드를 지배한다.
- 노드 2는 자기 자신만을 지배한다. 왜냐하면 다른 노드들은 1로부터 3을 거치는 통로를 통해 모두 도달할 수 있기 때문이다.
- 노드 3은 노드 1과 2를 제외한 모든 노드를 지배하고
- 노드 4는 노드 1, 2, 3을 제외한 모든 노드를 지배한다. 왜냐하면 노드 1로부터 모든 통로는 1 → 2 → 3 → 4나 1 → 3 → 4로 시작해야 하기 때문이다.
- 노드 5와 6은 각각 자기 자신만을 지배하고
- 노드 7은 노드 7, 8, 9, 10을 지배한다.
- 노드 8은 노드 8, 9, 10을 지배하고
- 노드 9와 10은 자기 자신만을 지배한다

10.2 기본 블록과 흐름 그래프

- 트리를 이용하면 지배자 정보를 쉽게 나타낼 수 있는데 이런 트리를 지배자 트리(dominator tree)라고 한다. 시작 노드는 루트이고 각 노드는 그 자손을 지배한다. 흐름 그래프에 대한 지배자 트리는 다음과 같다.



10.2 기본 블록과 흐름 그래프

- 자료 흐름 분석이 완료된 후 기본 블록에 대한 코드를 생성할 때, 각 블록에 대해

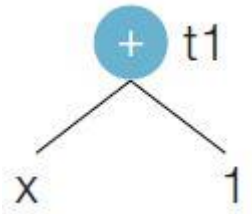
  DAG(directed acyclic graph; 비순환 유향 그래프)라는 자료 구조가 생성

  - DAG 는 기본 블록 내의 한 문장이 뒤의 문장에 사용되므로 값이 어떻게 계산되는지를 그림으로 나타내준다.
  - DAG는 주로 블록 내부에서 사용되지만, 블록 외부에서 어떤 문 장이 그 문장에서 계산한 값을 사용하는지 알아볼 수 있으므로 매우 중요하다.
  - DAG는 모든 산술식에 대한 노드를 가진다. 즉 내부 노드(interior node)는 연산자를 나타내고 그 자식 노드는 피연산자를 나타낸다.
  - 터미널 노드는 식별자나 상수이고 중간 노드는 연산자이다. 때때로 노드 옆에 여러 개의 식별자를 놓을 수 있다. 이것은 그 노드에서 계산을 한 후 결과 값을 이 식별자들이 공유함을 나타낸다.

10.2 기본 블록과 흐름 그래프

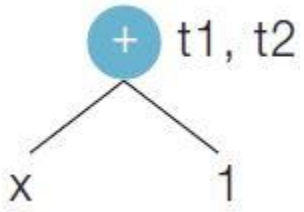
- **[알고리즘 10-2] DAG 구성 방법**
  - [입력] 기본 블록
  - [출력] 터미널 노드에 대한 레이블은 식별자 혹은 상수이고 내부 노드는 연산자가 되는 기본 블록에 대한 DAG
  - [방법] 생성되는 노드는 1개나 2개의 자식 노드를 가지게 된다. 1개의 자식 노드를 가질 경우 '왼쪽'과 '오른쪽' 자식 노드로 구분된다. DAG 구성은 다음 ❶부터 ❸ 단계를 블록의 각 문장에 대해 차례로 수행하면 된다. 현재의 3-주소 코드는 x = y op z, x = op y, x = y 등 세 가지 중 하나라고 가정한다. if i < = 20 goto와 같은 경우에 는 x = y op z에서 x가 정의되지 않은 것으로 간주한다.
  - ❶ x = y op z의 경우에는 만약 node(y)와 node(z)가 정의되어 있지 않다면 터미널 노드 y와 z를 생성한다.
  - ❷ x = y op z의 경우에는 왼쪽 자식이 node(y)이고 오른쪽 자식이 node(z)이면서 똑같은 노드 op가 존재하는 지 알아보고, 만약 존재하지 않으면 새 노드 node(op)를 생성한다. 이때 부모 노드는 이미 만들어진 노드를 찾거나 아니면 새롭게 생성되는 노드가 된다. x = op y의 경우에는 node(y) 하나만을 자식으로 가지고 있는 노드 op가 존재하는지 알아보고, 만약 존재하지 않으면 새 노드를 생성한다. 이때 부모 노드는 이미 만들어진 노드를 찾거나 아니면 새롭게 생성되는 노드가 된다. x = y의 경우에는 새로운 노드를 생성하지 않는다.
  - ❸ ❷ 단계에서 찾은 부모 노드에 있는 식별자 리스트에 x를 첨가한다.

- **[예제 10-7] DAG 그리기 1**
  - C의 치환문 x = (x + 1) * (x + 1)에 대해 3-주소 코드로 변환하고 [알고리즘 10-2]를 적용 하여 DAG를 그려보자.
  - [풀이]
  - 3-주소 코드는 다음과 같다.
  - t1 = x + 1
  - t2 = x + 1
  - t3 = t1 * t2
  - x = t3
  - 이 코드에 대한 DAG를 만들기 위해 [알고리즘 10-2]를 적용하면 첫 번째 문장은 t1 = x + 1이다. ❶ 단계에서 x와 1에 대한 터미널 노드를 생성하고, ❷ 단계에서 + 노드를 생성한다. 왼쪽 자식 노드는 x이고 오른쪽 자식 노드는 1이다. ❸ 단계에서 식별자 t1이 이 노드에 딸리도록 식별자 리스트에 첨가한다.

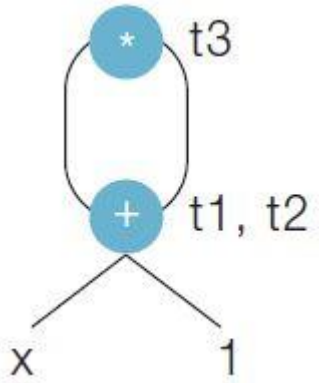


10.2 기본 블록과 흐름 그래프

- 두 번째 문장은 t2 = x + 1이다. 여기서 node(x)와 node(1)을 찾아보면 이미 존재하므로 새로운 노드를 생성하지 않고 t2를 노드 t1의 식별자 리스트에 추가한다

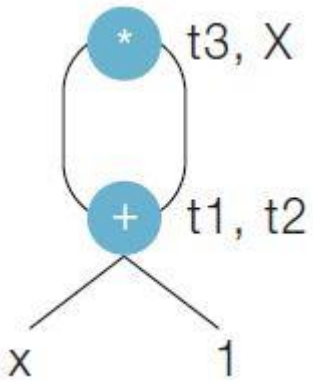


- 다음 문장은 t3 = t1 * t2이다. node(t1)과 node(t2)를 찾아보면 이미 존재하므로 새로운 노드를 생성하지 않는다. ❷ 단계에서 t1과 t2를 노드로 하는 * 노드를 생성한다. ❸ 단계에서 식별자 t3이 이 노드에 딸리도록 식별자 리스트에 첨가한다



10.2 기본 블록과 흐름 그래프

- 마지막 문장은 x = t3이다. 치환문이므로 새로운 노드를 생성하지 않고 x를 노드 t3의 리스트에 추가한다. 이렇게 하면 다음과 같은 DAG가 된다



흐름 분석

- 좋은 코드를 생성하기 위해 코드 최적화기(code optimizer)는 프로그램의 전체적인 정보를 모으고, 이 정보를 흐름 그래프의 각 블록에 제공해야 한다. 즉 최적화에 이용하기 위해 제어 흐름에 관한 정보를 기본 블록의 집합에 모아야 한다.
- 자료 흐름 분석은 프로그램이 실행되었을 때 어떤 일이 발생할지 추론하여 발견하는 것이다. 즉 프로그램 안에서 사용되는 변수의 통로에 관한 정보를 모으는 처리 과정으로 유효 변수(live variables), 공통 부분식 등에 대한 정보를 모으는 일을 한다. 최적화 컴파일러에서 자료 흐름 분 석의 가장 중요한 용도는 최적화의 기회가 제공되는 위치를 파악하고, 어떤 지점에서 코드 변형을 적용하는 것이 안전한지를 증명하는 것이다. 이러한 일들을 **정적 분석법(static analysis)**이라 한다.
- 변수 값이 다음에 언제 사용될지를 아는 것은 품질 좋은 코드를 생성하는 데 필수적이다. 3- 주소 코드 x = y * z에서 x를 **정의**(define), y와 z를 **사용**(use)이라고 부른다. 변수 x, y, z가 다음에 어디서 사용될지를 아는 것은 매우 중요한 정보이다. 이와 같이 어떤 변수가 다음에 어디서 사용될 것인가를 **다음 사용**(next-use) 정보라 한다.
- 그럼 다음에 사용될 y의 값은 프로그램 어디서 정의된 값인가? 이 질문의 답을 구하려면 'UD 체인(use-definition chaining)' 계산을 수행해야 한다. 이 분석 방법은 피연산자로 각 단순 변수가 나타날 때마다 해당하는 UD 체인을 제공한다. UD 체인은 프로그램의 어느 장소에 변수의 정의가 올바르게 나타나는지를 알려준다.

핍홀 최적화 기법

- 핍홀 최적화는 일반적으로 목적 코드가 생성된 다음 수행된다. 이 기법은 몇 개의 연속적인 명령어를 하나의 명령어나 더 짧은 명령어로 변환하는 것으로, 연속적인 몇 개 명령어의 집합을 '핍홀(peephole)' 또는 '창(window)'이라고 부른다. 핍홀 최적화 기법에는 중복 명령어 제거, 도달 불가능한 코드 제거, 제어 흐름 최적화, 대수학적 간소화, 세기 감축, 하드웨어 명령어 사용 등이 있다.
- **중복 명령어 제거**
  - 중복되는 로드(LOAD) 명령문이나 저장(STORE) 명령문을 제거하는 방법이다. 다음 명령어를 살펴보자.
  - LOAD R1, x
  - STORE x, R1
  - 여기서 저장 명령어를 삭제할 수 있다. 왜냐하면 이 명령어가 실행될 때마다 x의 값이 이미 레지스터 R1에 적재되어 있기 때문이다.

[예제 10-9] 중복된 명령어 제거

- 다음 소스코드에 대해 목적 코드로 변환하고 중복된 명령어가 있는지 찾아보자.
- x = y;
- z = x + 10;
- [풀이] 목적 코드는 다음과 같다.
- LOAD R1, y
- STORE x, R1
- LOAD R1, x
- ADD R1, 10
- STORE z, R1
- 여기서 두 번째 LOAD 명령문은 레지스터 R1에 이미 x의 값이 있으므로 불필요하다

도달 불가능한 코드 제거

- 불필요한 코드(useless code)에는 도달 불가능한 코드와 죽은 코드
  - 죽은 코드는 지역 최적화 기법에서 설명
  - 도달 불가능한 코드는 무조건 분기 바로 다음 문장에 있는 라벨이 없는 명령어 같은 것으로, 이는 도달할 수 없기 때문에 제거할 수 있다.
- **[예제 10-10] 도달 불가능한 코드 제거**
- 다음 3-주소 코드에 있는 도달 불가능한 코드를 제거해보자.

```
        if A = B then goto L3
        A = 2
        goto L5
   L3 : if A = B then goto L5
        A = 3
   L5 :
```

- 목적 코드에서 A = 3은 도달 불가능한 코드이므로 제거하면 다음과 같다.

```
        if A = B then goto L3
        A = 2
        goto L5
   L3 : if A = B then goto L5
```

10.3 최적화 기법

제어 흐름 최적화

- 중간 코드 생성은 분기에 대한 분기, 조건 분기에 대한 분기, 분기에 대한 조건 분기를 생성하는데 이러한 불필요한 분기들을 제거할 수 있다. 다음 코드를 살펴보자.

```
        goto L5
          ⋮
    L5 : goto L1
```

- 이 코드는 다음과 같이 변환할 수 있으며, 이렇게 함으로써 실행 시간을 줄일 수 있다.

```
        goto L1
          ⋮
    L5 : goto L1
```

10.3 최적화 기법

대수학적 간소화

- 수학적인 대수 법칙을 이용하여 식을 간소화하는 방법이다. 다음과 같은 대수 법칙은 코드 최적화에 유익한 기능을 제공한다.
- x = y + 0 ⇒ x = y (덧셈의 항등 법칙)
- x = 0 + y ⇒ x = y (덧셈의 항등 법칙)
- x = y – 0 ⇒ x = y (뺄셈의 항등 법칙)
- x = y * 1 ⇒ x = y (곱셈의 항등 법칙)
- x + y ⇒ y + x (덧셈의 교환 법칙)

- 교환 법칙을 만족하는 연산자를 필요할 때마다 적용하여 식을 간소화할 수 있다.
  - 예를 들어 a = 3 * b / 3은 *가 교환 법칙을 만족하여 a = b * 3 / 3이므로 a = b * 1이 되고 결국 a = b가 된다.

세기 감축

- 수행 비용이 높은 연산자를 수행 비용이 낮은 연산자로 대체하는 것
- 거듭 제곱 연산자는 곱셈 연산자로, 곱셈 연산자는 덧셈 연산자로, 나눗셈 연산자는 곱셈 연산자로 바꾼다.

[예제 10-11] 세기 감축하기

- 다음 문장에 대해 각각 세기 감축을 해보자.
- y = x ↑ 2 (↑는 거듭제곱 연산자)
- y = x * 3
- y = x / 5
- [풀이] 다음과 같이 치환한다.
- y = x ↑ 2 ⇒ y = x * x
- y = x * 2 ⇒ y = x + x
- y = x / 5 ⇒ y = x * 0.2

하드웨어 명령어 사용

- 어떤 특정 연산을 효율적으로 구현하기 위한 하드웨어 명령어 포함
  - 어떤 기계는 자동 증가(auto-increment)와 자동 감소(auto-decrement) 모드는 피연산자로부터 값을 사용하기 전이나 후에 피연산자에 1을 증가시키거나 감소시킬 때 사용하는데, 하드웨어로 구현하기 때문에 매우 큰 비용을 절감
  - 매개변수 전달에서 자료를 스택에 삽입하거나 삭제할 때도 하드웨어 명령어를 사용하여 품질을 향상한다. 이러한 모드는 i = i + 1과 같은 문장에서도 사용할 수 있다.

10.3 최적화 기법

지역 최적화 기법

- 지역 최적화는 부분적인 관점에서 비효율적인 코드를 구분해내고 좀 더 효율적인 코드로 개선하는 방법
  - 기본적으로 기본 블록 안에서 최적화가 이뤄짐
  - 공통 부분식 제거, 복사 전파, 죽은 코드 제거, 상수 폴딩, 대수학적 간소화 등이 있다.

공통 부분식 제거

- 공통 부분식은 프로그램에서 공통된 부분이 여러 번 나타나는 경우
  - 공통 부분식이 한 번만 계산되도록 함으로써 코드를 효율적으로 만들 수 있는데 이러한 방법을 공통 부분식 제거

[예제 10-12] 공통 부분식 제거 1

- 다음 치환문에서 공통 부분식을 제거해보자.
- A = B + C + D;
- E = B + C + F;
- K = (B + C) * G;
- [풀이]
- B + C는 3개의 치환문에 모두 공통이므로 다음과 같이 효율적으로 변경할 수 있다.
- T1 = B + C;
- A = T1 + D;
- E = T1 + F;
- K = T1 * G;

복사 전파

- f = g 형태를 치환문이라 하는데, 복사 전파란 f = g가 나타난 이후에 가능하면 f 대신에 g를 사용하는 것이다. 즉 치환문을 삭제하고 삭제된 치환문의 l-값 대신에 r-값을 사용하는 방법
- **[예제 10-14] 복사 전파하기**
- 다음 치환문에서 복사 전파를 해보자.
  - x = t3;
  - a[t2] = t5;
  - a[t4] = x;
  - goto B2;
  - [풀이]
  - x = t3;
  - a[t2] = t5;
  - a[t4] = t3;
  - goto B2;

죽은 코드 제거

- 어떤 변수가 특정 프로그램 지점 이후 전혀 사용하지 않는 값을 계산하는 문장을 말한다
- **[예제 10-15] 죽은 코드 제거하기**
- [예제 10-14]에서 복사 전파를 수행한 후의 코드는 다음과 같다. 이 코드에서 죽은 코드를 제거해보자.
  - x = t3;
  - a[t2] = t5;
  - a[t4] = t3;
  - goto B2;
  - [풀이]
  - x = t3;이라는 치환문은 더 이상 사용하지 않아 죽은 코드이다.
  - a[t2] = t5;
  - a[t4] = t3;
  - goto B2;

10.3 최적화 기법

■ 상수 폴딩

- 컴파일을 할 때 상수를 포함하는 연산이 계산될 수 있으면 계산을 미리 함으로써 코드를 줄이는 방법
- **[예제 10-16] 상수 폴딩하기**
- 다음 치환문에서 상수 폴딩을 해보자.
  - X = 3.14;
  - Y = 2 * X;
  - [풀이]
  - 이 문장에서 Y를 계산할 때까지 X의 값이 변하지 않는다면 컴파일 시간에 Y = 6.28;로 변환하는 것이다.

■ 대수학적 간소화

- 핍홀 최적화 기법에서 다루었다.

10.3 최적화 기법

루프 최적화 기법

- 전체 코드의 10%가 실행 시간의 90%를 차지하는 부분이 루프이므로 최적화에서 매우 중요한 부분
- while 문, do-while 문, for 문 등과 같이 프로그래밍 언어 구조는 루프를 생성
- 코드 이동, 세기 감축, 귀납 변수 최적화, 루프 융합, 루프 교환, 루프 전개
-

코드 이동

- 어떤 식의 값이 루프 수행 횟수와 상관없이 항상 같은 값이라면 루프 수행 전에 이 식을 계산할 수 있도록 이 식의 위치를 루프 내부로 들어가기 전의 루프 외부로 이동하는 것
- 루프 수행 횟수와 상관없이 항상 같은 값을 가진 루프 불변(loop invariant)이 있는 경우, 루프 불변을 그 루프의 바로 전 위치로 이동함으로써 계산 횟수를 대폭 줄일 수 있다.

- **[예제 10-17] 코드 이동하기**
- 다음 반복문에서 코드 이동을 해보자.
- For(k = 1; k < = 1000; k++)
- c[k] = 2 * (p - q) * (n - k + 1) / (sqrt(n) + n);
  - [풀이]
  - 이 문장에서 제어 변수 k는 1부터 1000까지 수행하지만 2 * (p - q)와 (sqrt(n) + n) 값이 변하지 않는 루프 불변이다. 그러므로 이 2개의 수식을 다음과 같이 루프 밖으로 보내면 된다.
  - fact = 2 * (p - q);
  - denom = sqrt(n) + n;
  - For(k = 1; k < = 1000; k++)
  - c[k] = fact * (n - k + 1) / denom;

세기 감축

- 핍홀 최적화 기법에서 설명했다.

10.3 최적화 기법

귀납 변수 최적화

- 루프를 돌 때마다 값이 일률적으로 변하는 변수
- 세기 감축으로 최적화할 수 있다.
- 이러한 변수로는 반복문 제어 변수 및 반복문 제어 변수에 특정 형태로 의존하는 다른 변수들
- 선정된 귀납 변수는 레지스터에 할당되어 보다 간단히 계산할 수 있다.
- 코드를 변경하는 데는 코드 이동도 포함될 수 있다.

10.3 최적화 기법

- **[예제 10-18] 귀납 변수 최적화하기**
- 3-주소 코드의 기본 블록에 대해 귀납 변수 최적화를 해보자.
- L1 : j = j - 1
- t4 = 4 * j
- t5 = a[t4]
- if t5 > v goto L1

  - [풀이]
  - 우선 귀납 변수를 찾는다. 루프가 한 번 수행될 때마다 j는 1씩 감소하고 t4는 4씩 감소한다.
  - L1 : j = j - 1
  - t4 = t4 - 4
  - t5 = a[t4]
  - if t5 > v goto L1

루프 융합

- 루프의 오버헤드를 줄이기 위한 방법으로, 루프의 범위가 같아서 여러 개의 루프를 하나의 루프로 만드는 것이다. 루프 제어의 명령이 많이 줄어든다.

- **[예제 10-19] 루프 융합하기**
- 다음 문장에 대해 루프 융합을 해보자.

```
for (j = 1; j < = 100; j++)
    a[j] = b[j] + c[j];
for (j = 1; j < = 100; j++)
    b[j] = 5 + c[j];
```

- [풀이]
- 2개의 루프로 구성되어 있는 프로그램을 하나로 묶으면 루프 제어의 명령을 줄일 수 있다.

```
for (j = 1; j < = 100; j++)
{
  a[j] = b[j] + c[j];
  b[j] = 5 + c[j];
 }
```

10.3 최적화 기법

루프 교환

- 내부 루프와 외부 루프를 교환하는 것
- 참조 지역성(locality of reference) 을 개선

10.3 최적화 기법

- **[예제 10-20] 루프 교환하기**
- 다음 문장에 대해 루프 교환을 해보자.

```
for (j = 1; j  < = 100; j++)
{
  for (k = 1; k  < = 200; k++)
  {
    a [j,k] = j + k;
  }
}
```

[풀이]

```
for (k = 1; k  < = 200; k++)
{
  for (j = 1; j  < = 100; j++)
  {
    a[j,k] = j + k;
   }
}
```

10.3 최적화 기법

루프 전개

- 루프의 계산을 빠르게 하기 위해 루프를 펼치는 효과를 내는 방법
- 증가와 조건 검사의 횟수가 감소하기 때문에 수행되는 명령어의 수가 줄어듦
- 제어 변수의 최종 값을 증가시킬 수도 있고, 제어 변수의 값을 바꿔서 구현할 수도 있다.
- store라든가 조건부 분기 명령(conditional jump)은 선행 제어를 어렵게 하므로 이런 경우 효과가 반감될 수 있다. 루프 전개를 위해서는 반복 횟수를 컴파일 시간에 알아야 한다

10.3 최적화 기법

- **[예제 10-21] 루프 전개하기 1**
- 다음 문장에 대해 루프 전개 해보자.

```
int x;
for (x = 0; x < 100; x++)
{
 delete(x);
 }
```

[풀이]

```
int x;
for (x = 0; x < 100; x += 5)
{
 delete(x);
 delete(x + 1);
 delete(x + 2);
 delete(x + 3);
 delete(x + 4);
 }
```

전역 최적화 기법

- 프로그램의 전체적인 흐름 분석을 통해 일련의 비효율적인 코드를 좀 더 효율적인 코드로 만드는 방법
- 지역 최적화와는 달리 기본 블록 간 정보와 흐름 그래프를 이용하고 프로그램의 전체적인 흐름 분석을 통한 최적화 기법
- 전역적 공통 부분식 제거, 상수 폴딩, 도달 불가능한 코드 제거

전역적 공통 부분식 제거

- 지역 최적화에서도 사용하는 방법으로, 여기서는 기본 블록 간에 나타나는 공통 부분식에 대해서도 단 한 번만 계산하도록 함으로써 코드를 개선하는 것
- [그림 10-6(a)]와 같은 코드에서 공통 부분식을 제거하면 [그림 10-6(b)]와 같이 된다. 이것도 좀 더 최적화하면 세기 감축이 가능하다. x3 = a / 5 + t를 x3 = a * 0.2 + t로 최적화할 수 있다.

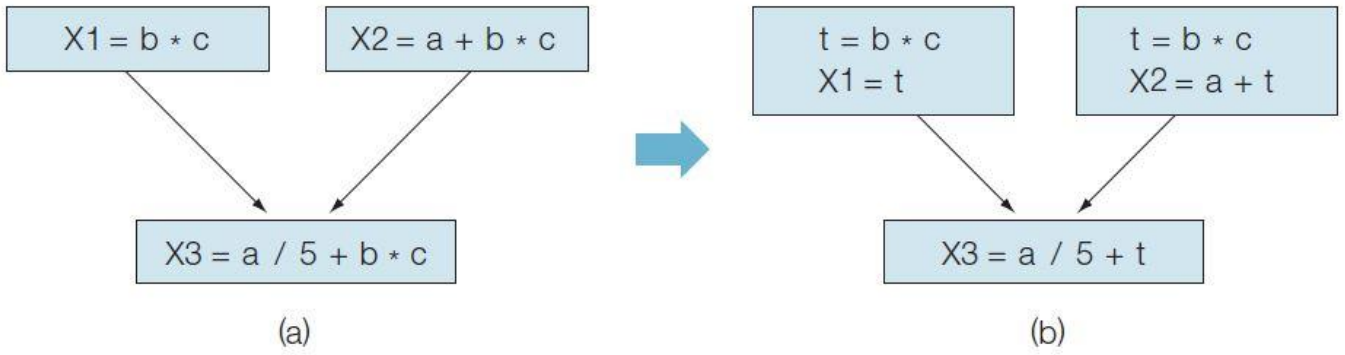


그림 10-6 전역적 공통 부분식 제거

상수 폴딩

- 상수 폴딩도 지역 최적화 기법에서 설명했다.
- 하나의 기본 블록 내에 어떤 변수에 대한 정의가 존재하지 않는다면 이 블록으로 도달할 수 있는 이전 블록 어딘가에 그 변수에 대한 정의가 있을 것이다. 이러한 특성을 이용하여 기본 블록 간에도 상수 폴딩을 할 수 있다.
- [그림 10-7(a)]에 대해 상수 폴딩을 하면 [그림 10-7(b)]와 같이 된다.

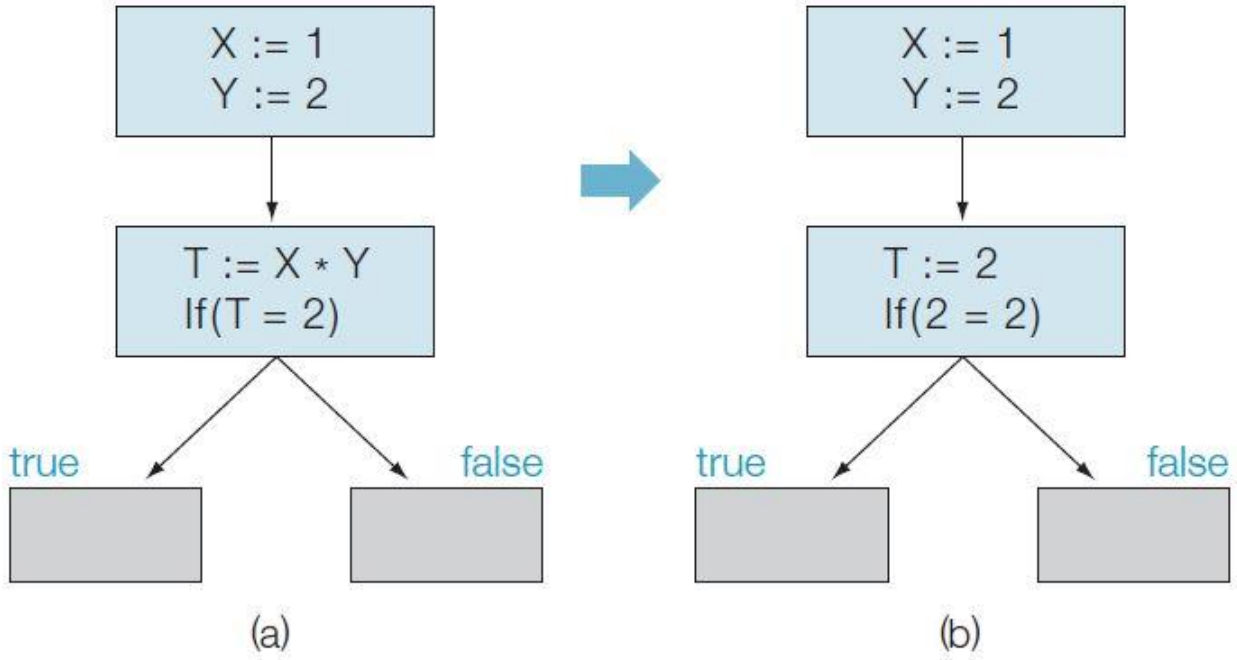


그림 10-7 상수 폴딩

도달 불가능한 코드 제거

- 도달 불가능한 코드 제거도 지역 최적화 기법에서 설명했다.
- 흐름 그래프의 입구에서부터 도달할 수 없는 기본 블록은 제거가 가능한데, 상수 폴딩에서 사용된 예의 경우에는 제어문이 항상 true이므로 false로 가는 것을 제거할 수 있다.
- [그림 10-8(a)]에서 도달 불가능한 코드를 제거하면 [그림 10-8(b)]와 같다.

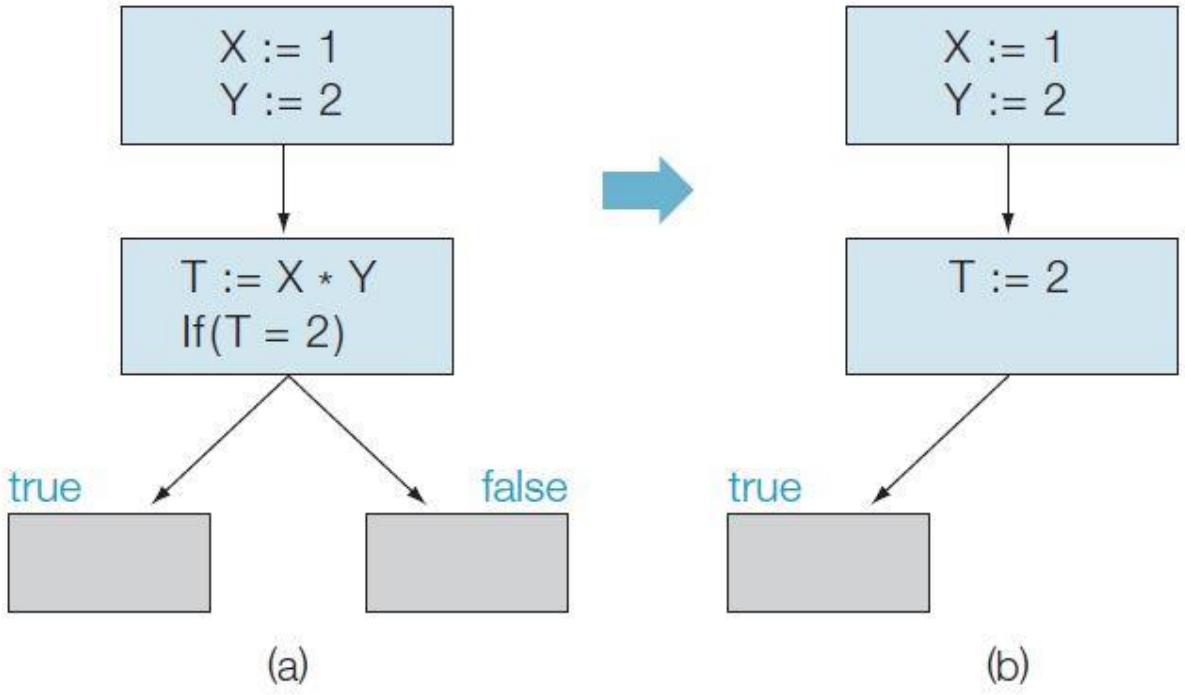


그림 10-8 도달 불가능한 코드 제거

10.3 최적화 기법

기계 종속적 최적화 기법

- 기계의 특성에 따라 아주 달라질 수 있는 기계 종속적 최적화 기법에는 중복된 LOAD 명령 어 제거, 효율적인 명령어 선택, 레지스터 할당과 배정, 명령어 스케줄링 등이 있다.

중복 명령어 제거

- 핍홀 최적화 기법에서 이미 설명했다.
- 중복되는 로드 명령문이나 저장 명령문을 제거함으로써 최적화하는 방법이다.

효율적인 명령어 선택

- 동일한 기능을 가진 더 효율적인 명령어로 대치함으로써 최적화하는 방법
- 소스코드 x = x + 10에 대한 명령어는 다음과 같다.
  - LOAD R1, x
  - ADD R1, 10
  - STORE x, R1
- 이와 같은 명령어에 대해 컴퓨터가 레지스터나 메모리에서 1을 증가시키는 INC(increase) 명령어를 가진다면 다음과 같은 하나의 명령어로 대치할 수 있다.
  - INC x

레지스터 할당과 배정

- 목적 코드의 생성에서 중요한 고려 대상 중의 하나는 레지스터를 어떻게 이용하느냐 하는 것이다. 자료가 레지스터에 들어 있으면 기억장치에 들어 있는 경우보다 액세스(access)가 빠르고 저장과 로드를 위한 명령이 필요 없어진다. 그러나 레지스터의 수는 극히 한정되어 있으므로 레지스터 할당 방법이 문제가 된다.
- 레지스터 할당 방법은 지역적 할당 방법과 전역적 할당 방법
  - 지역적 할당 방법은 프로그램을 기본 블록으로 분할했을 때 각 기본 블록 내에서 변수를 레지스터에 할당하는 방법
  - 전역적 할당 방법은 전체 프로시저를 고려하여 할당하는 방법
  - 레지스터 할당 방법은 그래프 착색(graph coloring) 방법으로 NP 완전(NP-complete) 문제이므로 경험적인 방법에 따라 할당하되, 전역적으로 잘 사용되는 자료는 되도록 레지스터에 넣어두는 편이 좋고, 지역적으로는 한 번 레지스터로 꺼내온 자료를 되도록 그곳에 보관하고 이용하는 편이 좋다.
  - 레지스터 할당 문제는 12장에서 좀 더 자세히 설명할 것이다.

10.3 최적화 기법

명령어 스케줄링

- 식의 연산 순서 변경에 의해 연산이 임시 결과(temporary result)를 저장(STORE)하거나 로드(LOAD)의 횟수를 최소로 함으로써 임시 변수 기억 공간을 절약할 뿐 아니라 효율적인 코드를 생산하는 것을 말한다.

[예제 10-23] 하나의 레지스터로 효율적인 명령어 만들기

- 산술식 A * B - C * D를 계산하는 데 하나의 레지스터 R0만을 이용할 수 있고, 일반적인 연산자 우선순위라고 가정하면 다음과 같이 8개의 명령을 갖는다. 식의 순서를 변경하여 효율적인 명령어를 만들어보자.
- [풀이]
- 원래대로 하면 다음과 같다.
- LOAD R0, A
- MULT R0, B
- STORE temp1, R0
- LOAD R0, C
- MULT R0, D
- STORE temp2, R0
- LOAD R0, temp1
- SUBT R0, temp2

10.3 최적화 기법

- A * B가 계산되기 전에 C * D를 먼저 계산하도록 연산 순서를 변경하면 다음과 같은 6개의 목적 코드를 생산할 수 있다.

  - LOAD R0, C
  - MULT R0, D
  - STORE temp1, R0
  - LOAD R0, A
  - MULT R0, B
  - SUBT R0, temp1